UNIVERSITÀ DEGLI STUDI DI UDINE

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

TESI DI LAUREA IN INFORMATICA

Algoritmi per la scelta del vicinato

CANDIDATO
Roberto Borelli

RELATORE
Prof. Agostino Dovier

CORRELATORE
Prof. Federico Fogolari

Anno accademico 2021-2022

Contatti dell'Istituto
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
https://www.dmif.uniud.it/

Ringraziamenti

Prima di procedere con la tesi e passare a tecnicismi, vorrei spendere giusto qualche riga per ringraziare dovutamente coloro che mi sono stati vicini in questo percorso ormai volto al termine. Innanzitutto vorrei ringraziare i miei relatori che si sono dimostrati sempre molto disponibili. Un sentito grazie va ai miei genitori che mi hanno consentito di svolgere al meglio questo percorso senza farmi mai mancare nulla. Un grazie alla mia ragazza che credendo in me, mi ha sempre sostenuto. Un grazie a tutti i miei amici che mi sono stati accanto in questi anni e un grazie ai miei compagni di corso con cui ho condiviso sessioni e notti di studio.

Sommario

Una delle operazioni di base usate in informatica è la ricerca entro un documento, sia esso testuale o multimediale; per questo scopo sono state proposte strutture dati e algoritmi per affrontare le varie situazioni efficientemente. Spesso non si è interessati alla soluzione esatta, ma ad una serie di risposte che più si avvicinano all'input; si pensi ad esempio ad una classica ricerca su un motore di ricerca: non si è interessati a trovare il sito che contenga esattamente tutte le parole digitate nell'esatto ordine, ma si vuole invece trovare il sito che in qualche modo sia *vicino* alla nostra ricerca. Questa è una prima intuizione del concetto di vicinato, oggetto di studio di questa tesi. Un altro esempio è il *content-based image indexing*: per cercare un'immagine in un ampio database, non si analizzerà l'immagine stessa ma si analizzeranno dei metadati come descrizione, titolo e altre informazioni. In quest'ultimo esempio risulta ancora più chiaro che non si vuole ricercare la *query* esatta, infatti sarebbe impossibile descrivere esattamente in linguaggio naturale un'immagine senza ambiguità, ma è possibile invece data una stringa di testo contenente qualche parola chiave, ricercare le immagini che in qualche modo (sfruttando i loro metadati) si avvicinano di più alle parole fornite. Il problema della determinazione del vicinato è un classico problema di geometria computazionale ampiamente studiato in letteratura. Le applicazioni pratiche e teoriche sono innumerevoli, oltre agli esempi già citati, prendono posto sistemi di classificazione, sistemi per compressione dei dati, sistemi predittivi, algoritmi di pattern recognition, stimatori per il calcolo dell'entropia.

Un primo esempio

Scendendo nel dettaglio, che si tratti di immagini, testi o numeri, gli elementi in considerazione saranno appartenenti ad un certo spazio con un certo numero di dimensioni.

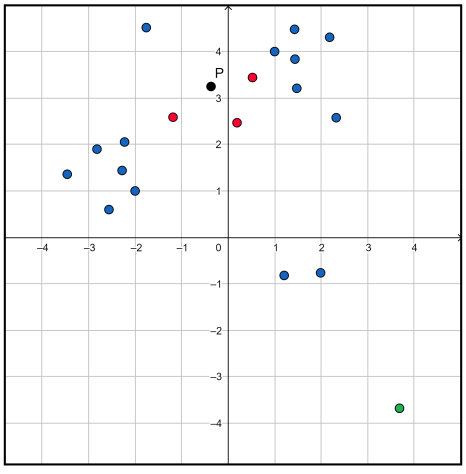


Figura 1: Esempio di vicinato per un punto P.

In figura 1 è rappresentato un insieme di 20 punti che si trovano in uno spazio bidimensionale. I punti colorati di rosso sono i 3 punti più vicini al punto P (colorato di nero). Per avere la certezza che i punti rossi siano effettivamente i 3 più vicini a P è sufficiente calcolare le distanze tra P e tutti gli altri punti e prendere i 3 punti a distanza minima. Quindi sono stati effettuati 19 calcoli delle distanze. Se ora volessimo calcolare i 3 vicini per ogni punto rappresentato, basterebbe reiterare 20 volte il processo descritto e quindi dovremmo calcolare ben $19 \times 20 = 380$ distanze. Se invece i punti fossero stati 10000 avremmo dovuto calcolare 99990000 distanze.

Sempre considerando la figura 1, sfruttando i nostri occhi e la mente umana, ci è subito chiaro che il punto in basso a destra (dipinto di verde) sicuramente sarà più distante rispetto a P dei punti rossi e sicuramente non farà parte dei 3 vicini per P. In sostanza, siamo riusciti a risparmiare un calcolo e, come abbiamo escluso il punto verde siamo immediatamente in grado, senza l'ausilio di nessuna formula, di escludere altri punti e di concentrarci solo sui punti del *vicinato* di P. Riassumendo dato un certo insieme di punti arbitrariamente grande e un punto P, per trovare i vicini di P non vorremmo dover analizzare tutte le distanze ma vorremmo essere in grado di escluderne la maggior parte per concentrarci invece sul vicinato di P.

Struttura della tesi

Questo documento racchiude i risultati principali e le rielaborazioni di molte ricerche che permettono di presentare il problema adeguatamente. Vengono discussi numerosi algoritmi ciascuno con il proprio pseudo codice e la relativa analisi delle prestazioni tramite risultati teorici ed empirici. Lo scopo degli algoritmi proposti sarà quello di evitare di ricercare il vicinato su tutti gli elementi ma al contrario si vorrà restringere la ricerca in una piccola regione di spazio opportunamente calcolata.

Nel **capitolo 1** viene data la definizione matematica del problema, viene analizzata la letteratura rilevante e le soluzioni proposte. Viene presentata una dimostrazione del limite inferiore di complessità che risulta di particolare importanza per permettere di misurare la bontà degli algoritmi che saranno esposti. Viene analizzato un algoritmo che risolve il problema direttamente con una ricerca esaustiva senza far uso di nessuna struttura dati particolare e viene inoltre analizzato un algoritmo semplice ed efficiente nel caso unidimensionale.

Nel **capitolo 2** si entra nel cuore del problema, vengono proposte due strutture dati ad-hoc e i relativi modi di operare. Queste strutture dati rappresentano entrambe (seppur in maniera diversa) delle generalizzazioni dei Binary Search Tree per spazi a più dimensioni. Per la prima struttura analizzata, il Quad-Tree, viene presentato l'algoritmo di Varadarajan [1]. Viene proposta inoltre una generalizzazione e prendendo ispirazione dall'algoritmo di Vaidya [2] vengono discusse delle euristiche per ottimizzare il tempo medio per la ricerca. Vengono studiati i K-D Tree in una versione migliorata rispetto a quella originariamente proposta da Bentley [3].

Nel **capitolo 3** si discutono soluzioni nel caso in cui lo spazio abbia delle condizioni periodiche. Questa situazione appare spesso nella stima dell'entropia e necessita di essere trattata opportunamente, gli algoritmi dei capitoli precedenti devono essere adeguatamente modificati e rianalizzati. La soluzione dei K-D Tree si adatta male al problema e viene quindi proposta la struttura dei VP-Tree che risolve il problema in un generico spazio metrico.

Nel **capitolo 4** si mostrano i risultati ottenuti dalla reale implementazione in linguaggio C degli algoritmi analizzati in precedenza. In particolare sono stati implementati i Quad Tree nella versione base e nella versione generalizzata, i K-D Tree, i K-D Tree nella variante per spazi con condizioni periodiche, i VP Tree e gli algoritmi Naive. Verrà presentato l'ambiente di test utilizzato e saranno commentati i grafici comparativi dei risultati sperimentali ottenuti.

Indice

1

Preliminari

In questo capitolo verrà dato un primo sguardo al problema, verrà definito formalmente e saranno analizzati alcuni approcci possibili alla risoluzione. Sarà discussa l'importanza di avere una soluzione efficiente attraverso la presentazione di applicazioni reali in cui viene utilizzato il calcolo del vicinato e, verranno quindi analizzati i limiti inferiori di complessità in tempo. Saranno infine presentati gli algoritmi *naive*.

1.1 Definizione del problema

Lo scopo di questa tesi è ricercare e confrontare varie soluzioni proposte per il problema della selezione del vicinato di ogni punto appartenente ad un insieme di input. Verrà ora definito il problema più formalmente, spesso riferito come *all-m-nearest-neighbors* [2] oppure come *M-N.N.* o ancora come *best-match-query* [4] e citato da Knuth in una forma più semplice come *the post-office problem* [5]. Siano dati un insieme V di n punti in uno spazio d-dimensionale e sia dato un numero intero $m < n$. Ogni punto $x \in V$ è quindi della forma $x = (x_1, \ldots, x_d)$. Si ricorda la definizione della distanza euclidea tra due punti $p, q \in V$:

$$d(p,q) = \sqrt{\sum_{i=1}^{d} (p_i - q_i)^2} \tag{1.1}$$

Problema 1.1 (ALL-MNN)**.** Per ogni punto $p \in V$ (con $|V| = n$) si vogliono trovare gli m punti più vicini a p secondo la distanza euclidea. Ovvero fissato p si vuole trovare un insieme di punti $N(p) \subseteq V \setminus \{p\}$ con $|N(p)| = m$ tale che ogni punto $x \in N(p)$ non sia più lontano a p rispetto ad ogni punto $y \in V \setminus N(p)$.

Il problema è generalizzabile anche ad altri tipi di distanze in domini diversi, ad esempio se i punti fossero stringhe appartenenti al linguaggio Σ^d dove Σ è un alfabeto di simboli, si potrebbe utilizzare la distanza di Hamming. In questo documento tuttavia ci si concentrerà su punti appartenenti allo spazio R^d e la distanza sarà misurata con la distanza euclidea salvo diversamente indicato.

Un'altra generalizzazione di ALL-MNN si ottiene considerando V un multiinsieme ovvero ammettendo che nei punti forniti in input ci siano delle ripetizioni. Questa nuova formulazione sarà applicabile a tutti gli algoritmi che saranno proposti nel seguito senza variazioni di strategie utilizzate e senza variazione di complessità in tempo e spazio.

Il problema è chiaramente decidibile e una soluzione banale di complessità quadratica è mostrata nell'algoritmo 2 nella sezione 1.3. Si noti che come spesso accade stiamo parlando di un problema di facile definizione ma la cui soluzione efficiente non è altrettanto scontata.

1.1.1 Tassonomia e lavori correlati

Il problema è stato ampiamente studiato in letteratura nel corso degli anni ma tuttavia non esiste ancora una soluzione che sia adatta ad ogni contesto e quindi ad ora si deve accettare un qualche *trade-off*. Alcune soluzioni riescono ad abbattere il limite dell' $O(n^2)$ dell'algoritmo 2 (che come sarà visto in sezione 1.3 è dovuto al calcolo di tutte le distanze) ma, nascondono tramite la notazione asintotica una costante dipendente da d che, se viene valutata esplicitamente, nel migliore dei casi fa degenerare l'algoritmo in questione in una ricerca di complessità quadratica e, nel peggiore dei casi cresce esponenzialmente con il numero di dimensioni portando alla completa inapplicabilità della soluzione trovata, anche per valori di d contenuti.

Innanzitutto è possibile dividere gli algoritmi proposti tra quelli in cui considerano problemi più semplici.

- Fissato un punto restituire il punto più vicino.
- Fissato un punto restituire gli m vicini.

Questi algoritmi si possono poi richiamare n volte per ottenere una soluzione al problema di nostra considerazione.

Ci sono poi algoritmi che prendono in considerazione fin dall'inizio che si è interessati ad ottenere una soluzione per ciascuno degli n punti. Alcune delle strade possibili possono essere le seguenti:

- Algoritmi che cercano di partizionare lo spazio con una struttura ad albero costruendo una sorta di indice. È questo il caso delle strutture Quad Tree [1, 6] (nel caso specifico $d = 2$), K-D Tree [3, 4, 7], VP Tree [8, 9, 10, 11, 12] e Ball Tree [13].
- Algoritmi approssimati, nel caso in cui $m = 1$ si permette che l'algoritmo restituisca punti che siano al massimo c volte più distanti dal punto più vicino. Tra questi si distinguono i metodi che utilizzano:
  - Grafi di prossimità come HNSW [14].
  - Locality sensitive hashing. In questo caso l'idea a differenza dell' hashing crittografico (in cui si cerca il più possibile di minimizzare le collisioni) è quella di massimizzare le collisioni se due punti sono simili. Formalmente, siano $c > 1$ un fattore di approssimazione opportunamente scelto, P_1 e P_2 due valori di probabilità e $R > 0$ una soglia. Una famiglia LHS è una famiglia di funzioni $h : V \to S$, dove S è lo spazio dei bucket e dove per ogni punto $p, q \in V$ e per ogni funzione di hash h valgono:
    * Se $d(p, q) \leq R$ allora $h(p) = h(q)$ con probabilità almeno P_1.
    * Se $d(p, q) \geq cR$ allora $h(p) = h(q)$ con probabilità al massimo P_2.
- Algoritmi che cercano di sfruttare la propagazione della vicinanza. Intuitivamente "Se io sono vicino a te, probabilmente i tuoi vicini potrebbero essere anche miei vicini".

Vaidya [2] nel 1989 propone un algoritmo $O(n\log(n))$ nel caso in cui $m = 1$ (ovvero se per ogni punto si è interessati solo al punto più vicino) e generalizzabile al caso generico con complessità $O(mn\log(n))$ basata sui *box* o *ipercubi* e sul loro partizionamento ad ogni passo. Clarkson [15] propone un algoritmo randomizzato e utilizza una tecnica simile basata sui *box* ma lo schema per dividerli è sostanzialmente diverso.

1.1.2 Limiti inferiori

Si consideri ora il problema più semplice in cui $m = 1$ e $d = 1$ e si consideri il caso in cui V sia un multiinsieme ovvero ci possano essere delle ripetizioni tra i punti. Risulta di estrema importanza limitare il problema inferiormente. Nella sezione precedente si è visto che sono stati proposti algoritmi $O(n\log(n))$, ma si può fare di meglio? Esisterà mai un algoritmo $O(n)$?

Per ragionare sulla complessità intrinseca del problema bisogna scegliere un modello di computazione. Un modello specifica le operazioni primitive ammesse ed il loro costo.

Un noto modello di computazione (con cui si possono codificare gli algoritmi che discuteremo nelle sezioni successive) è l'*algebraic decision tree*. In questo modello gli algoritmi sono rappresentati da alberi e ogni nodo dell'albero ha la forma di una comparazione

$$f(input) : 0 \tag{1.2}$$

dove f è un polinomio. Un albero di decisione T di ordine k per testare se $x \in W$ è un albero di decisione, dove le funzioni ammesse sono polinomiali di grado al più k e ogni foglia contiene la risposta *yes* o *no*, tale che per ogni $x \in R^d$, T decide correttamente se $x \in W$ [16]. Data un'istanza del problema, una computazione è un cammino dalla radice ad una foglia; per dimostrare i limiti inferiori si tenta di limitare l'altezza massima dell'albero ossia si limita la lunghezza della computazione. Nell'*algebraic decision tree* non è ammesso nessun calcolo che sfrutti una qualche forma di randomizzazione.

Teorema 1.1. Il problema 1.1 ha un limite inferiore di $\Omega(n\log(n))$ nell'*algebraic decision tree model*.

Di seguito viene presentata una dimostrazione per il teorema 1.1. Questo lower bound è da anni noto in letteratura, ad esempio è stato citato da Vaidya [15]; tuttavia, al meglio delle mie ricerche, la dimostrazione che propongo non è stata riportata in nessuna pubblicazione.

La dimostrazione si basa sulla nozione di riducibilità proveniente dalla teoria della complessità computazionale. Siano A e B due problemi decisionali, informalmente si vuole dire che se B si riduce ad A ($B \preceq A$), allora A è almeno difficile quanto il problema B. Si dice che B si riduce ad A se esiste una trasformazione R che per ogni input x per il problema B, produce un input *equivalente* per il problema A, dove *equivalente* significa che x è un'*istanza-yes* di B se e solo se $R(x)$ è un'*istanza-yes* di A.

Per costruire un algoritmo che decide B si può pertanto utilizzare un algoritmo che decide A: dato un input x per B, lo si trasforma tramite R in $R(x)$, si esegue l'algoritmo che risolve A con input $R(x)$ e si ottiene la risposta attesa. La seguente figura aiuta a visualizzare il processo.

Input per B		Input per A		Algoritmo per A		Risposta
x	$\rightarrow$	$R(x)$	$\rightarrow$	ALG_A	$\rightarrow$	$ALG_A(R(X)) = yes/no$

Viene inoltre chiesto che la trasformazione R sia calcolabile efficientemente. In questo modo se B necessita di algoritmi inefficienti, lo stesso varrà per A mentre se A può essere deciso efficientemente, la proprietà positiva ricadrà su B. Viene ora data la definizione formale di riducibilità.

Definizione 1.1 (Riducibilità)**.** Siano Σ_B e Σ_A due alfabeti. Dati due linguaggi $L_B \subseteq \Sigma_B^*$ ed $L_A \subseteq \Sigma_A^*$ si dice che L_B si riduce a L_A ($L_B \preceq L_A$) se esiste una funzione $R : \Sigma_B^* \to \Sigma_A^*$ tale che:

1. R è calcolabile da una macchina di Turing deterministica in spazio $O(\log n)$
2. $\forall x \in \Sigma_B^*\ x \in L_B \iff R(x) \in L_A$

R viene chiamata una *riduzione* da L_B a L_A.

Si può dimostrare (si veda ad esempio [17]) che se R è calcolata da una macchina di Turing M, allora M termina in un numero polinomiale di passi.

Come anticipato, nella dimostrazione del teorema 1.1 saranno effettuate delle riduzioni con i problemi di seguito definiti.

Problema 1.2 (closest-pair)**.** Data una lista V di n punti, si determini la coppia di punti (p, q) a distanza minima.

Problema 1.3 (element-uniqueness)**.** Data una lista V di n punti, si dica se gli elementi sono a due a due distinti.

Si noti che:

- I 3 problemi 1.2, 1.3 e 1.1 (ristretto al caso con $d = 1$ e $m = 1$) hanno lo stesso input (ossia un multiinsieme di n punti unidimensionali cioè una lista non ordinata di punti eventualmente con ripetizioni).
- *element-uniqueness* è di tipo decisionale.
- *closest-pair* e ALL-1NN sono di tipo funzionale.

Risulta quindi necessario definire più formalmente i problemi funzionali e introdurre una variante della relazione di riducibilità per trattarli adeguatamente.

Definizione 1.2 (Relazione decidibile polinomialmente [17])**.** Sia $R \subseteq \Sigma^* \times \Sigma^*$ una relazione binaria. R si dice polinomialmente decidibile se esiste una macchina di Turing deterministica che decide il linguaggio $\{x; y : (x, y) \in R\}$.

Definizione 1.3 (Relazione polinomialmente bilanciata [17])**.** Sia $R \subseteq \Sigma^* \times \Sigma^*$ una relazione binaria. R si dice polinomialmente bilanciata se $(x, y) \in R \implies |y| \leq |x|^k$ per qualche $k \geq 1$.

Definizione 1.4 (Problema funzionale associato a L [17])**.** Sia L un linguaggio in **NP**. Allora esiste (si veda la proposizione 9.1 di [17]) una relazione R_L decidibile polinomialmente e polinomialmente bilanciata tale che per ogni stringa x:

$$\exists y R_L(x, y) \iff x \in L \tag{1.3}$$

Il problema funzionale associato a L, denotato con FL è il seguente problema computazionale: Dato x, trovare una stringa y tale che $R_L(x, y)$. Se una tale stringa non esiste, restituire *no*.

Definizione 1.5 (Riducibilità tra problemi funzionali [17])**.** Dati due problemi A e B di tipo funzionale. Si dice che B si riduce ad A (in breve $B \preceq A$) se esistono delle funzioni tra stringhe R ed S calcolabili in spazio logaritmico tali che per ogni stringa x e z valgono:

1. x è istanza di $B \implies R(x)$ è istanza di A.

2. z è l'output corretto di $R(x) \implies S(z)$ è l'output corretto di x.

R quindi svolge lo stesso ruolo della R della definizione 1.1 ossia trasforma l'input tra due problemi; la novità della definizione 1.5 è S che è la trasformazione per l'output. Si noti che viene richiesto che entrambe le funzioni siano calcolabili efficientemente (in spazio logaritmico).

Analogamente a quanto visto per i problemi decisionali, per costruire un algoritmo che risolve B si può utilizzare un algoritmo che risolve A: dato un input x per B, lo si trasforma tramite R in $R(x)$, si esegue l'algoritmo che risolve A con input $R(x)$ e si ottiene come output z. La risposta attesa per B con input x è quindi $S(z)$. Il processo si può visualizzare dalla seguente figura:

Input per B		Input per A		Algoritmo per A		Risposta per A		Risposta per B
x	$\rightarrow$	$R(x)$	$\rightarrow$	ALG_A	$\rightarrow$	$ALG_A(R(X)) = z$	$\rightarrow$	$S(z)$

Dunque se $B \preceq A$, il tempo T_B per calcolare la soluzione per B tramite un algoritmo che risolve A (in tempo T_{ALG_A}) vale:

$$T_B = T_R + T_{ALG_A} + T_S \tag{1.4}$$

dove T_R e T_S denotano i tempi per calcolare rispettivamente le trasformazioni R ed S.

Diremo che $B \preceq A$ in tempo $O(f(n))$ se le trasformazioni R ed S sono calcolabili da una macchina di Turing deterministica che opera in tempo minore o uguale a $f(n)$.

Ora verranno presentati due lemmi che saranno utilizzati nella dimostrazione al teorema 1.1. Il lemma 1.2 formalizza un risultato già noto in letteratura.

Lemma 1.1 (*closest-pair* $\preceq$ ALL-1NN)**.** Il problema *closest-pair* si riduce a ALL-1NN in tempo $O(n)$.

Dimostrazione. I due problemi hanno lo stesso input quindi vi è solo la trasformazione per l'output. Data una soluzione di ALL-1NN nella forma $(x_1, \ldots, x_n, y_1, \ldots, y_n) \in R^{2n}$ dove gli x_i sono i punti di input e gli y_i sono i relativi punti più vicini, ossia tali che $\forall i, j \in \{1, \ldots, n\}$ $(d(x_i, y_i) \leq d(x_i, x_j))$ con $i \neq j$, si calcola in tempo $\Theta(n)$ una soluzione per il problema *closest-pair* come segue:

1. Si calcoli in tempo $\Theta(n)$ il vettore delle distanze $d_{xy} = (d_1, \ldots, d_n) \in R^n$ in cui per ogni i, $d_i = d(x_i, y_i)$.

2. Si trovi in tempo $\Theta(n)$, l'indice j dell'elemento più piccolo nel vettore d_{xy}.

3. La coppia (x_j, y_j) è la soluzione per *closest-pair*.

Si noti che tale procedimento è facilmente implementabile in spazio logaritmico.

Quindi il tempo per risolvere *closest-pair* tramite un algoritmo che risolve ALL-1NN vale:

$$T_{\textit{closest-pair}}(n) = \Theta(1) + T_{\text{ALL-1NN}}(n) + \Theta(n) \tag{1.5}$$

Si noti infine che se il limite inferiore in tempo per *closest-pair* è $\Omega(f(n))$ allora il limite inferiore in tempo per ALL-1NN è $\Omega(f(n)) + \Theta(n)$. □

Lemma 1.2 (*element-uniqueness* $\preceq$ *closest-pair*)**.** Il problema *element-uniqueness* si riduce a *closest-pair* in tempo $O(1)$.

Dimostrazione. I due problemi hanno lo stesso input quindi vi è solo la trasformazione per l'output. Data una soluzione al problema *closest-pair* nella forma $(x, y) \in R^2$, si calcola una soluzione per *element-uniqueness* in tempo $\Theta(1)$ come segue:

1. Si calcoli $d(x, y)$.
2. Se la distanza è 0, si restituisca *no*, altrimenti si restituisca *yes*.

Il tempo per risolvere *element-uniqueness* tramite la procedura descritta è:

$$T_{element\text{-}uniqueness}(n) = \Theta(1) + T_{closest\text{-}pair}(n) + \Theta(1) \tag{1.6}$$

da cui si ricava che se il limite inferiore per *element-uniqueness* è $\Omega(g(n))$ allora il limite inferiore in tempo per *closest-pair* è $\Omega(g(n)) + \Theta(1)$.

□

Teorema 1.2. Il problema 1.3 ha un lower-bound di $\Omega(n \log(n))$ nell'*algebraic decision tree model*.

Dimostrazione. Si veda il lavoro di Ben-Or [16] che dimostra il lower bound nel caso in cui gli elementi siano appartenenti ad R. Lubiw [18] dimostra che questo limite vale anche restringendosi ai numeri interi. □

Si è finalmente pronti per dimostrare il teorema oggetto di questa sezione ovvero il teorema 1.1.

Dimostrazione teorema 1.1. Il problema *closest-pair* si riduce a ALL-1NN in tempo $O(n)$ (si veda il lemma 1.1). È noto che il problema *element-uniqueness* si riduce a *closest-pair* in tempo $O(1)$ (si veda il lemma 1.2), inoltre è stato dimostrato [18] che *element-uniqueness* ha un lower bound di $\Omega(n \log(n))$ nell'*algebraic decision tree model* (si veda il teorema 1.2). La catena delle riduzioni è:

$$\textit{element-uniqueness} \preceq \textit{closest-pair} \preceq \text{ALL-1NN} \tag{1.7}$$

Dunque da un algoritmo che risolve ALL-1NN che opera in tempo $T(n)$ si è in grado di costruire un algoritmo che risolve *element-uniqueness* che opera in tempo $T(n) + \Theta(n)$. Se per assurdo $T(n) = o(n \log(n))$ potremmo risolvere *element-uniqueness* in $o(n \log(n))$ che contraddice il teorema 1.2. Si ricava che il limite inferiore per ALL-1NN è $\Omega(n \log(n))$ □

Corollario 1.1. Il limite inferiore per ALL-MNN dove d e $m < n$ sono arbitrari è $\Omega(n \log(n))$.

Dimostrazione. Basti notare che ALL-1NN è un caso particolare del problema 1.1 in cui d ed m sono arbitrari. □

Si ponga attenzione sul fatto che con questa serie di dimostrazioni, non viene asserito nulla su altri modelli di computazione ad esempio, il *quantum-computing* o il *cell-probe-model*, dove il teorema 1.2 non vale più e pertanto un algoritmo lineare potrebbe o meno esistere.

1.2 Il contesto

Vengono ora toccati e presentati più nel dettaglio alcune applicazioni, in particolare mostreremo come sfruttare gli algoritmi per il vicinato per il calcolo dell'entropia e come sfruttarli invece nel contesto del machine learning. I problemi di seguito esposti rendono ancora più chiara l'importanza degli algoritmi che saranno descritti nei capitoli 2 e 3 e l'importanza della ricerca di soluzioni efficienti.

1.2.1 L'entropia

In teoria dell'informazione un concetto fondamentale è quello di quantificare l'informazione in eventi e variabili causali. L'intuizione principale è che eventi che si verificano con bassa probabilità portano 'tanta' informazione ed eventi che si verificano con alta probabilità portano 'poca' informazione. Questa intuizione è di seguito formalizzata.

Definizione 1.6 (Informazione)**.** Sia E un evento, l'informazione portata da E vale:

$$I(E) = -\log_2 p(E) \tag{1.8}$$

Con $p(E) = 0.5$ l'informazione vale 1, ovvero serve 1 bit per codificare l'evento. Si vuole ora calcolare il numero medio di bit per rappresentare ottimamente un evento dalla distribuzione di una variabile casuale.

Definizione 1.7 (Shannon Entropy)**.** Sia X una variabile casuale discreta con supporto discreto $A = \{a_1, \ldots, a_k\}$. L'entropia di X è definita come:

$$H(X) = -\sum_{x \in A} p(x) \log(p(x)) \tag{1.9}$$

Equivalentemente l'entropia può essere definita come valore atteso della variabile casuale dell'informazione di X.

$$H(X) = E(I(X)) \tag{1.10}$$

Se X ha solo un evento certo ovvero con probabilità 1, l'entropia vale 0. Se tutti gli eventi sono equamente probabili l'entropia è massima e vale $-k(\frac{1}{k}\log(\frac{1}{k})) = \log(k)$.

Vengono ora definite l'entropia congiunta e l'entropia condizionale.

Definizione 1.8 (Entropia congiunta)**.** Siano X e Y variabili casuali discrete.

$$H(X,Y) = -\sum_{x,y} p(x,y) \log(p(x,y)) \tag{1.11}$$

L'entropia condizionale misura quanta incertezza/informazione rimane su una variabile X quando conosciamo Y.

Definizione 1.9 (Entropia condizionale)**.** Siano X e Y variabili casuali discrete.

$$H(X|Y) = -\sum_{x,y} p(x,y) \log(p(x|y)) \tag{1.12}$$

La mutua informazione tra X ed Y misura l'informazione che X e Y condividono.

Definizione 1.10 (Mutua informazione)**.** Siano X e Y variabili casuali discrete.

$$\begin{aligned} M(X,Y) &= -\sum_{x,y} p(x,y)\log(\tfrac{p(x,y)}{p(x)p(y)}) \\ &= H(X) - H(X|Y) \\ &= H(Y) - H(Y|X) \\ &= H(X) + H(Y) - H(X,Y) \end{aligned} \tag{1.13}$$

Se le variabili casuali sono indipendenti conoscere X non dà informazione su Y, pertanto $M(X,Y) = 0$. Le precedenti definizioni si possono dare similmente anche nel caso di variabili casuali continue.

La mutua informazione e altre misure più complesse come la *Transfer Entropy* e la *Global Transfer Entropy* sono ampiamente utilizzate nella comunità scientifica ad esempio nello studio della fisica di sistemi fondamentali come l'*ising model* [19].

Risulta chiaro che conoscendo la distribuzione di probabilità, queste quantità possano essere calcolate direttamente ed esattamente, ma nei contesti applicativi reali la distribuzione non è mai nota e va quindi stimata. Si ottengono quindi degli *Entropy Estimators* [19]. Uno dei noti approcci è quello di usare gli stimatori *nearest-neighbour*, l'idea è quella di stimare la massa di probabilità attorno a ciascun punto della distribuzione, usando gli m vicini del punto stesso. È quindi di reale importanza teorica e pratica trovare una soluzione quanto più possibile efficiente al problema 1.1.

1.2.2 Machine learning

Un algoritmo usato sia per risolvere problemi di classificazione sia problemi di regressione è *kNN*. Dato un insieme di n punti che costituiscono il *traning-set*, si vogliono stimare i valori di nuovi punti in ingresso considerando i k punti più vicini. La differenza rispetto al problema 1.1 è che in questo caso i punti di cui si vogliono ottenere i vicini non sono gli stessi punti forniti in input ma sono nuovi punti. L'algoritmo di Vaidya [2] in questo caso non risulta applicabile, ma le altre soluzioni al problema 1.1 che saranno discusse nelle sezioni successive sono facilmente applicabili per costruire un algoritmo *kNN* efficiente. Dato il *training-set* costituito da n elementi del tipo $\langle x^{[i]}, y^{[i]}\rangle$ dove $x^{[i]}$ è della forma $x^{[i]} = (x_1^{[i]}, \ldots, x_d^{[i]})$ e dato un nuovo punto $q = (q_1, \ldots, q_d)$ l'obiettivo è quello di stimare y_q: nel caso della classificazione verrà stimato come la moda tra gli $y^{[i]}$ dei k punti più vicini mentre nel caso della regressione al posto della moda si utilizzerà la media.

Più nel dettaglio, nel problema di classificazione gli $y^{[i]}$ rappresentano delle *label* (o classi) appartenenti ad un insieme finito di classi che caratterizzano i relativi $x^{[i]}$. L'obiettivo del classificatore *kNN* è quello di stimare per un punto q la relativa classe: per il calcolo si considera l'insieme dei k vicini di q, $q(k) = \left\{x^{[i_{q_1}]}, \ldots, x^{[i_{q_k}]}\right\}$ dove ogni $x^{[i_{q_j}]}$ è appartenente al *training-set* e ogni i_{q_j} è compreso tra 1 e n, e si calcola in seguito la moda delle *label* associate $y^{[i_{q_1}]}, \ldots, y^{[i_{q_k}]}$ ovvero si prende quella che occorre più volte. Si noti che a differenza del *majority-voting* non si richiede che la label compaia più del 50% delle volte.

Passando alla regressione, gli $y^{[i]}$ non sono più delle classi discrete ma sono dei valori reali e lo scopo del regressore è quello di stimare il valore reale associato a q. Si procede come per il classificatore, calcolando l'insieme $q(k)$, ma si stimerà poi il valore di output come la media tra $y^{[i_{q_1}]}, \ldots, y^{[i_{q_k}]}$.

In entrambi i casi, k va scelto accuratamente e può cambiare le performance del modello. In particolare scegliendo un k basso il rischio è quello che la stima sia instabile e abbia alta varianza, infatti la predizione in una data regione sarà totalmente dipendente dall'unico punto considerato. Il principale vantaggio di questo metodo consiste nel non essere parametrico, ovvero non c'è nessuna assunzione sulla forma dell'ipotesi e risulta quindi molto flessibile, al contrario altri modelli come la *linear regression* assumono una forma per l'ipotesi che in alcuni casi può non essere appropriata.

Il classificatore *kNN* e il regressore *kNN* [20] sono attualmente implementati nella nota libreria di machine learning *scikit-learn*, in particolare nelle rispettive classi `sklearn.neighbors.KNeighborsClassifier` [21] e `sklearn.neighbors.KNeighborsRegressor` [22]. In entrambi i casi *scikit-learn* da la possibilità tramite il parametro `algorithm`, che può contenere un valore tra `ball_tree`, `kd_tree`, `brute`, di scegliere l'algoritmo sottostante. Altri parametri permettono di modificare il numero di vicini k da prendere in considerazione (parametro `n_neighbors`) e le dimensioni delle foglie nel caso in cui l'algoritmo scelto sia un albero.

1.3 Soluzioni banali

Il problema descritto nei paragrafi precedenti è chiaramente decidibile e di complessità polinomiale e qui ne si dà la prova presentando l'algoritmo 2. Viene inoltre proposta una soluzione efficiente per il caso unidimensionale; la restrizione ad una sola dimensione permette di affrontare il problema elegantemente con una soluzione $\Theta(n\log(n)+nm)$.

1.3.1 Caso unidimensionale

Si consideri uno spazio unidimensionale (ovvero $d=1$), tutti i punti stanno su una retta. Ordinando tutti i punti di input, dato un punto p, gli m vicini di p sono adiacenti tra di loro nell'array ordinato. Ad esempio se p si trova in posizione i dell'array *points*, il vicino *p'* in posizione *i'* a minor distanza da p è definito come:

$$\begin{aligned} &i_{left} = i-1 \\ &i_{right} = i+1 \\ &i' = \arg\min_{j \in \{i_{left}, i_{right}\}} \{dist(points[j], p)\} \\ &p' = points[i'] \end{aligned} \tag{1.14}$$

Similmente si possono definire gli altri $m-1$ vicini di p. Detto *p"* in posizione *i"* il secondo punto più vicino a p, si ha:

$$\begin{aligned} &i'_{left} = \begin{cases} i'-1 & \text{se } i' = i_{left} \\ i_{left} & \text{altrimenti} \end{cases} \\ &i'_{right} = \begin{cases} i'+1 & \text{se } i' = i_{right} \\ i_{right} & \text{altrimenti} \end{cases} \\ &i'' = \arg\min_{j \in \{i'_{left}, i'_{right}\}} \{dist(points[j], p)\} \\ &p'' = points[i''] \end{aligned} \tag{1.15}$$

Chiaramente le equazioni 1.14 e 1.15 valgono solo se gli indici non sforano i limiti degli array, cioè devono essere numeri interi compresi tra 1 ed n. Ad esempio considerando p in posizione $i=n$, si avrebbe che i_{right} non può essere definito come nella formula 1.14 e in questo caso gli m vicini di p sarebbero banalmente $points[n-m], points[n-m+1], \ldots, points[n-1]$. Per ovviare a questo problema basta ridefinire la funzione *dist* come segue:

$$dist(p,q) = \begin{cases} \infty & \text{Se uno tra } p \text{ e } q \text{ non è definito} \\ |p-q| & \text{altrimenti} \end{cases} \tag{1.16}$$

L'algoritmo 1 codifica queste idee.

Analizzando l'algoritmo 1 si ha che la complessità del sorting vale $\Theta(n\log(n))$, le distanze vengono calcolate in tempo costante quindi la costruzione degli insiemi *sol* ha complessità $\Theta(nm)$. Si conclude che l'algoritmo 1 ha complessità in tempo $\Theta(n\log(n)+nm)$ ed è pertanto una soluzione ottimale al problema.

Algorithm 1 ALL-MNN caso banale con d=1

```
procedure ALL-MNN(points, n, m)
    points ← SORT(points)
    sol ← NEW-ARRAY(n)
    for i ← 1 to n do
        p ← points[i]
        i_right ← i + 1
        i_left ← i − 1
        j ← 1
        while j ≤ m do
            ld ← CALC-DISTANCE(points, i, i_left, n)
            rd ← CALC-DISTANCE(points, i, i_right, n)
            if ld < rd then
                sol[i] ← sol[i] ∪ points[i_left]
                i_left ← i_left − 1
            else
                sol[i] ← sol[i] ∪ points[i_right]
                i_right ← i_right + 1
            end if
            j ← j + 1
        end while
    end for
    return sol
end procedure
procedure CALC-DISTANCE(points, i, j, n)
    if j ≥ n + 1 or j ≤ 0 then
        return ∞
    else
        return |points[i] − points[j]|
    end if
end procedure
```

Si noti che il termine nm può diventare più grande di $n\log(n)$ con opportuni valori, ma si tratta ad esempio della restituzione dell'output e quindi ci sarà sempre in ogni algoritmo che risolve il problema 1.1. In altre parole nm rappresenta in questo caso la linearità sull'input.

1.3.2 Caso d-dimensionale

Con d arbitrario non si può più adottare l'approccio che ha funzionato bene nella situazione precedente. La soluzione più banale possibile quindi, consiste nel calcolare tutte le distanze tra punti e per ogni punto selezionare quindi gli m più vicini.

Algorithm 2 ALL-MNN soluzione banale nel caso d-dimensionale

```
procedure ALL-MNN(points, n, m, d)
    sol ← NEW-ARRAY(n)
    for i ← 1 to n do
        p ← points[i]
        d ← CALC-DISTANCES(points, i)
        d[i] ← null
        d_max ← LINEAR-SELECT(d, m)
        j ← 0
        while |sol[i]| ≤ m do
            if d[j] ≤ d_max and d[j] ≠ null then
                sol[i] ← sol[i] ∪ points[j]
            end if
            j ← j + 1
        end while
    end for
    return sol
end procedure
```

L'algoritmo proposto opera sotto l'assunzione che fissato un punto p, tra gli m vicini di p ve ne sia esattamente uno a distanza massima. Per ogni punto p l'algoritmo 2 calcola il vettore delle distanze contenente tutte le distanze con gli altri punti, invoca la procedura `LINEAR-SELECT` che restituisce in tempo lineare la distanza che finirebbe in posizione m-ima se il vettore delle distanze fosse ordinato e infine considera come soluzioni per p tutti i punti che hanno distanza minore o uguale a quella selezionata e supponendo quindi che la distanza dell'm-imo punto più vicino a p sia unica (tra le distanze che si trovano nel vettore delle distanze), la soluzione contiene esattamente gli m punti più vicini a p. La funzione `LINEAR-SELECT` può essere implementata dall' algoritmo median of medians [23].

Analizzando la complessità, l'algoritmo calcola n^2 distanze anche se con qualche banale accorgimento se ne possono calcolare solo $\frac{n(n+1)}{2} - n$ (si sfrutta il fatto che $d(p,p) = 0$) ma che restano comunque $O(n^2)$. Una chiamata alla procedura `CALC-DISTANCES` prende tempo $\Theta(nd)$, complessivamente le esecuzioni dell'algoritmo `LINEAR-SELECT` prendono tempo $O(n^2)$ e per ogni punto il ciclo *while* in cui viene costruita la soluzione per p prende tempo $O(n)$. Complessivamente l'esecuzione dell'algoritmo 2 prende tempo $O(dn^2)$.

L'ipotesi di cui sopra, non rispetta tuttavia la definizione del problema 1.1. Si pensi ad esempio alla situazione in cui fissato un punto p, esistano m punti a distanza d_m e $m-1$ punti a distanze $d_1, \ldots, d_{m-1}$ tutte minori (strettamente) di d_m. Dipendentemente dall'ordine in cui vengono scanditi i punti nel ciclo

while, può darsi il caso in cui l'algoritmo restituisca tutti e soli i punti a distanza d_m, che è quindi una soluzione completamente sbagliata.

L'ipotesi può essere rimossa facilmente operando come segue:

1. Si contino quante occorrenze di d_{max} si trovano in d vettore delle distanze. Sia j questo numero. Si noti che $j >= 1$.

2. Con una prima scansione si aggiungano al vettore delle soluzioni i k punti con distanza **strettamente** minore a d_{max}.

3. Con una seconda scansione si aggiungano al vettore delle soluzioni $m - k$ punti scelti arbitrariamente tra i j punti a distanza d_{max}.

Si noti che questa variante dell'algoritmo 2 che rispetta la definizione 1.1 non subisce variazioni nella complessità asintotica. Purtroppo in alcuni contesti applicativi questa soluzione pur essendo polinomiale, risulta ancora troppo inefficiente. Sarà analizzato in seguito se sia possibile o meno abbattere questo limite dal punto di vista teorico e pratico.

2

Partizionamento dello spazio

In questo capitolo analizzeremo delle soluzioni basate sul concetto di partizionamento dello spazio più complesse ma anche più efficienti di quelle viste nel capitolo precedente. Vedremo inoltre che queste soluzioni soffrono del problema *curse of dimensionality* ovvero, vedremo soluzioni che funzionano bene con un d sufficientemente piccolo ma che iniziano a peggiorare anche esponenzialmente al crescere di esso.

2.1 Quad Tree

I Quad Tree sono una struttura dati originariamente proposta da Finkel e Bentley [6] per memorizzare e ricercare informazioni che hanno una chiave composta.

Si consideri ora il caso bidimensionale con $m = 1$ e $d = 2$ del problema 1.1. La più semplice struttura per partizionare efficientemente lo spazio e risolvere `ALL-1NN` è per l'appunto quella dei Quad Tree.

Si supponga inoltre che lo *spread* tra i punti in input, ovvero il rapporto tra la più grande e la più piccola distanza tra i punti in V, sia limitato da un fattore polinomiale in n (ad esempio n^7).

Partendo con un approccio top-dow, lo pseudo codice della procedura generale che risolve `ALL-1NN` è descritto dal listato 3.

Algorithm 3 ALL-1NN caso bidimensionale utilizzando i Quad Tree

```
procedure ALL-1NN(points, n)
    T ← BUILD-QUAD-TREE(points, n)
    CALCULATE-HEIGHT(T)
    sol ← NEW-ARRAY(n)
    for i ← 1 to n do
        sol[i] ← QUERY(points[i], T)
    end for
    return sol
end procedure
```

Per prima cosa viene costruito il Quad Tree a partire dall'insieme di punti preso in input. Dell'albero ottenuto viene calcolata l'altezza e possiamo ipotizzare che questa informazione sia memorizzata nel nodo radice e verrà sfruttata dall'algoritmo di `QUERY`. Per ogni punto p si effettua quindi una chiamata alla procedura `QUERY` che restituisce il punto più vicino a p. L'algoritmo 3 termina quando è stata calcolata

la soluzione per ogni punto. Nei paragrafi successivi si entrerà nel cuore delle procedure, in particolare nel paragrafo 2.1.2 si presenterà un modo per costruire efficientemente l'albero e nel paragrafo 2.1.4 si analizzerà l'algoritmo di ricerca.

2.1.1 La struttura dati

Esistono numerose varianti dei Quad Tree, la definizione che sarà adottata in questo caso è la seguente.

Definizione 2.1 (Quad Tree su V insieme di punti)**.** Un Quad Tree è un albero tale che:

1. Ogni nodo ha al più 4 figli, precisamente un nodo interno ne ha 4 e un nodo foglia 0.
2. Ogni nodo rappresenta un quadrato[1], una porzione di spazio bidimensionale.
3. Ogni nodo foglia contiene 1 o 0 punti.
4. Ogni nodo interno e ogni nodo foglia con un punto, hanno un punto rappresentante tra i punti che contengono.
5. Il rappresentante di ogni foglia con un punto è il punto stesso.
6. L'albero rappresenta tutti i punti in V che sono memorizzati nelle foglie.

Oltre al punto 4 della definizione 2.1, non viene posta alcuna condizione su come scegliere il punto rappresentante nei nodi non foglia. Durante l'algoritmo di ricerca che verrà analizzato in sezione 2.1.4, la distanza tra il punto di query q e il punto rappresentante di un dato nodo non foglia servirà per limitare la distanza entro cui trovare il punto più vicino a q e tagliare quindi la ricerca.

2.1.2 Costruire il Quad Tree

Notazione 2.1. Si indica con $pts(n)$ l'insieme dei punti rappresentati dal nodo n. Si indica con F_j l'insieme dei quadrati corrispondenti ai nodi che sono a distanza j dalla radice.

Per costruire un Quad Tree radicato in R a partire da un insieme di punti V, si ottiene $pts(R) = V$ ossia si costruisce R come il più piccolo quadrato che contiene tutti i punti in V. $F_0 = \{R\}$. La radice viene suddivisa in 4 quadrati che saranno aggiunti ad F_1, in generale avendo ottenuto $F_0, \ldots, F_{j-1}$, si ottiene F_j considerando tutti i $q \in F_{j-1}$ con $|pts(q)| > 1$ e partizionando ciascun quadrato in 4 parti uguali q_0, q_1, q_2, q_3 che saranno i figli di q. Ad ogni passo tutti i quadrati con $|pts(.)| \leq 1$ saranno considerati foglie e non saranno suddivisi ulteriormente. Per ogni nodo non foglia o foglia con $|pts(.)| \geq 1$ non vi è necessità di memorizzare tutti i punti $pts(.)$ ma solo un punto rappresentante.

Si supponga che un nodo abbia la seguente pseudo-struttura:

```
struct node{
   int count;
   int x, y, l;
   int h;
   node[4] childrens;
```

[1]Nel seguito quadrato e nodo verranno spesso usati come sinonimi.

```
    point* rep;
    points* setOfPoints;
}
```

in cui `h` rappresenta l'altezza di un nodo, `count` rappresenta $pts(.)$, `x,y` sono le coordinate dell'angolo in basso a sinistra del quadrato, `l` è la lunghezza del lato. `setOfPoints` è un puntatore a punti usato solo in fase di costruzione e `rep` è un puntatore al punto rappresentante del nodo.

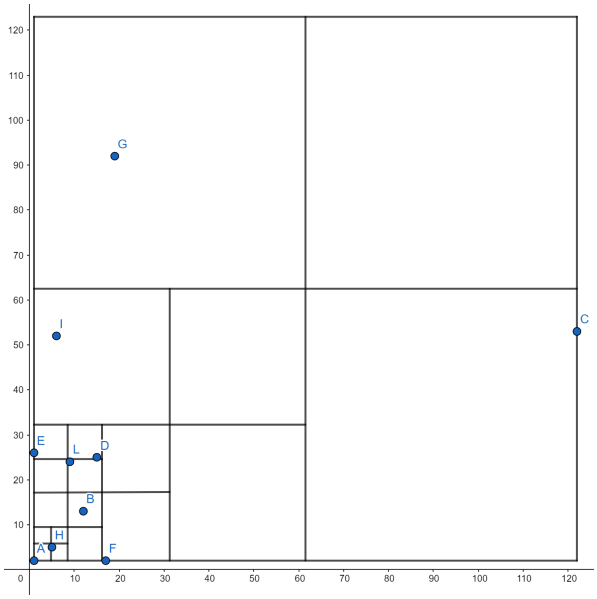
L'algoritmo 4 codifica le idee sopra descritte per la costruzione efficiente di un Quad Tree.

Algorithm 4 Costruzione di un Quad Tree su un insieme di punti

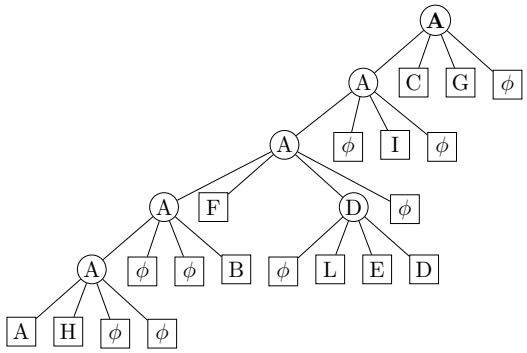
```
procedure BUILD-QUAD-TREE(points, n)
    root ← FIND-SMALLEST-SQUARE(points)
    root.count ← n
    root.rep ← points[0]
    F ← NEW-LIST-OF-SETS-NODES
    F_0 ← root
    j ← 0
    repeat
        j ← j + 1
        F_j ← NEW-SET-OF-NODES
        while (
            x ← NEXT(F_{j-1})) ≠ null do
            if x.count > 1 then
                childrens ← QUAD-SPLIT(x)
                for each c in childrens do
                    F_j ← F_j ∪ c
                end for
            end if
        end while
    until CONTAINS-NON-LEAFS(F_j)
    return root
end procedure
procedure NEXT(F)                                   ▷ F is passed by reference.
    result ← F.currentNode
    F ← F.next
    return result
end procedure
```

Lemma 2.1. La procedura `BUILD-QUAD-TREE` dell'algoritmo 4 prende tempo $O(n \log(n))$.

Dimostrazione. La procedura `FIND-SMALLEST-SQUARE` si implementa banalmente in $O(n)$. La procedura `CONTAINS-NON-LEAFS` prende tempo $O(|F_j|)$ dove F_j è l'insieme di nodi passati in input (per ogni nodo ci basta verificare il campo *node.count*). Il ciclo `repeat-until` viene eseguito esattamente i volte dove i è l'altezza dell'albero. Poiché i punti hanno dispersione polinomiale si ha $i \in O(\log(n))$. Sia $m_{k,j}$ il numero di punti contenuti in un certo quadrato $x_k \in F_{j-1}$, una chiamata alla procedura `QUAD-SPLIT`(x) prende tempo $O(m_{k,j})$. La procedura `QUAD-SPLIT`(x) crea 4 quadrati che partizionano x e li aggiunge a $x.childrens$, vengono scanditi tutti i punti in x.`setOfPoints` e ciascuno viene aggiunto ad esattamente un figlio di x, per ogni figlio f_i vengono inoltre calcolati gli attributi $f_i.count$ e nel caso questo valore sia maggiore di 0, viene scelto arbitrariamente un punto rappresentante che soddisfi



(a) Partizionamento dello spazio dovuto al Quad Tree.



(b) Rappresentazione ad albero del Quad Tree. I cerchi rappresentano nodi interni (e contengono l'identificativo del loro rappresentante) mentre i quadrati sono le foglie.

Figura 2.1: QuadTree risultante dall'insieme di punti $V = \{(1,2),(12,13),(122,53),(15,25),(1,26),(17,2),(19,92),(5,5),(6,52),(9,24)\}$ rispettivamente etichettati con A, B, ..., L. Di ogni quadrato sia il rappresentante il punto etichettato con una lettera 'minore'.

$f_i.rep \in f_i.\texttt{setOfPoints}$; infine viene eliminato lo spazio per $x.\texttt{setOfPoints}$.[2] Si ha che fissato j, $\sum_k O(m_{k,j}) = O(n)$, infatti in ciascun livello j tutti i quadrati $x \in F_j$ rappresentano al più tutta la regione di spazio dei punti di input. Allora in totale il costo dell'algoritmo 4 si suddivide in:

- Costo per la costruzione del quadrato contenente tutti i punti.
- Costo complessivo della procedura `QUAD-SPLIT`.
- Costo complessivo della procedura `CONTAINS-NON-LEAFS`.

E quindi si ottiene:

$$\begin{aligned} T(n) &= O(n) + \textstyle\sum_j \sum_k O(m_{k,j}) + \sum_{j=1}^{i} |F_j| \\ &= O(n) + iO(n) + iO(|F_j|) \\ &= O(n) + O(n\log(n)) + iO(n) \\ &= O(n\log(n)) \end{aligned} \tag{2.1}$$

□

2.1.3 Calcolare l'altezza

La procedura `CALCULATE-HEIGHT`, eseguita dopo la costruzione del Quad Tree calcola banalmente l'altezza di ogni nodo dell'albero riempiendo il relativo campo h. Per convenzione l'altezza di una foglia viene posta pari a 0.

[2] Si ricorda che er ogni nodo non foglia non si mantiene tutto l'insieme di punti contenuto nella relativa regione di spazio ma viene mantenuto solo un rappresentante.

2.1.4 Query

Varadarajan [1] propone il seguente modo di operare per implementare la procedura `QUERY`. Si supponga che la variabile best abbia due attributi:

`best.point`

`best.distance`

che memorizzano un punto e una distanza rispettivamente.

Algorithm 5 Esecuzione di una query

procedure `QUERY`$(p, root)$
  $l \leftarrow root.h$
  $E_0 \leftarrow \{root\}$
  $best_0.point \leftarrow null$
  $best_{-1}.point \leftarrow null$
  for $j \leftarrow 1$ **to** $l + 1$ **do**
    if $E_{j-1} = \phi$ **then break**
    end if
    $best_{j-1} \leftarrow$ `CALCULATE-BEST-DISTANCE`$(p, E_{j-1}, best_{j-2}.point)$
    $E_j \leftarrow$ `CALCULATE-BEST-SET`$(p, E_{j-1}, best_{j-1}.distance)$
  end for
  $R \leftarrow$ `CALCULATE-BEST-DISTANCE`$(p, E_j, best_{j-1}.point)$
  return $R.point$
end procedure

La variabile *best* viene usata per rappresentare un'approssimazione della minore distanza (e il punto che genera tale distanza) trovata attualmente tra il punto in input p e tutti gli altri punti. La procedura `CALCULATE-BEST-SET` prende in input la miglior distanza d_{max} precedentemente calcolata, un insieme di nodi E e il punto p. La procedura restituisce l'insieme

$$E' = \{q \mid q = E.childrens[i] \text{ per qualche i } \textbf{and } d_{square}(p, q) \leq d_{max}\} \tag{2.2}$$

ovvero l'insieme dei figli dei quadrati in E che distano al massimo d_{max} dal punto p. Si noti che $d_{square}(p, q)$ in questo caso non è la distanza euclidea ma è la distanza del punto p dal quadrato q; siano $\overline{ab}$, $\overline{bc}$, $\overline{cd}$ e $\overline{da}$ i lati di q, allora:

$$d_{square}(p, q) = \begin{cases} 0 & \text{se } p \text{ è contenuto in } q \\ \min_{l \in \{\ \overline{ab}, \overline{bc}, \overline{cd}, \overline{da}\}} d_{seg}(p, l) & \text{altrimenti} \end{cases} \tag{2.3}$$

Sia p'_l punto intersezione della retta r (definita dal segmento l) e della retta r' perpendicolare a r e passante per p, allora d_{seg} vale:

$$d_{seg}(p, l) = \begin{cases} d(p'_l, p) & \text{se } p'_l \text{ sta nel segmento } l \\ \min_{v \in \{\text{vertici di } l'\}} d(v, p) & \text{altrimenti} \end{cases} \tag{2.4}$$

Resta ora da capire che semantica abbia la procedura `CALCULATE-BEST-DISTANCE`. Sia definito $best_{j-1}.point$ come il punto più vicino a p tra i rappresentanti degli insiemi di quadrati $E_0, \ldots, E_{j-1}$ e sia invece $best_{j-1}.distance$ la distanza tra p e $best_{j-1}.point$. La procedura `CALCULATE-BEST-DISTANCE`

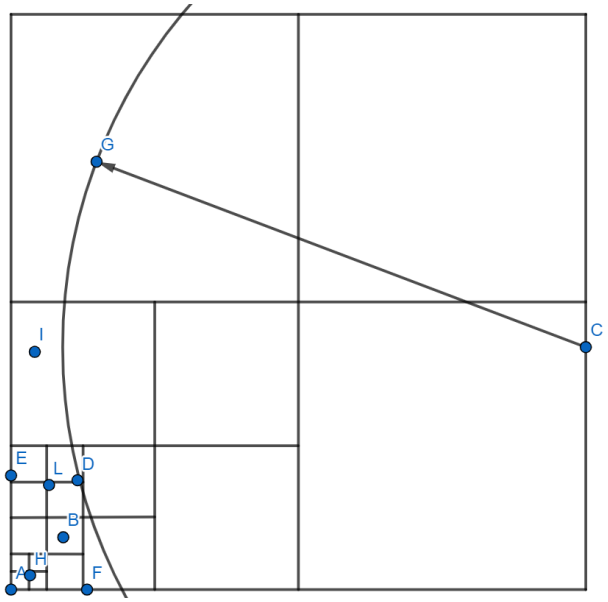


Figura 2.2: Esempio di esecuzione di `QUERY`$(C, root)$: $root.0.0.0$ viene scartato nel calcolo di E_3.

calcola in maniera efficiente tale risultato: Si vede banalmente che $best_{j-1}.point$ per $j > 2$ è il punto più vicino tra i rappresentanti di E_{j-1} e il punto $best_{j-2}$. In questo modo la complessità della procedura è $O(|E_{j-1}|)$.

Concludendo qui, l'algoritmo 5 ritornerebbe p stesso (che è a distanza 0 p) e non p' a distanza minima da p: per ovviare al problema basta ignorare il punto p nel caso la `CALCULATE-BEST-DISTANCE` lo incontri.

Analisi di una query

L'algoritmo 5 procede per passi visitando tutti i livelli dell'albero. Ad ogni passo si supponga di star visitando i quadrati in E_{j-1}, viene calcolata una stima della distanza minima tra p e il (vero) punto più vicino a p' e si scende nei figli di E_{j-1} che sono più vicini a p della distanza calcolata. Viene presentato un esempio di esecuzione: si consideri la situazione rappresentata in figura 2.1 e si provi ad eseguire `QUERY`$(C, root)$. Se n è un nodo, viene indicato con $n.0$ il quadrato in basso a sinistra, $n.1$ quello in basso a destra, $n.2$ quello in alto a sinistra e $n.3$ quello in alto a destra. Nella rappresentazione ad albero adottata in figura 2.1b dato un nodo n, i nodi figli $n.0, \ldots, n.3$ sono nel livello sotto ad n da sinistra a destra. In questo esempio, di ogni quadrato sia il rappresentante il punto etichettato con una lettera 'minore' quindi ad esempio $root.0.rep = A$ e $root.0.2.rep = I$.

1. Si inizi con $E.0 = root$ e si entri nel ciclo.

2. $best_0.point = root.rep = A$.

3. Si ha che il lato del quadrato è 60.5 e graficamente $best.0.dist >> 60.5$ quindi nessun figlio di $root$ viene scartato e $E_1 = \{root.0, root.1, root.2, root.3\}$. In realtà si può già scartare $root.3$ vedendo che non ha figli e non ha rappresentante.

4. Per calcolare $best_1.point$ si considerino $d(A, C), d(C, C), d(G, C)$ ma si scarti $d(C, C)$ poichè sarebbe 0 quindi si ottiene $best_1.point = G$.

5. Si ottiene $E_2 = \{root.0.0, root.0.1, root.0.2, root.0.3\}$ ma si scartino già $root.0.3$ e $root.0.1$ che non hanno né figli né rappresentante.

6. Per calcolare $best_2$ si considerino le distanze con i punti A, I oltre che col punto $best_1.point$ ossia G. E anche in questo caso si riottiene $best_2.point = G$.

7. Per calcolare E_3 si consideri solo $root.0.0$ che è l'unico ad avere figli e si ottiene $E_3 = \{root.0.0.1, root.0.0.2, root.0.0.3\}$ ma si scarti $root.0.0.3$ che non ha né figli né rappresentante. Si noti che $root.0.0.0$ è stato scartato nel calcolo di E_3 poiché troppo distante da C ovvero $d_{square}(C, root.0.0.0) > d(C, G)$ (si veda la figura 2.2).

8. Per calcolare $best_3$ si considerino le distanze con i punti D, F oltre che con $best_2.point = G$ e si ottiene ancora una volta $best_3.point = G$

9. Per calcolare E_4 si consideri solo $root.0.0.2$ che è l'unico con figli e si inseriscano $root.0.0.2.1$ e $root.0.0.2.3$ in E_4. Nel calcolo di E_4, $root.0.0.2.0$ e $root.0.0.2.2$ vengono scartati poiché troppo distanti rispetto a C ovvero $d_{square}(C, root.0.0.2.0) > d(C, G)$ e $d_{square}(C, root.0.0.2.2) > d(C, G)$ (si riveda la figura 2.2).

10. Per calcolare $best_4$ si considerino le distanze con D, L ma $best_4.point$ varrà ancora G.

11. $E_5 = \phi$ in quanto $root.0.0.2.1$ e $root.0.0.2.3$ non hanno figli.

12. Si conclude anticipatamente il ciclo e si calcola il risultato tra G, D, L e si ottiene G.

Dall'esempio è interessante notare che:

- Non necessariamente il risultato viene raffinato con i vari passi: può darsi il caso in cui il più vicino viene selezionato subito.
- Non necessariamente vengono scanditi tutti i livelli.
- Se il risultato viene trovato ai primi passi, viene tramandato nei vari $best_j$, mentre se il più vicino viene trovato solo a livello delle foglie e non è rappresentante di nessun quadrato, il risultato sarà raffinato ad ogni passo e verrà trovato visitando l'ultimo livello.

Studiando ora la complessità dell'algoritmo 5, e ricordando che fissato un livello j entrambe le procedure richiamate nel ciclo, `CALCULATE-BEST-DISTANCE` e `CALCULATE-BEST-SET` prendono tempo lineare sulla cardinalità dell'insieme di nodi che si stanno considerando.[3] Si conclude che la complessità in tempo è $O(|E_0| + \cdots + |E_l|)$ con l numero di livelli nell'albero.

Analisi di tutte le query

Si dimostra [1] che fissato un punto, a ogni passo, E_j contiene un numero costante di quadrati e quindi ricordando che il Quad Tree ha altezza logaritmica si ottiene che il tempo complessivo per l'algoritmo 3 (che esegue n volte l'algoritmo 5 e 1 volta l'algoritmo 4) è $O(n \log(n))$.

[3] In realtà `CALCULATE-BEST-SET` deve operare anche sui figli dei nodi ma si ricorda che in un Quad Tree così come è stato definito i figli sono un numero costante ovvero 0 (nel caso di una foglia) oppure 4.

A dimostrazione sono stati ricavati i seguenti dati sperimentali da considerarsi solo come una misura qualitativa:

n	$\mathbf{h_n}$	$\mathbf{E_n}$	$\frac{\mathbf{E_n}}{\mathbf{n}}$
1000	10,2	31038	31,0
10000	13,7	421254	42,1
100000	17,1	5390060	53,9

Tabella 2.1: Dati sperimentali dell'algoritmo 3

È stato eseguito l'algoritmo 3 per 50 volte, prima con $n = 1000$, poi $n = 10000$ e infine $n = 100000$ e ogni volta sono stati generati n punti pseudo-casualmente: fissato n, ogni coordinata di ogni punto è un numero intero non negativo nel range $[0, 20{*}n{-}1]$; assumendo una funzione pseudocasuale uniformemente distribuita, gli n punti hanno dispersione polinomiale.

Con E_i indichiamo $\sum_{p \in V} (|E_{0,p}| + \cdots + |E_{l,p}|)$ per l'esecuzione $i - ma$ dell'algoritmo ossia la somma delle cardinalità di tutti gli insiemi di quadrati presi in considerazione. Indichiamo quindi con E_n la media degli E_i ossia $E_n = \frac{1}{50} \sum_{i=1}^{50} E_i$. Analogamente h_n è l'altezza media degli alberi ottenuti sulle 50 iterazioni. I dati dimostrano empiricamente che E_n cresce proporzionalmente a $\log(n)$.

2.1.5 Euristica shrink and prune

Prendendo ispirazione dall'algoritmo di Vaidya [2], è possibile implementare un'euristica per migliorare la costruzione del Quad Tree. Si può costruire un Quad Tree applicando le seguenti idee:

1. **Shrink:** Ad ogni nodo con quadrato associato q, si sostituisca l'associazione di q con q'. q' è il più piccolo quadrato contenente tutti i punti che il nodo rappresenta e q' è contenuto in q.

2. **Prune:** Si introduca il campo `countChilds` in ogni nodo e si eliminino dall'albero le foglie che non contengono punti.

La proprietà 1 garantisce che splittando un nodo n con più di 2 punti, esistano due figli di n con almeno un punto. La proprietà 2 garantisce che nell'albero siano presenti esattamente n foglie. Queste due modifiche si possono implementare nella procedura `QUAD-SPLIT` senza cambiarne il costo asintotico. Si noti inoltre che ogni nodo interno ha da 2 a 4 figli e quindi il numero totale di nodi nell'albero è $2n$ [2]. Mantenendo l'ipotesi di spread polinomiale, l'altezza dell'albero è $O(\log(n))$. Da qui in poi si supponga dunque che il Quad Tree sia implementato con queste euristiche.

2.1.6 Generalizzazione a m vicini

Si propone ora una generalizzazione dell'algoritmo di Varadarajan al caso con m generico. Alla struttura nodo vengono aggiunti i seguenti campi:

```
boolean isActive;
node* linkToRepLeaf;
node* parent;
```

Vengono create due nuove procedure di seguito descritte. La procedura `CALULATE-LEAFS` in tempo

$O(n\log(n))$ restituisce l'insieme delle foglie con un punto dell'albero radicato in T. La procedura `LINK-REP-LEAFS` riempirà il campo `node.linkToRepLeaf`, per fare questo si supponga di aver già calcolato per ogni nodo dell'albero il campo `node.parent`, si prenda in input l'insieme restituito da `CALULATE-LEAFS` e per ogni foglia f si risalga sul cammino verso la radice e se per un nodo n vale `n.rep = f.rep` allora si imposti `n.linkToRepLeaf = f`. Le foglie con (esattamente) un punto sono n, l'altezza dell'albero è $O(\log(n))$ e quindi anche questa procedura è implementabile in $O(n\log(n))$. Viene ora data un'intuizione di come funzionerà l'algoritmo:

1. Si trovi il punto p' più vicino a q con l'algoritmo `QUERY` sopra descritto.
2. Si ponga (in tempo costante) `p'.isActive = false`.
3. Si riesegua l'algoritmo della query considerando solo i punti attivi attualmente.
4. Si iteri m volte, ciascuna volta disattivando un punto in più.
5. Trovati gli m vicini per q, in tempo $O(m)$ si riattivino tutti i punti.

Vengono ora formalizzate le precedenti idee con lo pseudo-codice delle nuove procedure:

Algorithm 6 ALL-MNN caso bidimensionale utilizzando i Quad Tree

```
procedure ALL-MNN(points, n)
    T ← BUILD-QUAD-TREE(points, n)
    CALCULATE-HEIGHT(t)
    leafs ← CALULATE-LEAFS(t)
    LINK-REP-LEAFS(leafs)
    sol ← NEW-ARRAY(n)
    for i ← 1 to n do
        sol[i] ← M-QUERY(T, leafs, leafs[i], m)
    end for
    return sol
end procedure
```

Algorithm 7 M-QUERY caso bidimensionale utilizzando i Quad Tree

```
procedure M-QUERY(T, leafs, p, m)
    deactivatedNodes ← φ
    sol ← φ
    for i ← 1 to m do
        q ← QUERY(leafs[i].repPoint, T)
        q.isActive ← false
        deactivatedNodes ← deactivatedNodes ∪ q
        sol ← sol ∪ q.repPoint
    end for
    for i ← 1 to m do
        q[i].isActive ← true
    end for
    return sol
end procedure
```

Si noti che l'algoritmo 5 `QUERY` rimane uguale con le seguente modifiche:

- Restituisce in output l'intera foglia contenente il punto risultato e non solo il punto stesso.
- I campi di `best.point` sono di tipo nodo e non semplici punti.
- Utilizza una versione modificata della procedura `CALCULATE-BEST-DISTANCE` che sarà descritta nel seguito.

È stato chiarito in precedenza che in `CALCULATE-BEST-DISTANCE` vengono ignorati i nodi disattivati. Ma nell'algoritmo 7 vengono disattivate solo foglie, come comportarsi dunque con nodi non foglia ma che hanno come rappresentante un punto di una foglia disattivata? Qui viene in aiuto il campo `node.linkToRepLeaf`: adottando strategie simili ad alcune implementazioni delle strutture dati per insiemi disgiunti, si decide di tenere il campo `node.isActive` aggiornato solo nei nodi foglia, i nodi interni per decidere se sono attivi o meno, accederanno alla foglia del loro rappresentante e controlleranno se tale foglia sia attiva o meno.

Le idee vengono formalizzate con il listato 8.

Algorithm 8 IS-ACTIVE

```
procedure IS-ACTIVE(n)
    if n.countPoints == 1 then
        return n.isActive
    else
        return n.linkToRepLeaf.isActive
    end if
end procedure
```

Con questo modo di operare disattivando una foglia, vengono disattivati automaticamente, in tempo costante, anche tutti i nodi che hanno come rappresentante il punto di tale foglia.

Quindi assumendo la correttezza dell'algoritmo 5, la correttezza dell'algoritmo 7 segue direttamente. I punti disattivati è come non fossero presenti nel Quad Tree e fissato il punto p, vengono disattivati progressivamente $p_1, p_2, \dots p_{m-1}$ e quindi otteniamo gli m punti più vicini. Si noti inoltre che l'algoritmo 6 non solo è corretto ma per ogni punto restituisce gli m punti più vicini ordinati rispetto alla distanza.

Analisi

Si è tentati a concludere immediatamente che l'algoritmo 6 abbia complessità $O(mn\log(n))$ in quanto l'algoritmo 5 viene chiamato mn volte. In realtà scelto un m costante e sufficientemente piccolo rispetto a n si mostra empiricamente (si veda la sezione 4.2.3 del capitolo 4) che le performance sono assimilabili a $O(mn\log(n))$. Ad esempio con $m = 10$ e $n = 10000$ le performance sono ottimali ovvero l'algoritmo prende esattamente tempo $10T(n)$ dove $T(n)$ è il tempo di esecuzione per l'algoritmo 3. Incrementando però m e facendolo avvicinare a n o ad una frazione di n sufficientemente grande si vede che le performance iniziano a degradare fino al vero caso peggiore che è $O(n^3)$. Ci si chiede quindi da cosa derivi il comportamento cubico dell'algoritmo 6. Facendo un'analisi più raffinata si nota che, continuando a disattivare nodi, ad ogni passo la cardinalità di E_j non è più costante ma è $O(n)$; infatti disattivando un punto, vengono disattivati anche tutti i nodi che utilizzano tale punto come rappresentante e se questi nodi iniziano ad essere la totalità o quasi, scendendo in ogni livello non diminuiamo mai la cardinalità di E_j poiché si è costretti a visitare tutti i figli dei nodi nell'insieme, e inoltre la stima *best* non migliora mai

o migliora difficilmente: se tutti i nodi di un livello sono disattivati, verranno ignorati dalla procedura `CALCULATE-BEST-DISTANCE`. In tali situazione si arriva a visitare tutti i nodi di ciascun livello e la scelta del $k - imo$ vicino di p avviene solo a livello delle foglie controllando le distanze tra p e tutte le altre $O(n)$ foglie. Concludendo, se $m \in \Theta(n)$, con n punti, ogni chiamata alla procedura `QUERY` costa $O(n)$[4] e otteniamo quindi costo $\Theta(n)\Theta(n)O(n) = O(n^3)$. Si noti che anche se le performance fossero davvero state $O(mn\log(n))$ scegliendo $m \in \Theta(n)$ l'algoritmo si comporterebbe peggio rispetto alla soluzione naive che usa gli n^2 confronti.

Euristica per la scelta del rappresentante

Nell'algoritmo 4 per la costruzione del Quad Tree, in ogni nodo interno il rappresentante è scelto con massimo grado di libertà. Ci si chiede ora se una scelta ottimale del rappresentante possa fare la differenza sul costo complessivo dell'algoritmo 6. Si inizi col notare che operando con qualsivoglia euristica il caso peggiore rimarrà $O(n^3)$: basta pensare al caso in cui $m = n - 1$. Per ottimizzare l'algoritmo si vuole che un punto sia utilizzato il minor numero di volte come rappresentante, in questo modo la sua disattivazione avrà il minor impatto possibile. Sotto ipotesi di dispersione polinomiale e con l'euristica *shrink and prune*, ci sono $O(n)$ nodi interni e n foglie non vuote (il caso migliore è quello di un Quad Tree completo in cui si hanno n foglie e $\sum_{i=0}^{\log_4(n)} 4^i - n = \frac{4n-1}{3} - n = \frac{n-1}{3}$ nodi interni) quindi con una scelta ottimale si può chiedere che un punto sia rappresentante di al più un numero costante di nodi interni. Questa richiesta non impatta sui tempi dell'algoritmo di costruzione del Quad Tree visto in precedenza.

Strade alternative

Dato un Quad Tree un altro algoritmo per `ALL-MNN` avrebbe potuto invocare l'algoritmo `QUERY` (una volta per punto) per poi risalire spiralmente l'albero a partire dalla foglia contenente il punto più vicino. Tuttavia per un simile algoritmo si dovrebbe definire come procedere per risalire a spirale e potrebbero essere necessari attributi ulteriori per `node` e cambiamenti sostanziali alla struttura dati.

Un'altra possibilità, dato un punto p, sarebbe quella di mantenere durante la discesa dell'albero un *best-set* di cardinalità m dove sono contenuti gli m vicini per p che vengono inizialmente stimati e via via raffinati. Una strategia simile è adottata(e verrà descritta dettagliatamente) nella sezione successiva nel caso dei K-D Tree ed è stata adottata anche nell'algoritmo di Vaidya [2] in cui per ogni punto b sono costruiti gli insiemi *Attractors(b)*[5], e *Neighbors(b)* e a termine dell'esecuzione dell'algoritmo gli insiemi *Neighbors(b)* contengono la soluzione cercata.

2.1.7 Generalizzazione a d dimensioni

Tutti i ragionamenti e algoritmi proposti fin'ora in questa sezione possono essere banalmente generalizzati al caso con d dimensioni. Si modifichi il punto 1 della definizione 2.1 di un Quad Tree:

- Ogni nodo ha al più 2^d figli, precisamente un nodo interno ne ha 2^d e un nodo foglia 0.

[4] Si ricordi che con l'euristica *shirnk and prune* il numero totale di nodi nell'albero è $O(n)$.
[5] I punti b' in *Attractors(b)* sono quelli tali che $b \in$ *Neighbors(b')*.

Con questa definizione tutti gli algoritmi rimangono completamente invariati, ma ciò che cambia è la complessità computazionale, si vede infatti un aggravio esponenziale sulla dimensionalità praticamente su tutte le procedure viste.

- **Calcolare una distanza.** Costa banalmente $O(d)$.
- **Costruzione del tree.** Ogni nodo non foglia va suddiviso in 2^d iper-cubi.
- **Calcolare il best set.** Dobbiamo controllare tra tutti i figli di ogni nodo nell'insieme in input, e anche i questo caso i figli sono 2^d.
- **Tutte le 1-query.** In totale abbiamo costo $O(2^d n \log(n))$.

Applicabilità dell'esponenziale

Fissato d, si può calcolare la costante 2^d. Si può immaginare che per i casi $d = 1, 2, 3, 4$ la soluzione sia ancora valida e utile nella pratica per battere l'algoritmo quadratico, ma spingendosi oltre con la dimensionalità (anche di poco, ad esempio con $d = 12$) si osservano subito tempi e costi inaccettabili.

2.1.8 Conclusioni

In questa sezione è stata presentata una prima semplice struttura per partizionare lo spazio e i relativi primi algoritmi per calcolare `ALL-1NN` in tempo $O(2^d n \log(n))$ e `ALL-MNN` che nel caso peggiore prende tempo $O(2^d n^3)$. L'esponenzialità rende i risultati inapplicabili per alta dimensionalità, ma scelto un d costante e un m sufficientemente piccolo rispetto a n, l'algoritmo 6 per `ALL-MNN` ha performance vicine a $O(mn \log(n))$ e riesce a battere (teoricamente e praticamente) il tempo di esecuzione della soluzione *naive*.

2.2 K-D Tree

Consideriamo ora un'altra struttura dati per partizionare lo spazio. La prima versione dei K-D Tree è stata proposta da Bentley nel 1975 [3] insieme ad un algoritmo per il problema del vicinato che opera sotto alcune restrizioni, ad esempio, la distanza deve essere una metrica, i.e. deve soddisfare anche la disuguaglianza triangolare e la soluzione è efficiente solo per la metrica di *Minkowski* ∞. La struttura dati proposta è una generalizzazione dei *binary tree* al caso in cui la chiave abbia d dimensioni[6]. I K-D Tree possono essere utilizzati per una varietà di problemi tra i quali *region queries*, *exact match queries*, ossia determinare se un punto p sta nell'albero T, e *partial match queries*. Lo stesso Bentley insieme a Friedman e Finkel [4] ha in seguito proposto una versione migliorata dei K-D tree con l'obiettivo specifico di risolvere il problema degli m vicini di un punto. In questa sezione, seguendo un approccio *bottom-up*, saranno definiti prima la struttura dati e gli algoritmi per la costruzione e poi l'algoritmo per risolvere il problema 1.1.

2.2.1 La struttura dati

Viene ora presentata la struttura dati usata in [4] che è una variante di quella originaria proposta in [3].

Definizione 2.2 (K-D Tree su un insieme di n punti V)**.** Un K-D Tree è un albero binario tale che:

1. La radice contiene e rappresenta tutti i punti in V, ogni nodo rappresenta un sottoinsieme di V.
2. Ogni nodo ha 0 o 2 figli uno destro (detto anche `hison`) e uno sinistro (detto anche `loson`).[7]
3. Ogni nodo interno ha un campo `discriminator`, detto discriminatore che è un intero positivo nell'intervallo $[1, d]$.
4. Ogni nodo interno ha un campo `partitionValue`, il valore di partizione.
5. Ogni punto $p \in V$ viene memorizzato in esattamente una foglia.
6. Ogni nodo foglia è un *bucket* e contiene al più b punti.
7. I nodi foglia rappresentano piccoli insiemi mutamente esclusivi di punti in V.
8. **Proprietà K-D Tree:** Per ogni nodo n, sia $j = n.discriminator$, per ogni punto $p = (p_1, \ldots, p_d)$ contenuto nel sottoalbero radicato in `n.left` vale $p_j < n.partitionValue$ e analogamente per ogni punto p nel sottoalbero radicato in `n.right` vale $p_j \geq n.partitionValue$.

La semantica di questa definizione è che ogni nodo con il suo discriminatore e valore di partizione, definisce i limiti del dominio dei nodi figli. Il discriminatore definisce in ogni nodo la coordinata su cui vengono partizionati i punti. Per questo si può immaginare che ogni nodo abbia i campi `upperBounds` e `lowerBounds`, due array con d elementi ciascuno che definiscono i limiti geometrici entro cui i punti rappresentati da quel nodo si trovano. La radice è inizializzata con $upperBounds = (\infty, \ldots, \infty)$ e

[6] La dicitura K-D sta per *K-dimensional*, tuttavia in questo documento, per coerenza si continuerà a utilizzare d per denotare il numero di dimensioni.

[7] `loson` sta per *lower-son* e analogamente `hison` sta per *higher-node*.

$lowerBounds = (-\infty, \ldots, -\infty)$. In realtà nell'implementazione dell'algoritmo non servirà memorizzare esplicitamente questi campi ma saranno calcolati *run-time* nella discesa dell'albero se necessari.

Nell'algoritmo originario, i nodi in ogni livello hanno lo stesso discriminante. La radice ha discriminante 0, i due figli hanno discriminante 1 e in ogni livello i discriminanti si succedono ciclicamente, ovvero viene definita la funzione $NEXTDISC$ come $NEXTDISC(i) = (i + 1) \mod d$. Inoltre fissato un nodo n, fissato $n.discriminator = j$, il valore di partizione è un valore casuale scelto tra l'insieme $\{p_j | p$ è un punto in una foglia dell'albero radicato in $n\}$.
Nel nostro caso vedremo dettagliatamente come scegliere i valori di partizione e il discriminatore per costruire quello che è chiamato un *K-D Tree ottimizzato*. Sarà inoltre discusso come scegliere la dimensione dei bucket b.

2.2.2 Costruire il K-D Tree

Si supponga di avere a disposizione le procedure `CHOOSE-DISCRIMINATOR`$(points, d)$ e `CHOOSE-PARTITION-VAL`$(points)$ che dato un nodo calcolino il discriminatore e il valore di partizione rispettivamente in tempo lineare su nd e sulla cardinalità dell'insieme *points*. Sarà discusso nelle prossime sezioni come scegliere questi valori e conseguentemente implementare queste procedure.

Supponiamo che la struttura di un nodo sia:

```
struct node{
   bool isLeaf;
   int partitionValue;
   int discriminator;
   node* hison;
   node* loson;
   points* setOfPoints;
}
```

in cui `loson` è il figlio sinistro e `hison` quello destro. Il campo `setOfPoints` è settato solo nelle foglie.

Algorithm 9 Costruzione di un K-D Tree su un insieme di punti

procedure `BUILD-KD-TREE`$(points)$
  if $|points| \leq b$ **then**
    return `MAKE-LEAF`$(points)$
  end if
  $pv \leftarrow$`CHOOSE-PARTITION-VAL`$(points)$
  $disc \leftarrow$`CHOOSE-DISCRIMINATOR`$(points, d)$
  $points_{left} \leftarrow \{q = (q_1, \ldots, q_d) | q \in points, q_{disc} < pv\}$
  $points_{left} \leftarrow \{q = (q_1, \ldots, q_d) | q \in points, q_{disc} \geq pv\}$
  $node_{left} \leftarrow$ `BUILD-KD-TREE`$(points_{left})$
  $node_{right} \leftarrow$ `BUILD-KD-TREE`$(points_{right})$
  return `MAKE-INTERNAL-NODE`$(disc, pv, node_{left}, node_{right})$
end procedure

L'algoritmo di costruzione 9 procede ricorsivamente e senza sapere le cardinalità di $points_{left}$ e

$points_{right}$, otteniamo che una chiamata ha costo in tempo $T(n)$ con $n = |points|$:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq b \\ T(\alpha n) + T(\beta n) + \Theta(nd) & \text{altrimenti} \end{cases} \tag{2.5}$$

con $\alpha + \beta = 1$. Si noti già da ora, senza guardare le proprietà dell'algoritmo di ricerca, che per non arrivare al costo $T(n) = O(n^2)$ si vogliono bilanciare α e β, e un modo per farlo potrebbe essere quello di scegliere come valore di ripartizione la mediana dei punti passati in input.

2.2.3 Query

L'algoritmo di ricerca proposto [4] è descritto anch'esso ricorsivamente e prende in input un punto $p = (p_1, \ldots, p_d)$ e il nodo u che si sta visitando. Viene mantenuta una max-coda con priorità[8] di lunghezza m, contenente la stima attuale degli m vicini per p, in cui la priorità è determinata dalla loro distanza con p. In questo modo è possibile avere in tempo $O(1)$ la distanza $m - ima$ trovata fino ad ora. Ad ogni passo si consideri di visitare un nodo. Se il nodo è una foglia vengono considerati tutti i punti contenuti e la coda viene aggiornata. Se il nodo è un nodo interno con valore di partizione pv e discriminatore $disc$, si scende ricorsivamente nel nodo u' ovvero $hison$ se $p_{disc} \geq pv$, $loson$ altrimenti. Quando il controllo ritorna dalla chiamata ricorsiva è necessario determinare se bisogna fare una seconda chiamata ricorsiva nell'altro figlio (u''). Si consideri un iper-sfera centrata in p e che ha come raggio la distanza $m - ima$ che si trova nella coda. Se l'ipersfera interseca i limiti geometrici del nodo figlio u'' ancora da visitare, allora è necessario fare una chiamata ricorsiva su di esso. La funzione `BOUNDS-OVERLAP-BALL` calcola questo test. Quando si è finito di visitare il nodo corrente (sia esso un nodo foglia oppure uno interno), si controlla se sia necessario continuare l'algoritmo oppure se si può terminare la procedura. Questo test consiste nel controllare che l'ipersfera descritta prima sia completamente contenuta nei limiti geometrici del noto corrente u ed è implementato dalla funzione `BALL-WITTHIN-BOUNDS`. Gli algoritmi 10, 11 e 12 formalizzano queste idee. Si noti che è possibile implementare il test `BOUNDS-OVERLAP-BALL` calcolando la più piccola distanza tra il punto p e i limiti geometrici in considerazione; se questa distanza è maggiore del raggio r dell'ipersfera, il test fallisce ossia non vi è nessuna intersezione. Per calcolare questo risultato efficientemente, si mantiene una variabile $minSquareDist$, si considera una coordinata alla volta e la si incrementa del valore opportuno (ovvero il quadrato della differenza tra p_j e il limite geometrico nella coordinata j) e se la radice di $minSquareDist$ è maggiore di r significa che non serve procedere col considerare le altre coordinate (la distanza è già maggiore del raggio della sfera) quindi si termina il test restituendo false. Si noti inoltre che nella procedura `VISIT-LEAF` è necessario (come fatto nel capitolo dei Quad Tree) ignorare il punto p stesso se trovato in un *bucket*.

2.2.4 Analisi e costruzione del K-D Tree ottimizzato

L'obiettivo del K-D Tree ottimizzato è scegliere il discriminatore e il valore di partizione per minimizzare il costo atteso per cercare il vicinato di un punto, in particolare si vuole minimizzare il numero di punti esaminati durante la ricerca. La soluzione dipende in generale dalla distribuzione dei punti nello spazio considerato, che in generale non è nota, quindi si procederà col definire una procedura indipendente

[8] Implementabile con una max-heap.

Algorithm 10 Test dei limiti geometrici

```
procedure BALL-WITHIN-BOUNDS(p)
    if GET-HEAP-LENGTH(mh) < m then
        return false
    end if
    r ← GET-HEAP-MAX(mh)                          ▷ r is the distance of the m-th point.
    for i = 1 to d do
        if r ≥ |p_i − lowerBounds[i]|  or  r ≥ |p_i − upperBounds[i]| then
            return false
        end if
    end for
    return true
end procedure
procedure BOUNDS-OVERLAP-BALL(p)
    if GET-HEAP-LENGTH(mh) < m then
        return true
    end if
    minDist ← 0
    r ← GET-HEAP-MAX(mh)                          ▷ r is the distance of the m-th point.
    for i = 1 to d do
        if p_i < lowerBounds[i] then
            minDist ← minDist + (p_i − lowerBounds[i])^2
            if ∛minDist > r then
                return false
            end if
        else if p_i > upperBounds[i] then
            minDist ← minDist + (p_i − lowerBounds[i])^2
            if ∛minDist > r then
                return false
            end if
        end if
    end for
    return true
end procedure
```

Algorithm 11 Esecuzione di una query

```
procedure QUERY(p = (p_1, ..., p_d), u)
    if u.isLeaf = true then
        VISIT-LEAF(p, u)
        if BALL-WITHIN-BOUNDS(p) = true then
            exit: solution found
        end if
        return
    end if
    disc ← u.discriminator
    pv ← u.partitionValue
    if p_disc < pv then                          ▷ Recursive call on the right side.
        tempBound ← upperBounds[disc]
        upperBounds[disc] ← pv
        QUERY(p, u.loson)
        upperBounds[disc] ← tempBound
    else
        tempBound ← lowerBounds[disc]
        lowerBounds[disc] ← pv
        QUERY(p, u.hison)
        lowerBounds[disc] ← tempBound
    end if
    if p_disc < pv then          ▷ Recursive call on the 'opposite' side (if necessary).
        tempBound ← lowerBounds[disc]
        lowerBounds[disc] ← pv
        if BOUNDS-OVERLAP-BALL(p) = true then
            QUERY(p, u.hison) end if
        lowerBounds[disc] ← tempBound
    else
        tempBound ← upperBounds[disc]
        upperBounds[disc] ← pv
        if BOUNDS-OVERLAP-BALL(p) = true then
            QUERY(p, u.loson) end if
        upperBounds[disc] ← tempBound
    end if
    if BALL-WITHIN-BOUNDS(p) = true then
        exit: solution found
    end if
    return
end procedure
procedure VISIT-LEAF(p, u)
    for each point q ≠ p in u.setOfPoints do
        d_max, q_max ← GET-HEAP-MAX(mh)          ▷ q_max is the point. d_max is the distance from p.
        dist ←CALCULATE-DISTANCE(p, q)
        if GET-HEAP-LENGTH(mh) < m then
            HEAP-INSERT(mh, dist, q)
        else if dist < d_max then
            HEAP-DEQUEUE(mh, q_max)
            HEAP-INSERT(mh, dist, q)
        end if
    end for
end procedure
```

Algorithm 12 ALL-MNN utilizzando i K-D Tree

```
global b                                          ▷ Size of buckets.
global m                                          ▷ Number of neighbors.
global mh                                         ▷ The max-priority queue.
global upperBounds
global lowerBounds
procedure ALL-MNN(points)
    T ← BUILD-KD-TREE(points)
    sol ← NEW-ARRAY(n)
    for i ← 1 to n do
        p ← points[i]
        mh ←NEW-MAX-HEAP(m)
        upperBounds ← (∞, ..., ∞)
        lowerBounds ← (−∞, ..., −∞)
        QUERY(p, T)
        sol[i] ← COPY-HEAP(mh)
    end for
    return sol
end procedure
```

dalla distribuzione e che per ogni nodo u prenda in considerazione solo i punti che esso contiene per evitare il problema di ottimizzazione di un albero binario che è noto essere NP-completo [24]. Si ricorda che l'informazione portata da una scelta binaria è massima quando le due possibilità sono equamente probabili (si riveda la sezione 1.2.1). In questo caso dato un punto si vuole che abbia la stessa probabilità di stare su ciascun lato della partizione. Si sceglie quindi come valore di partizione la mediana dei punti che il nodo u rappresenta, indipendentemente dalla scelta del discriminatore; si noti che questa decisione porta ad ottenere $\alpha = \beta = \frac{1}{2}$ nell'equazione 2.5. Nell'algoritmo 11 viene evitata la chiamata ricorsiva nel lato opposto della partizione se il test `BOUNDS-OVERLAP-BALL` fallisce ovvero se la distanza dalla partizione è maggiore del raggio della iper-sfera centrata in p e con raggio la distanza tra p e l'm-imo vicino di p. La probabilità che il test fallisca è quindi minima per la coordinata con più dispersione prima della partizione.

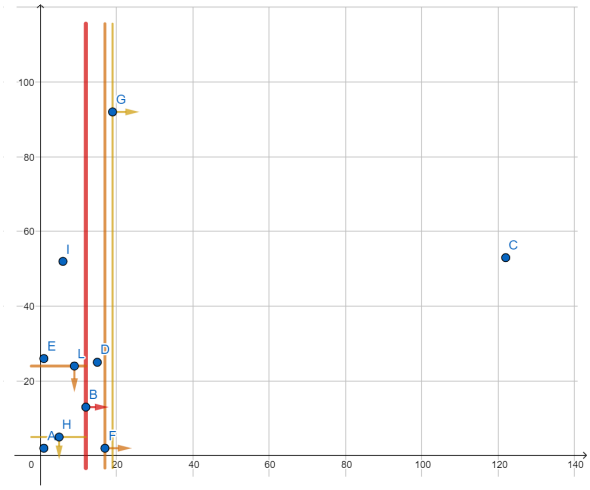
Nelle figura 2.3 viene rappresentato un K-D Tree costruito sullo stesso insieme di punti della figura 2.1. Si notino sostanziali differenze rispetto ai Quad Tree, in particolare nei Quad Tree vengono eseguiti degli splitting senza tenere conto di come siano distribuiti i punti. Entrambi gli alberi sono una generalizzazione dei binary tree nel caso unidimensionale ma le scelte prese sono sostanzialmente opposte: i K-D Tree restano degli alberi binari, mentre nei Quad Tree viene aumentata l'arietà esponenzialmente col numero delle dimensioni.

Lemma 2.2. L'algoritmo 9, con le scelte appena descritte per la selezione del discriminatore e del valore di partizione, prende tempo $\Theta(dn \log(n))$. Lo spazio utilizzato per memorizzare il K-D Tree è $O(n)$.

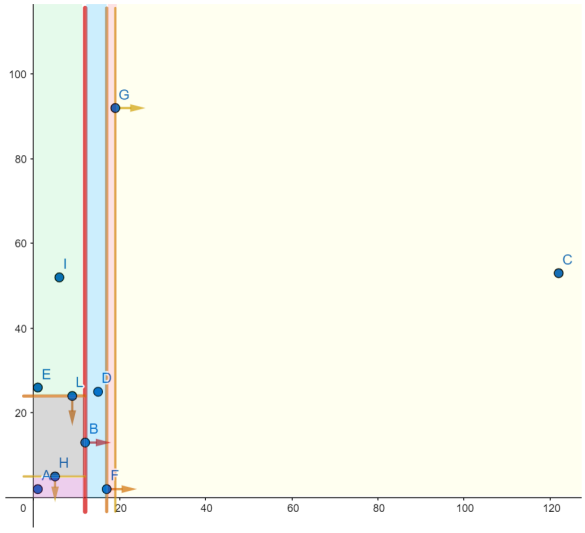
Dimostrazione. La procedura `CHOOSE-DISCRIMINATOR` prende tempo $O(nd)$: per ogni coordinata (sono d in totale) si calcola lo spread in tempo $\Theta(n)$ e si prende dunque il massimo. La procedura `CHOOSE-PARTITION-VAL` è semplicemente la mediana che si può calcolare quindi in tempo $\Theta(n)$[9]. Nell'equazione 2.5 $\alpha = \beta = \frac{1}{2}$, da cui segue direttamente la tesi. Per quanto riguarda lo spazio usato dal

[9] Ad esempio utilizzando un algoritmo della selezione che operi in tempo lineare.

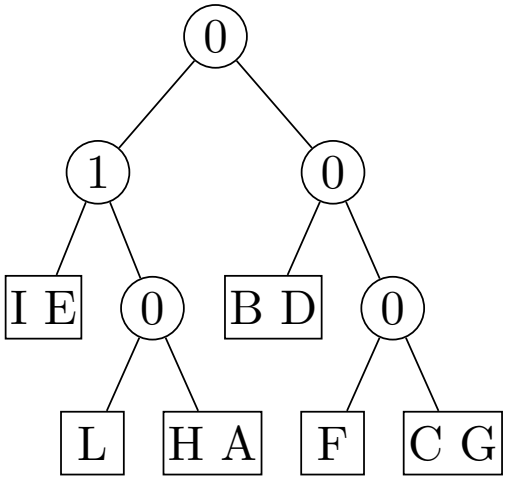
K-D Tree, basti notare che i punti vengono memorizzati una sola volta nelle foglie e i nodi dell'albero sono $O(n)$ (in particolare $\lceil \frac{n}{b} \rceil$ foglie e $\lceil \frac{n}{b} \rceil - 1$ nodi interni). □



(a) Partizionamento dello spazio del K-D Tree costruito su V. I valori di partizione dei vari nodi sono indicati da linee e le frecce disambiguano il nodo in cui stanno i punti con una coordinata sui valori di ripartizione (stanno in `hison`). La linea rossa (la più spessa) indica come viene partizionato lo spazio tra i nodi figli della radice. I due segmenti arancioni determinano la divisione dello spazio tra i 4 nodi figli dei figli della radice, infine i due segmenti giallo ocra (i più fini) dividono lo spazio per i 4 nodi precedenti che contengono più di 2 punti.



(b) K-D Tree costruito su V con evidenza sui bucket. Ogni bucket è contraddistinto da un colore. Ci sono 6 bucket, di cui 4 contenenti due punti e due contenenti un solo punto: il bucket grigio in basso a sinistra contiene solo L e il bucket arancione contiene solo F.



(c) Rappresentazione ad albero del K-D Tree costruito su V. I cerchi sono i nodi interni e contengono il valore del discriminatore. I quadrati sono i bucket.

Figura 2.3: K-D Tree risultante dall'insieme di punti $V = \{(1,2),(12,13),(122,53),(15,25),(1,26),(17,2),(19,92),(5,5),(6,52),(9,24)\}$ rispettivamente etichettati con A, B, ..., L. Dimensione dei bucket $b = 2$.

Le performance dell'algoritmo dipendono da:

1. n numero di punti
2. d dimensione dello spazio
3. m numero di vicini
4. b numero di punti massimo in un bucket (nodo foglia)
5. la misura della distanza
6. la densità dei punti nello spazio

Si noti che avendo scelto la mediana come valore di partizione, ogni bucket avrà da $\lceil \frac{b}{2} \rceil$ a b elementi e scegliendo di dividere sulla dimensione con più dispersione si garantisce che la forma geometrica dei bucket sia ragionevolmente compatta. Gli autori Bentley, Friedman e Finkel [4] (e Skrodzki [7] in una formulazione più recente) hanno dimostrato che il numero di punti R esaminati è limitato da:

$$R \leq (m^{\frac{1}{d}} + b^{\frac{1}{d}})^d \tag{2.6}$$

Questa espressione è minimizzata scegliendo $b = 1$. In questo caso:

$$R \leq (m^{\frac{1}{d}} + 1)^d \tag{2.7}$$

Si noti che per $m = 1$ si ha $R \leq 2^d$. Quindi si conclude che:

1. Il numero atteso di punti esaminati è indipendente da n.
2. Il numero di punti esaminati è minimizzato ponendo $b = 1$.

La costanza del numero di punti esaminati con il crescere dei punti, implica che il tempo richiesto per una query dell'algoritmo 11 è equivalente a cercare un punto in un albero binario bilanciato da cui si ricavano i seguenti lemmi.

Lemma 2.3. Il tempo atteso per un'esecuzione di una query dell'algoritmo 11 è $O(\log(n))$

Lemma 2.4. Il tempo atteso per la risoluzione del problema 1.1 con un'esecuzione dell'algoritmo 12 è $O(dn \log(n))$.

Si noti che:

- Il numero di punti visitati cresce meno che linearmente su m numero di best-matches a cui si è interessati.
- Nel caso peggiore (ad esempio con d elevato), per ogni query vengono visitati al più tutti i nodi e vengono calcolate le distanze di al più tutti i punti quindi l'algoritmo non è esponenziale.

Il prossimo lemma formalizza i risultati:

Lemma 2.5. Il caso peggiore per l'algoritmo 11 è $O(nd(\frac{1}{b} + \log(m)))$.

Dimostrazione. La complessità della procedura `BALL-WITTHIN-BOUNDS` è $\Theta(d)$ e quella della procedura `BOUNDS-OVERLAP-BALL` è $O(d)$. La complessità per l'algoritmo 11 vale quindi:

$$T(n) = \begin{cases} O(bd\log(m)) & \text{se } n \leq b \\ 2T(\frac{n}{2}) + O(d) & \text{altrimenti} \end{cases} \tag{2.8}$$

Svolgendo i calcoli si ottiene:

$$\begin{aligned} T(n) &= \textstyle\sum_{i=0}^{\log_2(\frac{n}{b})-1} k''d2^i + (k'bd\log(m))\frac{n}{b} \\ &= k''d\frac{2^{\log_2(\frac{n}{b})}-1}{2-1} + (k'bd\log(m))\frac{n}{b} \\ &= k''d(\tfrac{n}{b}-1) + (k'nd\log(m)) \\ &\in O(\tfrac{dn}{b} + nd\log(m)) \\ &= O(nd(\tfrac{1}{b} + \log(m))) \\ &= O(nd\log(m)) \end{aligned} \tag{2.9}$$

□

2.2.5 Analisi sulla dimensione dei bucket

Nella sezione precedente si è discusso come $b = 1$ minimizzi il numero di punti visitati durante una query, ma questa non è l'unica misura di performance dell'algoritmo. Come suggerito dalle equazioni 2.8 e 2.9, bisogna prendere in considerazione anche che per ogni nodo interno si ha costo $O(d)$ dovuto dalla chiamata alla procedura `BOUNDS-OVERLAP-BALL`. Se il limite geometrico è distante dal punto in considerazione, l'implementazione fornita dall'algoritmo 10 restituisce false esaminando poche coordinate e quindi viene velocemente escluso un ramo del K-D Tree. Se invece il limite geometrico è vicino e quindi c'è overlap, la procedura considera tutte le dimensioni e arriva al caso pessimo di $\Theta(d)$. Questa situazione accade vicino alle foglie dell'albero, in cui i punti sono vicini tra loro e quindi sarebbe più efficiente visitare direttamente i nodi e omettere alcune chiamate al test. Con un punto per bucket il test deve essere effettuato per ogni punto vicino al punto in considerazione, nella parte bassa dell'albero. Con diversi punti per bucket il test deve essere effettuato solo una volta per bucket. Bentley, Friedman e Finkel mostrano empiricamente [4] come questa intuizione sia vera e che è più efficiente avere bucket più grandi di $b = 1$ anche se questo fa aumentare il numero di punti esaminati. Anche questo comportamento si mostra essere indipendente dalla dimensione d, dal numero di best-matches m e dal numero di punti n.

3

Spazi con condizioni periodiche

Quello che spesso accade è che nel contesto della stima dell'entropia si ha a che fare con spazi con condizioni periodiche. L'algoritmo 2, con un'opportuna definizione della funzione `CALC-DISTANCES` continua a funzionare. Già considerando l'algoritmo 1 che funzionava efficientemente nel caso unidimensionale si iniziano a vedere i problemi: l'algoritmo ordina i punti e anche con un'opportuna definizione della funzione che calcola le distanze, non prende in considerazione i punti che si trovano nella parte opposta. Questo problema lo si trova anche utilizzando gli algoritmi proposti per i Quad Tree ed i K-D Tree.
Per costruire un algoritmo che risolva il problema 1.1 con condizioni periodiche, assumiamo di ricevere in input non solo gli n punti ma anche un vettore di coppie $bounds = ((inf_1, sup_1), \ldots, (inf_d, sup_d))$ in cui per ogni coordinata vengono specificate le condizioni periodiche. Ad esempio se un'informazione oraria è costituita da una coppia $(ore, minuti)$, abbiamo $bounds = ((0, 23), (0, 59))$.

3.1 K-D Tree modificati

Esistono sostanzialmente 2 soluzioni [19] che si possono applicare all'algoritmo originario per farlo funzionare con questo tipo di spazi. Tuttavia nessuna delle 2 sarà applicabile per punti in alta dimensionalità.

3.1.1 Aumentare i punti nello spazio

Si consideri per semplicità il caso unidimensionale. Sia $l = |bounds_1.sup - bounds_1.inf|$. La definizione della distanza rimane invariata, ma per dare l'effetto di uno spazio periodico si crea l'insieme V' contenente $3n$ punti: V' contiene tutti i punti in V ma vengono aggiunti n punti sommando ad ogni punto in V la quantità l e vengono aggiunti ulteriori n punti sottraendo l ad ogni punto in V. Questo nuovo insieme V' viene utilizzato per creare il K-D Tree con l'algoritmo proposto nel capitolo precedente, ma solo i punti in V vengono utilizzati nell'algoritmo di query. Ora però il risultato può contenere dei punti che non erano nello spazio di partenza, quindi è necessario effettuare l'operazione di shifting inversa.

Se $p(m) = \{q_{p_1}, \ldots, q_{pm}\}$ è l'insieme degli m vicini per p, si ottiene la vera soluzione $p(m)' =$

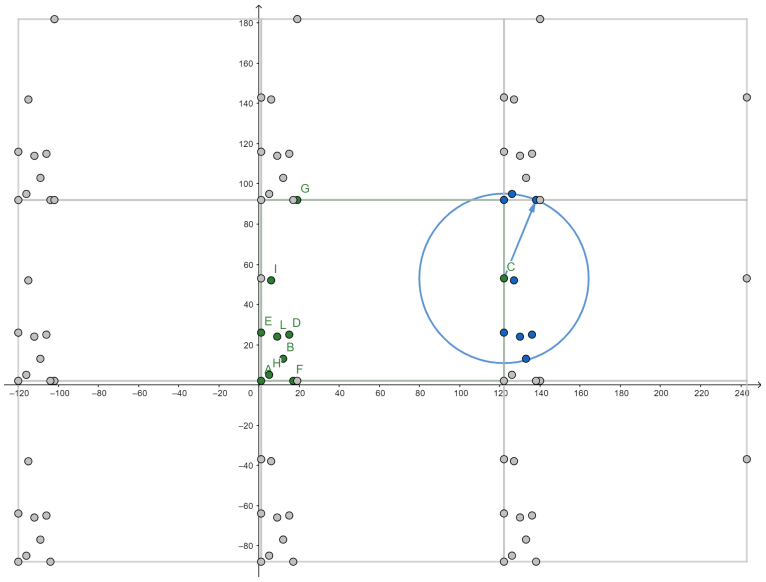


Figura 3.1: Insieme di punti V' partendo dall'insieme di punti $V = \{(1,2),(12,13),(122,53),(15,25),(1,26),(17,2),(19,92),(5,5),(6,52),(9,24)\}$ con $bounds = ((1,122),(2,92))$. In blue sono evidenziati gli 8 vicini per $C = (122,53)$.

$\{q'_{p_1}, \ldots, q'_{pm}\}$ in cui:

$$\forall i \in 1 \ldots m \; q'_{p_i} = \begin{cases} q_{p_i} & \text{se } bounds_1.inf \leq q_{p_i} \leq bounds_1.sup \\ q_{p_i} - l & \text{se } q_{p_i} < bounds_1.inf \\ q_{p_i} + l & \text{se } q_{p_i} > bounds_1.sup \end{cases} \tag{3.1}$$

Più in generale con d generico è necessario ripetere questo processo per ogni coordinata e quindi il nuovo insieme V' contiene $n(3^d)$ punti. In figura 3.1 è mostrato l'effetto delle $3^d - 1$ duplicazioni di punti nello spazio bidimensionale utilizzato anche negli esempi delle strutture K-D Tree (si veda figura 2.3a) e Quad Tree (si veda figura 2.1).

Analisi

Lo spazio richiesto dal K-D Tree è ora $O(n3^d)$, il tempo di costruzione del K-D Tree è $O(n3^d \log(n3^d))$ e il tempo atteso per una query è $O(n3^d \log(n3^d))$. In conclusione questo metodo può andare bene solo per d molto piccolo ad esempio compreso tra 1 e 3.

3.1.2 Aumentare le query

Un approccio simmetrico al precedente è quello di lasciare invariato il K-D Tree che occuperà quindi spazio $O(n)$, ma per ogni punto p vengono effettuate 3^d query in cui ogni volta viene shiftata opportunamente una coordinata di p. In questo caso fissato un punto p, vanno mantenute due heap: la prima è quella classica dell'algoritmo 12 che non solo mantiene il vicinato ma permette anche di scartare dei rami dell'albero in base a certe condizioni. Questa prima heap viene inizializzata vuota ad ogni nuova query. La seconda heap invece tiene conto della soluzione globale per p, ovvero ogni volta che aggiungiamo un

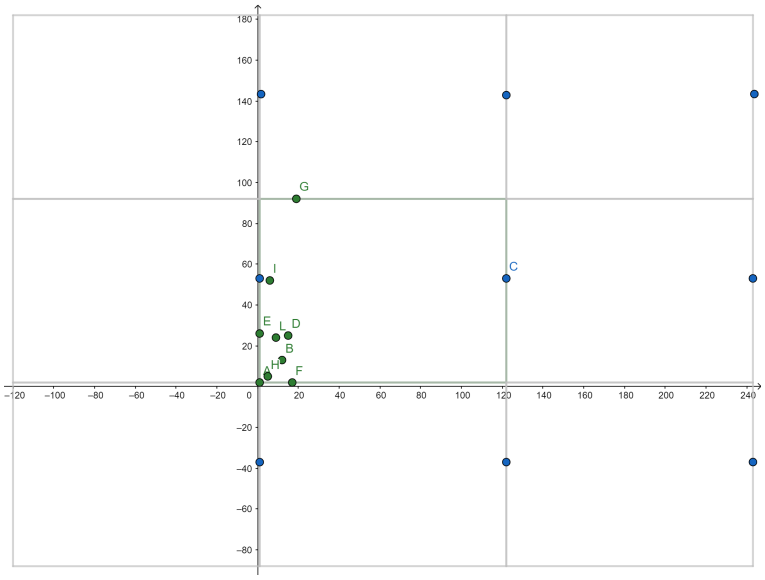


Figura 3.2: Punti risultanti dalla procedura `CREATE-IMAGES` eseguita su $C = (122, 53)$ partendo dall'insieme di punti $V = \{(1, 2), (12, 13), (122, 53), (15, 25), (1, 26), (17, 2), (19, 92), (5, 5), (6, 52), (9, 24)\}$ e $bounds = ((1, 122), (2, 92))$. I punti in V sono rappresentati in verde e i punti duplicati in blue.

punto eseguendo una query su un'immagine di p, aggiorniamo anche questa heap che alla fine delle 3^d query per p ne conterrà il vicinato.

Nel listato 13 vengono formalizzate queste idee e viene presentato lo pseudo codice delle procedure modificate; le procedure non presenti nel listato sono le stesse presentate nella sezione 2.2.3. La heap *mh* è quella che guida la discesa nell'albero mentre la heap *supportMh* tiene conto della soluzione per p in tutte le 3^d discese. La procedura `CREATE-IMAGES` è implementabile in tempo $O(3^d)$ e crea tutte le immagini per p, ovvero crea tutti i punti p' shiftati opportunamente in ciascuna coordinata considerando i *periodicBounds*.

In figura 3.2 sono mostrati tutti i punti p', ottenuti dalla procedura `CREATE-IMAGES` con input un punto p.

Analisi

La complessità della procedura `VISIT-LEAF` rimane invariata, infatti una chiamata alla procedura `UPDATE-NEIGHBOURHOOD` prende tempo $O(\log(m))$. La complessità in tempo dell'algoritmo che risolve il problema 1.1 con condizioni di periodicità mediante i K-D Tree vale $O(n3^d \log(n3^d))$ e la complessità in spazio vale $O(n)$. Questa soluzione è migliore della precedente in quanto almeno lo spazio non diventa esponenziale ma per quanto riguarda il tempo valgono le stesse considerazioni fatte in sezione 3.1.1.

Algorithm 13 ALL-MNN con condizioni periodiche utilizzando i K-D Tree: codice modificato

```
global b                                              ▷ Size of buckets.
global m                                              ▷ Number of neighbors.
global mh                                             ▷ The max-priority queue.
global supportMh                                      ▷ Another max-priority queue.
global upperBounds
global lowerBounds
global periodicBounds                                 ▷ Array of couples containing the periodic conditions
procedure ALL-MNN(points)
    T ← BUILD-KD-TREE(points)
    sol ← NEW-ARRAY(n)
    for i ← 1 to n do
        p ← points[i]
        supportMh ←NEW-MAX-HEAP(m)
        images ←CREATE-IMAGES(p, periodicBounds)
        for k ← 1 to 3^d do
            mh ←NEW-MAX-HEAP(m)
            upperBounds ← (∞, ..., ∞)
            lowerBounds ← (−∞, ..., −∞)
            QUERY(images[i], T)
        end for
        sol[i] ← COPY-HEAP(supportMh)
    end for
    return sol
end procedure
procedure VISIT-LEAF(p, u)
    for each point q ≠ p in u.setOfPoints do
        d_max, q_max ← GET-HEAP-MAX(mh)               ▷ q_max is the point. d_max is the distance from p.
        dist ←CALCULATE-DISTANCE(p, q)
        if GET-HEAP-LENGTH(mh) < m then
            HEAP-INSERT(mh, dist, q)
            UPDATE-NEIGHBOURHOOD(dist, q)
        else if dist < d_max then
            HEAP-DEQUEUE(mh, q_max)
            HEAP-INSERT(mh, dist, q)
            UPDATE-NEIGHBOURHOOD(dist, q)
        end if
    end for
end procedure
procedure UPDATE-NEIGHBOURHOOD(dist, q)
    d_max, q_max ← GET-HEAP-MAX(supportMh)
    if GET-HEAP-LENGTH(supportMh) < m then
        HEAP-INSERT(supportMh, dist, q)
    else if dist < d_max then
        HEAP-DEQUEUE(supportMh, q_max)
        HEAP-INSERT(supportMh, dist, q)
    end if
end procedure
```

3.2 VP Tree

Come visto nella sezione precedente i K-D Tree non risultano essere una buona soluzione per il problema del vicinato con condizioni periodiche, infatti i K-D Tree funzionano solo con uno spazio euclideo e categorizzano i punti basandosi sulla loro proiezione in uno spazio a dimensionalità più bassa (per questo vengono classificati come *projective trees*). Per ottenere la soluzione al problema con condizioni di periodicità bisogna simulare lo spazio aumentando esponenzialmente il tempo per le query e/o lo spazio di memorizzazione dell'albero. Si rende necessario considerare quindi una nuova struttura. I VP Tree sono stati originariamente proposti da Yianilos [8] che come obbiettivo aveva quello di proporre una struttura più generale che partizionasse i punti basandosi solo sull'informazione data dalla funzione della distanza.

> « *So despite considerable progress for Euclidian space, the development of more general techniques is important; not just because of the problems with high dimension, but also because there is no a priori reason to believe that all or even most useful metrics are Euclidian.* » [8]

Indipendentemente da Yianilos, Uhlmann [9] propone la stessa struttura chiamandola *metric tree*; questa tipologia di albero infatti, lavora in un generico spazio metrico. Si ricorda la definizione di spazio metrico.

Definizione 3.1 (Spazio metrico)**.** Uno spazio metrico è una coppia ordinata (M, d) dove M è un insieme e d è una metrica su M ovvero una funzione $d : M \times M \to R$ tale che per ogni $x, y, z \in M$ valgono:

1. $d(x, y) = 0 \iff x = y$ (Identità degli indiscernibili)
2. $d(x, y) = d(y, x)$ (Simmetria)
3. $d(x, y) \leq d(x, z) + d(z, y)$ (Disequazione triangolare)

Dalle proprietà enunciate si ricava facilmente la non negatività di d ossia $d(x, y) \geq 0$ per ogni $x, y \in M$.

3.2.1 La struttura dati

La struttura dei VP Tree (Vantage Point Tree ossia 'alberi a punto di vantaggio') è stata proposta in numerose varianti. Yianilos propone [8] la struttura base insieme ai VP^s Tree e ai VP^{sb} Tree. In questa sede utilizzeremo una variante base dei VP Tree simile all'originaria ma con la differenza che tutti i punti dell'insieme di partenza V saranno memorizzati nelle foglie come nei K-D Tree descritti in sezione 2.2.1. La struttura generale è molto simile ai K-D Tree, si tratta di un albero binario, ma in ogni nodo ci saranno dei campi aggiuntivi che permettono di partizionare i punti. L'intuizione di questa struttura dati è che ogni elemento dello spazio metrico ha una sua prospettiva su tutto lo spazio, formata considerando la sua distanza con quella di tutti gli altri punti.

Definizione 3.2 (VP Tree su un insieme di n punti V)**.** Un VP Tree è un albero binario tale che:

1. La radice contiene e rappresenta tutti i punti in V, ogni nodo rappresenta un sottoinsieme di V.

2. Ogni nodo ha 0 o 2 figli uno destro e uno sinistro.
3. Ogni nodo interno ha un campo `vantagePoint` che contiene un punto.
4. Ogni nodo interno ha un campo `radius` che contiene un numero reale.
5. Ogni punto $p \in V$ viene memorizzato in esattamente una foglia.
6. Ogni nodo foglia è un *bucket* e contiene al più b punti.
7. I nodi foglia rappresentano piccoli insiemi mutamente esclusivi di punti in V.
8. **Proprietà VP Tree:** Per ogni nodo n, sia $vp = n.vantagePoint$, per ogni punto $p = (p_1, \ldots, p_d)$ contenuto nel sottoalbero radicato in `n.left` vale $d(p, vp) < n.radius$ e analogamente per ogni punto p nel sottoalbero radicato in `n.right` vale $d(p, vp) \geq n.radius$.

Si noti che le proprietà 1,2,5,6,7 sono le stesse della definizione dei K-D Tree (si veda la sezione 2.2.1). Quello che in questo caso cambia è solo il modo di partizionare i punti tra i vari nodi, detto in altri termini, in ogni nodo viene scelto un punto di vantaggio vp e un raggio r, tutti i punti che stanno nel sottoalbero sinistro, sono contenuti nell'ipersfera centrata nel punto di vantaggio vp e di raggio r; analogamente tutti i punti che stanno nel sottoalbero destro non sono contenuti in tale ipersfera. Si noti che nella definizione non viene specificato come scegliere il punto di vantaggio nè come scegliere il raggio.

3.2.2 Costruire il VP Tree

Si supponga che la struttura di un nodo sia:

```
struct node{
   bool isLeaf;
   point* vantagePoint;
   real radius;
   node* left;
   node* right;
   points* setOfPoints;
}
```

in cui il campo `setOfPoints` è settato solo nelle foglie. La procedura di costruzione del VP Tree è del tutto analoga a quella dei K-D Tree.

L'algoritmo 14 formalizza queste idee. Viene inoltre fornita la procedura `CHOOSE-RADIUS` nel modo più semplice possibile: vengono calcolate le distanze di tutti i punti con il punto di vantaggio e si prende come raggio la distanza mediana, questo garantisce che in media la metà dei punti finiscano nel figlio di sinistra ossia che l'albero costruito sia bilanciato. Il tempo atteso di costruzione è $O(nd\log(n))$.

Si noti che in realtà non è possibile dire a priori quanti punti finiranno nel sottoalbero destro o sinistro di un nodo, infatti se ci sono molti punti sulla superficie dell'ipersfera centrata in vp di raggio r, finiranno molti più punti a destra che a sinistra. Tuttavia ipotizzando che nella superficie di tale ipersfera finiscano pochi punti, si ottiene che il numero di punti che finiranno a destra e a sinistra è circa lo stesso e l'albero sarà bilanciato.

Algorithm 14 Costruzione di un VP Tree su un insieme di punti

procedure BUILD-VP-TREE($points$)
  if $|points| \leq b$ **then**
    return MAKE-LEAF($points$)
  end if
  $vp \leftarrow$CHOOSE-VANTAGE-POINT($points$)
  $r \leftarrow$CHOOSE-RADIUS($points, vp$)
  $points_{left} \leftarrow \{q = (q_1, \ldots, q_d) | q \in points, d(q, vp) < r\}$
  $points_{left} \leftarrow \{q = (q_1, \ldots, q_d) | q \in points, d(q, vp) \geq r\}$
  $node_{left} \leftarrow$ BUILD-VP-TREE($points_{left}$)
  $node_{right} \leftarrow$ BUILD-VP-TREE($points_{right}$)
  return MAKE-INTERNAL-NODE($vp, r, node_{left}, node_{right}$)
end procedure
procedure CHOOSE-RADIUS($points, vp$)
  $distances \leftarrow$ CALC-DISTANCES($points, vp$)
  $r \leftarrow$FIND-MEDIAN($distances$)
  return r
end procedure

Si noti inoltre che la computazione in un particolare caso può divergere e non terminare mai. Si pensi alla situazione in cui in ogni chiamata, scelto un vantage point e il raggio, esattamente tutti i punti si trovino sulla superficie dell'ipersfera di cui sopra; in questo caso ad ogni passo le chiamate ricorsive non partizionano lo spazio e tutti gli elementi finiscono a destra. Possiamo supporre che tale situazione non capiti mai considerando dataset reali. Questo problema è anche presente nell'algoritmo di costruzione dei K-D Tree: dato un nodo, scelto il discriminatore, viene effettuata la proiezione di tutti i punti sulla coordinata indicata e se le proiezioni sono tutte uguali al valore di partizione per quel nodo, la computazione analogamente diverge.

Per risolvere il problema nel caso dei VP Tree in cui $|points_{left}| = 0 \ \vee \ |points_{right}| = 0$ è sufficiente scegliere un nuovo punto di vantaggio. Nella rara possibilità in cui vale:

$$\forall p \mid p \text{ scelto come punto di vantaggio } (|points_{left}| = 0 \ \vee \ |points_{right}| = 0) \tag{3.2}$$

risulta necessario considerare tale nodo una foglia anche se la cardinalità è maggiore di b. Una strategia analoga si può adottare nei K-D Tree.

Scelta del punto di vantaggio

Per ora si consideri che il punto di vantaggio sia scelto casualmente tra quelli ricevuti in input. Questa scelta è perfettamente lecita in accordo con la definizione 3.2 ma la discussione su come effettuare una selezione intelligente di tale punto è lasciata alle sezione 3.2.5. Si ricorda che il raggio invece viene sempre scelto in modo da dividere i punti equamente tra le due porzioni di spazio.

3.2.3 Query

L'algoritmo di query anche in questo caso è molto simile a quello dei K-D Tree. Viene di seguito descritto l'algoritmo proposto in [10] che è la naturale generalizzazione a m vicini di quello originariamente proposto da Yianilos in [8] che calcola un solo vicino. Il funzionamento è il seguente. Si scende l'albero

con una visita *depth-fisrt* partendo dalla radice e si mantiene una heap degli m vicini trovati. L'ordine di attraversamento (ovvero se scendere prima nel ramo destro o nel sinistro), cambia sensibilmente le performance ed è una scelta che va effettuata accuratamente. Sia τ una soglia che indica la distanza massima entro cui trovare l' m-imo vicino di p. Ponendo inizialmente $\tau = \infty$ si indica che l' m-imo vicino di p possa trovarsi ovunque nello spazio, ponendo un valore finito invece, le performance dell'algoritmo migliorano ma non è detto che la soluzione restituita sia quella corretta ossia non vi è garanzia che nell'ipersfera centrata in p con raggio τ si trovino almeno m punti.

Da qui si supponga dunque che inizialmente $\tau = \infty$ e che il valore venga aggiornato durante la ricerca: da quando la heap contiene m punti, τ all'istante t sarà la distanza $d(p, p_{m-th})$ dove p_{m-th} è l'm-imo vicino di p trovato all'istante di tempo t. Il significato è che si è interessati alla ricerca di una soluzione, eventualmente migliore dell'attuale, solo nell'ipersfera centrata in p, di raggio τ.

Se in un nodo n, ricercando il vicinato per p, si ha $d(p, n.vantagePoint) \geq n.radius + \tau$ (si veda la figura 3.3a) allora il sottoalbero sinistro di n può essere escluso dalla ricerca poiché nessuno degli m vicini potrà trovarsi in quella porzione di spazio. Similmente se troviamo $d(p, n.vantagePoint) < n.radius - \tau$ (si veda la figura 3.3b) si può ignorare il sottoalbero destro.

È quindi chiaro che la sola informazione portata dalla prospettiva di un singolo punto può essere sufficiente, in alcuni casi, per tagliare la ricerca. Se invece $n.radius - \tau < d(p, n.vantagePoint) < n.radius + \tau$ (si veda la figura 3.3c) allora bisogna necessariamente visitare entrambi i rami.

Da queste considerazioni si intuisce che la capacità di *pruning* dipende anche da una buona scelta di $n.vantagePoint$. Se il punto p è contenuto nell'ipersfera centrata in $n.vp$ con raggio $n.radius$, si eseguono prima la chiamata sul figlio sinistro e poi quella sul figlio destro (se necessaria) altrimenti prima la chiamata sul figlio destro ed eventualmente quella sul figlio sinistro in seguito. L'algoritmo 15 formalizza queste idee. Si noti che tra i 4 rami `if` più interni alla procedura `QUERY` che contengono le chiamate ricorsive alla procedura stessa, almeno una chiamata viene eseguita, infatti dalla non negatività della funzione di distanza[1] valgono:

$$\begin{aligned} d(p, n.vantagePoint) < n.radius &\implies d(p, n.vantagePoint) < n.radius + \tau \\ d(p, n.vantagePoint) \geq n.radius &\implies d(p, n.vantagePoint) \geq n.radius - \tau \end{aligned} \tag{3.3}$$

dove n è un nodo e si sta considerando una query per p.

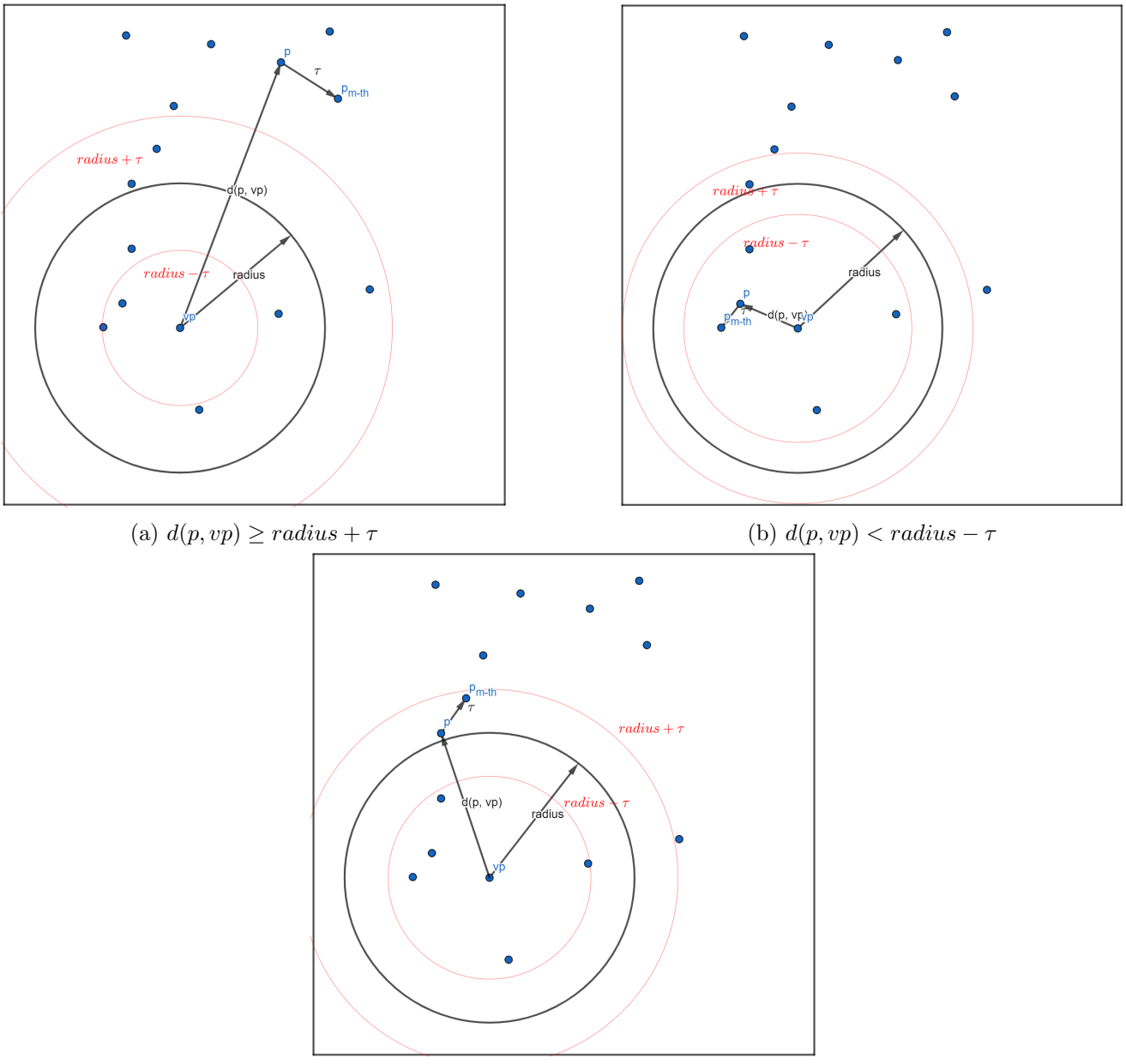
3.2.4 Analisi

Come i K-D Tree, i VP Tree possono essere memorizzati in spazio $O(n)$. L'equazione ricorsiva per la complessità della procedura `QUERY` nel listato 15 è:

$$T(n) = \begin{cases} O(bd \log(m)) & \text{se } n \leq b \\ 2T(\frac{n}{2}) + \Theta(d) & \text{altrimenti} \end{cases} \tag{3.4}$$

infatti nella procedura `QUERY`, calcolare una distanza costa $\Theta(d)$ che è equivalente al costo dei test `BALL-WITHIN-BOUNDS` e `BOUNDS-OVERLAP-BALL` nei K-D Tree. Si conclude quindi con il costo del caso

[1] Si ricorda che τ è una distanza.



(a) $d(p, vp) \geq radius + \tau$

(b) $d(p, vp) < radius - \tau$

(c) $radius - \tau < d(p, vp) < radius + \tau$

Figura 3.3: Caso bidimensionale, si consideri una query per p. p_{m-th} è l'm-imo vicino di p finora trovato, $\tau = d(p, p_{m-th})$. Il cerchio nero rappresenta il boundary di un nodo con con punto di vantaggio vp e raggio $radius$. I cerchi rossi rappresentano rispettivamente $radius - \tau$ e $radius + \tau$.

Algorithm 15 ALL-MNN utilizzando i VP Tree

```
global b                                              ▷ Size of buckets.
global m                                              ▷ Number of neighbors.
global τ                                              ▷ The threshold which limits the search space.
global mh                                             ▷ The max-priority queue.
procedure ALL-MNN(points)
    T ← BUILD-VP-TREE(points)
    sol ← NEW-ARRAY(n)
    for i ← 1 to n do
        p ← points[i]
        mh ←NEW-MAX-HEAP(m)
        τ ← +∞
        QUERY(p, T)
        sol[i] ← COPY-HEAP(mh)
    end for
    return sol
end procedure
procedure QUERY(p, u)
    if u.isLeaf = true then
        VISIT-LEAF(p, u)
    else
        dist ← CALCULATE-DISTANCE(u.vantagePoint, p)
        if dist < u.radius then                       ▷ Test used to decide the search order.
            if dist < u.radius + τ then
                QUERY(p, u.left) end if
            if dist ≥ u.radius − τ then
                QUERY(p, u.right) end if
        else
            if dist ≥ u.radius − τ then
                QUERY(p, u.right) end if
            if dist < u.radius + τ then
                QUERY(p, u.left) end if
        end if
    end if
    return
end procedure
procedure VISIT-LEAF(p, u)
    for each point q ≠ p in u.setOfPoints do
        d_max, q_max ← GET-HEAP-MAX(mh)               ▷ q_max is the point. d_max is the distance from p.
        dist ←CALCULATE-DISTANCE(p, q)
        if GET-HEAP-LENGTH(mh) < m then
            HEAP-INSERT(mh, dist, q)
        else if dist < d_max then
            HEAP-DEQUEUE(mh, q_max)
            HEAP-INSERT(mh, dist, q)
        end if
        if GET-HEAP-LENGTH(mh) = m then
            τ ← dist end if
    end for
end procedure
```

peggiore identico a quello del caso peggiore dei K-D Tree. Per i calcoli completi si veda l'equazione 2.9 nella sezione 2.2.4.

Anche il tempo medio è lo stesso: il tempo atteso per la procedura `QUERY` nell'algoritmo 15 è $O(dn\log(m))$ ed empiricamente si mostra (si vedano [8, 10]) che il comportamento dei VP Tree è lo stesso e in alcuni casi migliore rispetto a quello dei KD Tree. Se si pensa a quanto i risultati dei KD Tree siano considerati eccellenti, i VP Tree stupiscono ancora di più infatti riescono ad ottenere lo stesso risultato pur essendo ignoranti rispetto alla struttura delle coordinate dello spazio.

I calcoli in questa sezione sono stati eseguiti considerando una scelta casuale del vantage point. Casi particolari saranno analizzati in sezione 3.2.5. Si noti che la complessità del caso peggiore $O(dn\log(m))$ è indipendente dal bilanciamento dell'albero: con un albero bilanciato quello che cambierà sarà il costo del caso medio.

3.2.5 Scelta del punto di vantaggio

La scelta banale del punto di vantaggio è quella di considerare casualmente uno tra i punti ricevuti in input in un dato nodo. Tuttavia Yianilos [8] osserva che spendere del tempo per scegliere un miglior vantage point può portare a dei risparmi di tempo non banali. L'obbiettivo della scelta di un buon punto di vantaggio è in generale quello di diminuire la complessità in tempo dell'algoritmo 15; tuttavia questo può essere fatto in diversi modi ponendosi diversi obiettivi e alcuni di questi saranno di seguito discussi.

Punto di vantaggio ottimale

Come visto in sezione 3.2.3 considerando un'interrogazione per p, in un nodo n, vengono visitati entrambi i sottoalberi se troviamo $n.radius - \tau < d(p, n.vantagePoint) < n.radius + \tau$ (si riveda la figura 3.3c). Per minimizzare la probabilità che vengano effettuate le ricerche su entrambi i rami si vuole minimizzare la porzione di spazio dell'ipersfera con raggio $n.radius + \tau$ centrata in $n.vantagePoint$ sottratta all'ipersfera con stesso centro ma con raggio $n.radius - \tau$. Detto in altri termini si vuole minimizzare la superficie dell'ipersfera centrata in $n.vantagePoint$ e di raggio $n.radius$. Non avendo la possibilità di manipolare τ si ha invece la possibilità di manipolare il centro dell'ipersfera ovvero il punto di vantaggio. Yianilos [8] mostra intuitivamente che in un quadrato unitario, il centroide è la scelta peggiore mentre i punti negli angoli dello spazio sono la scelta migliore (si rimanda a [8, Figure 4 - p.315]).

Un punto di vantaggio ottimale deve avere la seguente proprietà [11]:

- La distribuzione dei valori delle distanze tra il punto di vantaggio e gli altri punti è vicina ad una distribuzione uniforme.

Intuitivamente avendo questa proprietà il numero di punti nella regione di spazio concentrica è minimizzato quindi la probabilità di esplorare entrambi i sottoalberi di un nodo è minima. Si noti inoltre che un nodo con un punto di vantaggio che soddisfi questa proprietà avrà i figli destro e sinistro approssimativamente con lo stesso numero di punti.

Scegliere un punto di vantaggio ottimale è però molto costoso e per questa ragione nella pratica, non viene mai utilizzata questa tecnica negli algoritmi di costruzione.

Algoritmo randomizzato

Come concluso nella sezione precedente scegliere il punto di vantaggio ottimale è troppo costoso, per questo si procede usualmente con l'algoritmo randomizzato proposto da Yianilos [8] e dimostrato da lui stesso avere buone performance. Il listato 16 riporta l'algoritmo di Yianilos [8].

Algorithm 16 Algoritmo randomizzato per il calcolo del punto di vantaggio

```
procedure CHOOSE-VANTAGE-POINT(points)
    S_1 ← CHOOSE-RANDOM-SET(points, n_1)
    best_vp ← null
    max_dev ← 0
    for each vp in S_1 do
        S_2 ←CHOOSE-RANDOM-SET(points, n_2)
        distances ← CALC-DISTANCES(S_2, vp)
        μ ← CALCULATE-MEAN(distances)
        σ ← CALCULATE-STANDARD-DEVIATION(μ, distances)
        if σ ≥ max_dev then
            best_vp ← vp
            max_dev ← σ
        end if
    end for
    return best_vp
end procedure
```

Dati un insieme di punti considerati in un nodo, viene richiamata la procedura `CHOOSE-VANTAGE-POINT` sullo stesso insieme di punti. Viene estratto casualmente un primo insieme di punti S_1 di cardinalità n_1 che costituisce l'insieme dei punti candidati a diventare punti di vantaggio. Per ogni candidato vp viene estratto un secondo insieme di punti S_2 di cardinalità n_2 e viene calcolata la deviazione standard della distanza tra vp e i punti in S_2. Il punto di vantaggio restituito è quello con la massima deviazione standard. Questo algoritmo cerca di stimare un punto di vantaggio che soddisfi la proprietà del punto di vantaggio ottimale.

L'algoritmo prende tempo $\Theta(n_1 n_2 d)$ pertanto per fare in modo che il tempo di costruzione dell'albero rimanga $O(nd\log(n))$ (si veda la sezione 3.2.2) è necessario che $n_1 n_2 \in O(n)$, ad esempio una scelta ragionevole è quella di porre $n_1 = n_2 = \sqrt{n}$.

Minimizzare il computo delle distanze

In [10] viene osservato che al crescere di d il calcolo delle distanze diventa più inefficiente e durante un'interrogazione per p, in ogni nodo interno visitato viene effettuato un calcolo della distanza. L'autore di [10] propone di utilizzare un singolo punto di vantaggio per ogni livello, in questo modo durante la ricerca verrà effettuato un solo calcolo per livello. Il vantage point della radice viene scelto con l'algoritmo 16 della sezione precedente mentre, per ottenere un partizionamento efficace dello spazio, si richiede che il secondo punto di vantaggio sia il punto più distante dal primo; il terzo punto di vantaggio deve essere il punto più distante tra il secondo e il primo e così via. Si noti ora che a differenza dell'algoritmo originario, dato un nodo non è detto che il suo punto di vantaggio associato, ricada nella regione di spazio che il nodo rappresenta.

Rieseguendo i calcoli della complessità dell'algoritmo di query si ottiene:

$$\begin{aligned} T(n) &= \Theta(d\log(\tfrac{n}{b})) + \begin{cases} O(bd\log(m)) & \text{se } n \leq b \\ 2T(\tfrac{n}{2}) + \Theta(1) & \text{altrimenti} \end{cases} \\ &= \Theta(d\log(\tfrac{n}{b})) + O(\tfrac{n}{b} + nd\log(m)) \\ &= O(dn\log(m)) \end{aligned} \tag{3.5}$$

La complessità asintotica quindi non cambia, ma confrontando i risultati con l'equazione 2.9 in sezione 2.2.4 (che è l'equazione per la procedura `QUERY` dei VP Tree senza nessuna scelta particolare del vantage point) è interessante notare che alcuni termini hanno meno peso da cui si ricava che nella pratica questo metodo riduce effettivamente i calcoli necessari.

3.2.6 Query in spazi con condizioni periodiche

Come precedentemente analizzato, i VP Tree dipendono solo dalla funzione della distanza utilizzata e da nient'altro, quindi a patto di fornire un'adeguata funzione, i VP Tree possono essere utilizzati così come sono stati presentati, senza ulteriori modifiche, nel caso in cui si lavori con spazi con condizioni periodiche. Il listato 17 fornisce una possibile definizione della procedura `CALC-DISTANCE`. Con questa funzione, fissato un insieme di punti V, eseguendo l'algoritmo 15 e l'algoritmo modificato dei K-D Tree 13 si ottiene la stessa soluzione ma in tempo decisamente a vantaggio dei VP Tree. Il tempo atteso per `ALL-MNN` nel caso dei VP Tree, rimane $O(n\log(n))$ contro $O(n3^d\log(n3^d))$ (si veda la sezione 3.1.2).

Algorithm 17 Una definizione di distanza per spazi con condizioni periodiche

procedure `CALC-DISTANCE`$(p = (p_1, \ldots, p_d), q = (q_1, \ldots, q_d), d, bounds)$
  $dist \leftarrow 0$
  $l \leftarrow$ `NEW-ARRAY`(n) ▷ The array which stores the lengths of the bounds for each coordinate.
  for $i = 1$ **to** d **do**
    $l_i \leftarrow |bounds_i.sup - bounds_i.inf|$
    $dist \leftarrow dist + \min\left\{(p_i - q_i)^2, (p_i - (q_i - l_i))^2, (p_i - (q_i + l_i))^2\right\}$
  end for
  return $\sqrt{dist}$
end procedure

3.2.7 Varianti dei VP Tree e lavori correlati

Vengono ora citati ed elencati alcuni lavori e ricerche aperte riguardanti i VP Tree e le relative varianti.

VP Tree di arietà k

Un primo aspetto che viene considerato è dove memorizzare il VP Tree. Se viene memorizzato in memoria secondaria, è necessario minimizzare gli accessi al disco e le foglie vanno dimensionati appropriatamente. La tecnica che si usa in questo caso [11] è quella di aumentare il *fanout* similmente a quanto accade nei B Tree. In un VP Tree di arietà k (con $k \geq 2$), in ogni nodo vengono calcolati $k - 1$ raggi $r_1, \ldots, r_k$

e ogni nodo ha k figli che denotano gli insiemi di punti $S_1, \ldots, S_k$. Fissato un nodo n, ponendo per convenzione $r_0 = 0$, deve valere:

$$p \in S_i \implies r_{i-1} < d(p, vp) \leq r_i \tag{3.6}$$

Data una query q, bisogna esplorare l'insieme S_i solo se $r_{i-1} - \tau < d(p, vp) \leq r_i + \tau$. Con questa variante, in un nodo n è possibile escludere $\frac{k-1}{k}$ punti dall'insieme di punti che n contiene e quindi il tempo di ricerca diminuisce (anche se la complessità asintotica rimane la stessa). È inoltre possibile scegliere k e b per ottimizzare sensibilmente le performance del disco:

- Si scelga il massimo k tale che in una pagina di memoria vengano memorizzati esattamente k nodi interni. Fissato un nodo interno, se i suoi k figli sono anch'essi nodi interni vengono memorizzati in una pagina di memoria.
- Si scelga il massimo b per cui tutti e soli i punti di una foglia vengano memorizzati in una pagina di memoria.

Multi Vantage Point Tree

Ozsoyoglu e Bozkaya [12] introducono la struttura MVP Tree (ossia Multi VP Tree). Le idee adottate sono due:

1. Ogni nodo interno ha due punti di vantaggio.
2. Nei nodi foglia vengono mantenute le distanze tra i punti e i punti di vantaggio.

Nella costruzione dell'MVP Tree si tiene conto che in un dato nodo, il relativo punto di vantaggio può essere fuori dalla regione di spazio che il nodo rappresenta e in particolare lo stesso punto di vantaggio può essere utilizzato per tutti i nodi in un determinato livello dell'albero.

Ogni nodo dell'MVP Tree può essere visto come due livelli del VP Tree in cui tutti i nodi del livello più in basso adottano lo stesso punto di vantaggio. Sia k il numero di partizioni che crea un punto di vantaggio. In un MVP Tree binario il primo punto di vantaggio vp_1 partiziona lo spazio in due parti e il secondo punto di vantaggio vp_2 partiziona ognuna di queste due parti in altre due parti e quindi il *fanout* vale 4. In generale in un nodo n, vp_1 partiziona n in k parti e vp_2 partiziona ognuna di queste parti in altre k e quindi il *fanout* è k^2.

I 2 punti di vantaggio possono essere generalizzati a v punti di vantaggio: in questo caso un nodo può essere visto come v livelli di un classico VP Tree e con questa variante il *fanout* vale v^k.

Per quanto riguarda il punto 2 si osservi che nell'algoritmo di costruzione classico dei VP Tree, per ogni punto p contenuto in una foglia f, vengono calcolate le distanze tra p e tutti i punti di vantaggio nel percorso tra la radice ed f. In [12] si mostra che è possibile mantenere queste informazioni per migliorare le operazioni di ricerca: in particolare si usano le distanze precalcolate in costruzione per escludere il calcolo della distanza tra il punto di query e alcuni punti nelle foglie. Gli autori Ozsoyoglu e Bozkaya [12] mostrano infine che gli MVP Tree con k basso funzionano meglio e sono sensibilmente più veloci della variante base dei VP Tree. Viene inoltre osservato che ogni euristica per la scelta del punto di vantaggio nei VP Tree può essere applicata anche negli MVP Tree.

Lavorare in alta dimensionalità

Anche i VP Tree non sfuggono al problema *curse-of-dimensionality* quindi è desiderabile lavorare con meno dimensioni possibili, ma mappare un insieme di punti ad alta dimensionalità in uno spazio a bassa dimensionalità fa perdere troppa informazione e quindi i vicini trovati nell spazio a bassa dimensionalità possono non essere dei 'veri' vicini.

L'idea che si usa in questo caso [11] è quella di utilizzare più VP Tree ognuno dei quali lavora su un piccolo sottoinsieme specifico di dimensioni. Il problema è ora che (come mostrato da Chiueh in [11]) unire le soluzioni dei vari Tree non è banale e non sempre è possibile trovare una soluzione esatta al problema del vicinato.

4

Implementazione e sperimentazione

Dopo avere visto una serie di algoritmi ciascuno con il proprio pseudo-codice, e una serie di risultati teorici, in questo capitolo verrà presentato il lavoro svolto nella reale implementazione in C degli algoritmi presi in considerazione.

4.1 Aspetti implementativi

Come anticipato sono stati implementati in C i seguenti programmi che risolvono il problema del vicinato: Naive, Quad Tree (versione generalizzata per m e d arbitrari), K-D Tree e VP Tree. Nel caso degli algoritmi Naive, K-D Tree e VP Tree sono stati creati anche i programmi che funzionano in spazi con condizioni periodiche.

4.1.1 Il linguaggio di programmazione

La scelta del linguaggio di programmazione è ricaduta sul C. Come ben noto il C è un linguaggio compilato molto efficiente e con istruzioni che permettono la gestione esplicita della memoria. Seppur più prolisso (rispetto ad esempio al python) il linguaggio C è l'ideale per l'implementazione e la valutazione sperimentale di algoritmi: la presenza dell'istruzione per la deallocazione esplicita della memoria assicura che a tempo di esecuzione non ci siano comportamenti inaspettati rispetto a quelli programmati. Se il linguaggio di programmazione scelto avesse incluso un *garbage collector* ad esempio, i tempi di misurazione presi durante l'esecuzione dell'algoritmo del vicinato sarebbero stati peggiori in quanto influenzati dall'esecuzione del *garbage collector* stesso che non è sotto il controllo del programmatore.

4.1.2 Scelte implementative

Tutti gli algoritmi in considerazione implementano una funzione `solve` della forma:
`float*** solve(float** points, int n, int m, int d)`
in cui se `r` è il risultato restituito, `r[i][j][k]` è la $k-ima$ coordinata di uno degli m vicini di `points[i]`.

Per ogni algoritmo è stato creato il relativo programma e ogni programma può essere utilizzato da riga di comando con la seguente sintassi:
`<nomeProgramma> <inputFile> <outputMode>`
dove `<outputMode>` può avere un valore tra `VERBOSE, SILENT, TIME`, mentre `<inputFile>` è il nome

di un file di testo contenente `nValue` punti a `dValue` dimensioni con la seguente sintassi:

`n=<nValue> d=<dValue> m=<mValue>`

$p1_1$ $p1_2$ ... $p1_d$

$p2_1$ $p2_2$ ... $p2_d$

...

pn_1 pn_2 ... pn_d

Se il programma viene eseguito con l'opzione `VERBOSE`, vengono stampati in output tutti i vicini per ogni punto nel formato:

[p1] -> [$v1_1$] [$v1_2$] ... [$v1_m$]

[p2] -> [$v2_1$] [$v2_2$] ... [$v2_m$]

...

[pn] -> [vn_1] [vn_2] ... [vn_m]

Diversamente se viene specificata l'opzione `SILENT` verrà stampata solo una stringa quando l'algoritmo ha terminato, mentre se viene scelta l'opzione `TIME` verrà restituito in output il tempo di esecuzione (in nanosecondi) dell'algoritmo, garantendo un errore relativo massimo nella misura dell' 1‰(si rimanda alla sezione 4.1.4 per una discussione più approfondita).

4.1.3 Ambiente di test

Nell'implementazione degli algoritmi in questione è fondamentale avere a portata un ambiente di test che permetta di rilevare degli errori di programmazione. Assumendo di implementare correttamente un algoritmo per il vicinato che chiameremo A, per testare un nuovo algoritmo A' è possibile implementare il *Back to Back testing* [25] ossia controllare che $A(x) = A'(x)$ per opportuni valori di x. Nel nostro caso è conveniente dapprima implementare l'algoritmo Naive (e da qui in avanti ne si assumerà un'implementazione corretta) e in seguito implementare gli altri applicando la procedura di test descritta.

Nel caso specifico, assumendo che i programmi implementati abbiano un'interfaccia da riga di comando come descritta in sezione 4.1.2, si può facilmente implementare una procedura di *back to back testing* con i seguenti comandi *bash*:

```
./naive_algo input_file VERBOSE | egrep '\[(.)*\[' | sort > tmp.txt
perl -lape '$_ = qq/@{[sort @F]}/' tmp.txt > naive_sol.txt; rm tmp.txt

./new_algo input_file VERBOSE | egrep '\[(.)*\[' | sort > tmp.txt
perl -lape '$_ = qq/@{[sort @F]}/' tmp.txt > new_algo_sol.txt; rm tmp.txt

cmp naive_sol.txt new_algo_sol.txt
```

Si esegue il programma Naive con un file di input in modalità `VERBOSE`, si selezionano le righe che contengono delle parentesi quadre (quindi le righe che contengono la soluzione) e le si ordinano in modo da rendere la procedura di test indipendente dall'ordine in cui il programma stampa l'output. Ordinate

le righe, si ordinano i punti di ciascuna riga e così facendo la procedura di test è indipendente dall'ordine con cui vengono stampati i vicini di ogni punto. Si salva il risultato ottenuto nel file `naive_sol.txt`. Si esegue con lo stesso input il programma da testare e si esegue lo stesso procedimento di rielaborazione dell'output salvando il risultato nel file `new_algo_sol.txt`. Si comparano i due file e se non vengono trovate differenze il test è da considerarsi passato mentre se ne vengono trovate il test è fallito.

4.1.4 Misura dei tempi

Con argomento `TIME` i programmi implementati stampano il tempo di esecuzione in nanosecondi. Per la misura dei tempi si può utilizzare la funzione C `clock_gettime` ma in questo modo la misura è dipendente dalla risoluzione del clock ovvero il minimo intervallo di tempo misurabile.

Sia x un valore e $\widehat{x}$ un'approssimazione di x, si ricorda la formula dell'errore relativo:

$$\varepsilon = \frac{|\widehat{x} - x|}{x} \tag{4.1}$$

Nel caso specifico, sia x il tempo effettivo di un algoritmo e sia $\widehat{x}$ il tempo misurato. Fissato un errore relativo massimo ε_{max}, la procedura corretta di misurazione prevede di eseguire l'algoritmo per k volte, misurare il tempo totale di esecuzione e dividerlo per k per trovare il tempo di una singola esecuzione ma il cui errore relativo massimo sia limitato da ε_{max}.

Si ottiene:

$$\varepsilon \leq \frac{R}{k\widehat{x} - R} \leq \varepsilon_{max} \tag{4.2}$$

dove R è la risoluzione del clock e $k\widehat{x}$ è il tempo misurato di k esecuzioni dell'algoritmo. L'obbiettivo è quindi misurare il tempo per k esecuzioni tale che k soddisfi:

$$k\widehat{x} \geq \frac{R}{\varepsilon_{max}} + R = t_{min} \tag{4.3}$$

ossia si continua a ripetere l'esecuzione dell'algoritmo finché il tempo misurato non è maggiore di t_{min}.

4.2 Risultati sperimentali

In questa sezione verranno presentati e discussi i risultati sperimentali ottenuti dall'esecuzione dei programmi descritti nella sezione 4.1, con obbiettivo quello di verificare i risultati teorici dei capitoli 1, 2 e 3.

4.2.1 Ambiente di misura

Verrà ora descritta la procedura utilizzata per effettuare tutti i test e le misurazioni che saranno discusse nelle sezioni successive.

I parametri che si possono far variare nell'esecuzione dei programmi, oltre al file di input sono n, m e d. Sia x uno di questi parametri e si supponga di voler misurare il tempo di esecuzione del programma P al variare di x tra x_{min} e x_{max} con k campioni.

Si vogliono ottenere quindi k misure sui punti $x_{min} = x_1, x_2, \ldots, x_k = x_{max}$.

Innanzitutto per ogni x_i vengono creati q file di input $f_{i,1}, \ldots, f_{i,q}$ dove ogni file rispetta il vincolo $x = x_i$.

Per misurare il tempo medio di esecuzione del programma P con un input $x = x_i$ si eseguono:

`./P` $f_{i,1}$ `TIME`

`./P` $f_{i,2}$ `TIME`

...

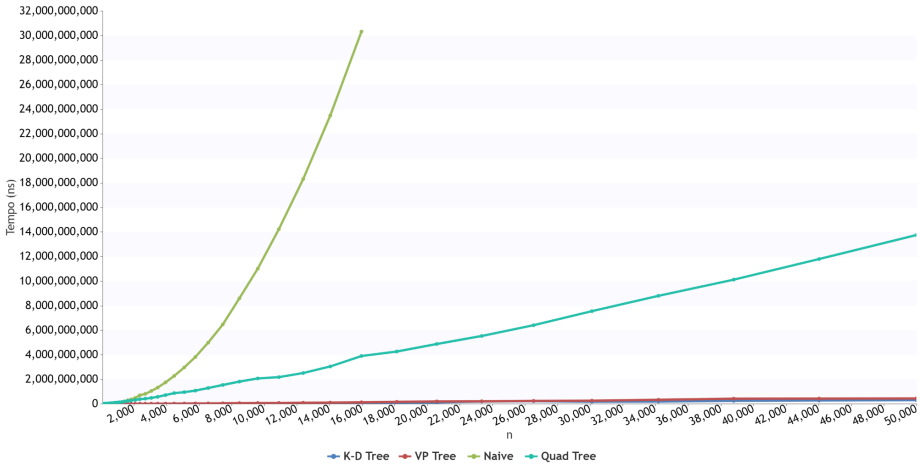
`./P` $f_{i,q}$ `TIME`

e si computa la media dei tempi ottenuti. In tutti i test effettuati nelle sezioni successive è stato posto $q = 20$ ossia per ogni punto x_i si è presa la media di 20 tempi di esecuzione su input file diversi (ma con gli stessi parametri n, m e d).

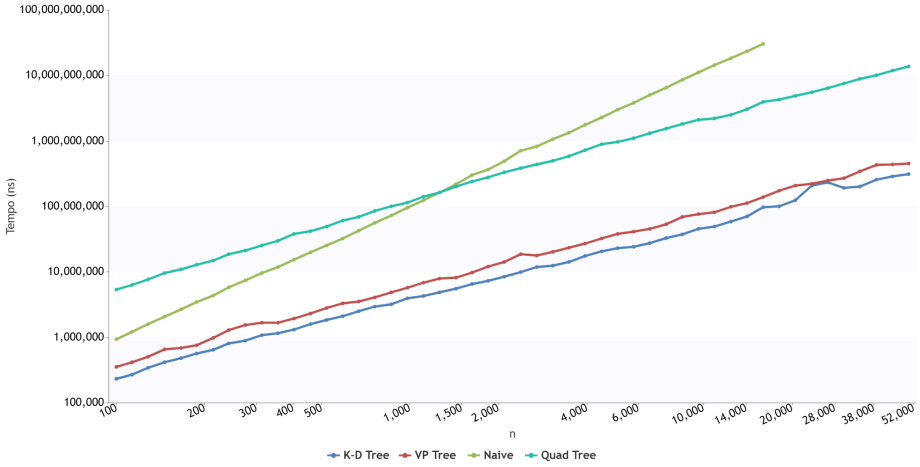
Per quanto riguarda invece la scelta dei k campioni, non sono stati presi da una distribuzione lineare di punti (ossia con distanza costante tra coppie adiacenti di x_i) ma sono stati presi da una distribuzione esponenziale in cui le coppie di x_i adiacenti hanno distanza costante nei grafici in scala doppiamente logaritmica.

4.2.2 Risultati al variare di n

In figura 4.1 è rappresentato il grafico delle performance ottenuto facendo variare il numero di punti n tra 100 e 50000 e ponendo $m = 5$ e $d = 2$. Sono stati eseguiti gli algoritmi Naive, Quad Tree, K-D Tree e VP Tree, dove per Naive si intende l'algoritmo 2 ossia quello che effettua una ricerca esaustiva calcolando n^2 distanze. Partendo dai K-D Tree e VP Tree, si nota che hanno un comportamento quasi equivalente a meno di un fattore costante in questo caso a favore dei K-D Tree. Il tempo atteso $O(n \log(n))$ è quindi confermato. L'algoritmo Naive è chiaramente il peggiore e ha un comportamento quadratico che è in particolare osservabile nel grafico in scala doppiamente logaritmica (figura 4.1b) in cui la retta ha una maggiore inclinazione rispetto alle rette dei Quad Tree e K-D Tree e si osserva che al raddoppio di n, il tempo di esecuzione quadruplica. Per quanto riguarda i Quad Tree (che implementano la generalizzazione ad m vicini proposta in sezione 2.1.6) si può notare sempre un comportamento assimilabile a $O(n \log(n))$ infatti nella scala doppiamente logaritmica in figura 4.1b l'inclinazione della retta è la stessa rispetto a quella dei K-D Tree e VP Tree. Tuttavia la costante moltiplicativa è nel



(a) Scala lineare.



(b) Scala doppiamente logaritmica.

Figura 4.1: Tempo di esecuzione degli algoritmi Naive, Quad Tree, K-D Tree, VP Tree, al variare di n tra 100 e 50000. $m = 5$, $d = 2$.

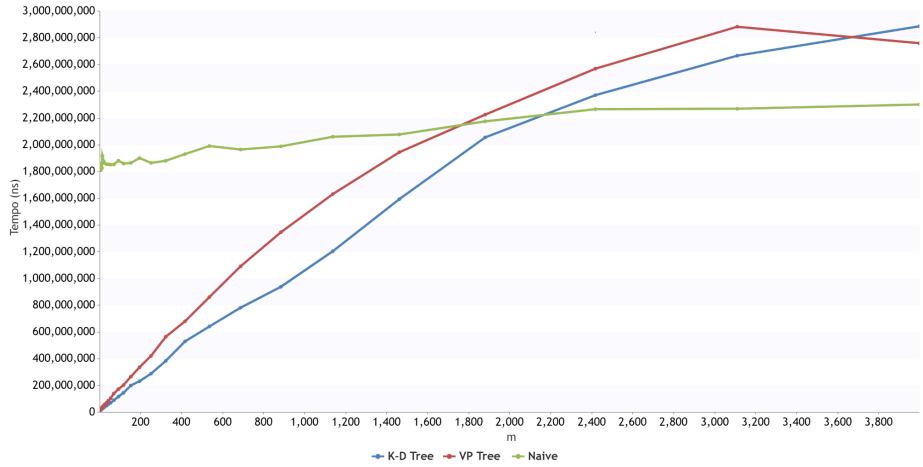
caso dei Quad Tree molto più alta e per n inferiore a 1000 l'algoritmo Naive si dimostra più veloce nella pratica.

4.2.3 Risultati al variare di m

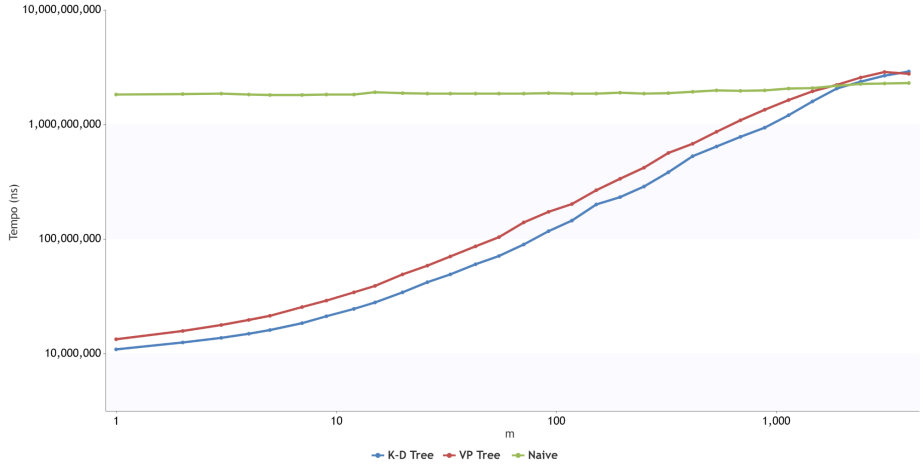
In figura 4.2 è rappresentato il grafico delle performance ottenute facendo variare il numero di punti da includere nel vicinato di ogni punto m, tra 1 e 3999 e ponendo $n = 4000$, $d = 2$. Dai grafici emerge sostanzialmente il comportamento predetto dalle formule 2.6 e 2.8 in sezione 2.2.4 per quanto riguarda i K-D Tree e dalla formula 3.4 in sezione 3.2.4 per i VP Tree. Aumentando m aumenta la necessità di esplorare più punti dello spazio e quindi si arriva a raggiungere un tempo di esecuzione quadratico.

Passando all'algoritmo Naive è interessante notare che all'aumentare di m non peggiora le sue performance, infatti anche se il ciclo `while` del listato 2 prende tempo lineare in m, lo stesso sarà sempre minore asintoticamente alla chiamata alla procedura `SELECT` che prende tempo lineare in n e, ricordando che $m \in O(n)$, si ottiene che l'aumento del tempo di esecuzione del ciclo `while` non ha effetti sul tempo asintotico di esecuzione della soluzione Naive.

In figura 4.3 è invece rappresentato il grafico delle performance ottenute eseguendo l'algoritmo che utilizza i Quad Tree facendo variare m tra 1 e 698 e ponendo $n = 700$ e $d = 2$. Il comportamento



(a) Scala lineare.

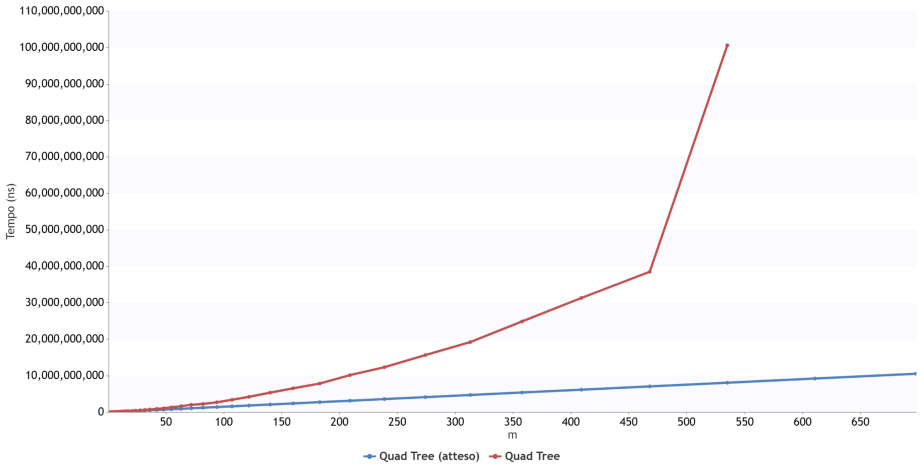


(b) Scala doppiamente logaritmica.

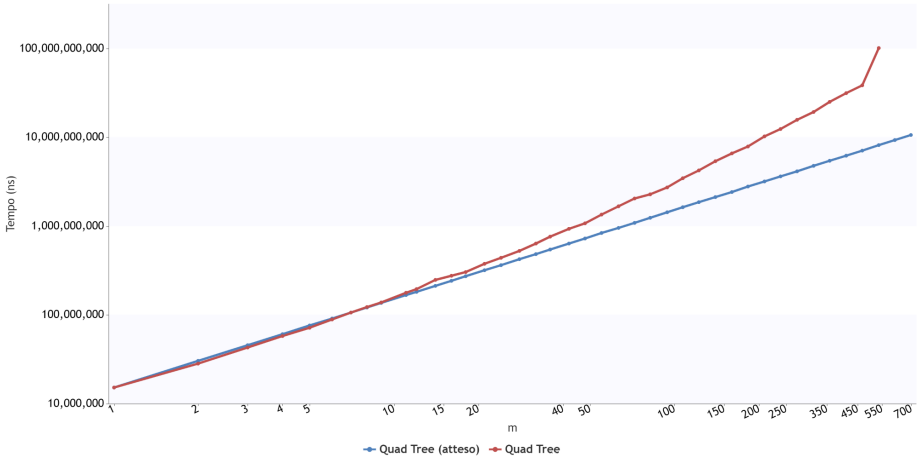
Figura 4.2: Tempo di esecuzione degli algoritmi Naive, K-D Tree, VP Tree, al variare di m tra 1 e 3999. $n = 4000$, $d = 2$.

reale (linea rossa `Quad Tree`) è confrontato con quello atteso (linea blue `Quad Tree (atteso)`) dove, richiamando le analisi fatte in sezione 2.1.6, per comportamento atteso si intende $O(mn\log(n))$. Tale comportamento per m sufficientemente piccoli è verificato dai risultati in figura 4.3. Per $m = 20$ si nota che il comportamento reale inizia a distaccarsi da quello atteso e per $m = 550$ si vede un drastico cambio di pendenza nel grafico in scala doppiamente logaritmica in figura 4.3b. Viene dunque confermato che per valori di m costanti rispetto ad n il tempo di esecuzione è $O(mn\log(n))$ (ossia fissato n, al raddoppio di m, raddoppia il tempo di esecuzione), ma per valori di m che si avvicinano sempre più ad n, i nodi disattivati fanno in modo che in ogni chiamata alla procedura `QUERY` si visitino $O(n)$ nodi e il tempo di esecuzione complessivo dell'algoritmo che utilizza i Quad Tree diventa cubico in n.

Con queste considerazioni ora si può anche capire il perché di una costante così grande per i Quad Tree nelle figure 4.1a e 4.1b: si noti che se il test effettuato fosse stato con $m = 1$ anziché $m = 5$, tutti i dati per i Quad Tree sarebbero uguali ma divisi per un fattore 5, mentre nel caso dei VP Tree e K-D Tree il passaggio da $m = 5$ a $m = 1$ fa rimanere sostanzialmente invariati i dati poiché c'è un'alta probabilità che fissato p i 5 vicini per p si trovino nello stesso bucket oppure in un bucket raggiungibile con poco *back-tracking*.



(a) Scala lineare.



(b) Scala doppiamente logaritmica.

Figura 4.3: Tempo di esecuzione dell'algoritmo che utilizza i Quad Tree al variare di m tra 1 e 698. $n = 700$, $d = 2$.

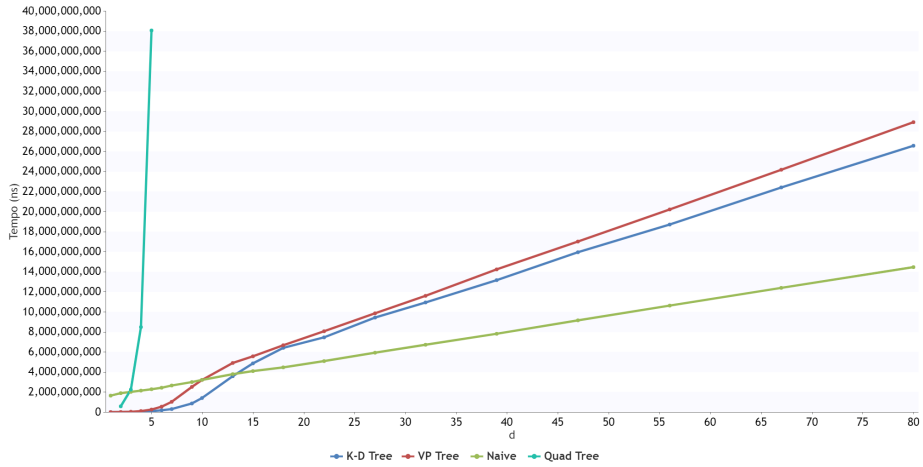
4.2.4 Risultati al variare di d

In figura 4.4 si possono vedere i risultati dei vari algoritmi al variare del numero di dimensioni d tra 1 e 100. In questo caso sono stati posti $m = 5$ e $n = 4000$. Questo test risulta di particolare importanza perché evidenzia il problema *curse of dimensionality* che affligge ogni algoritmo che risolve il problema esatto del vicinato conosciuto ad oggi, in particolare ogni algoritmo implementato per questa tesi.

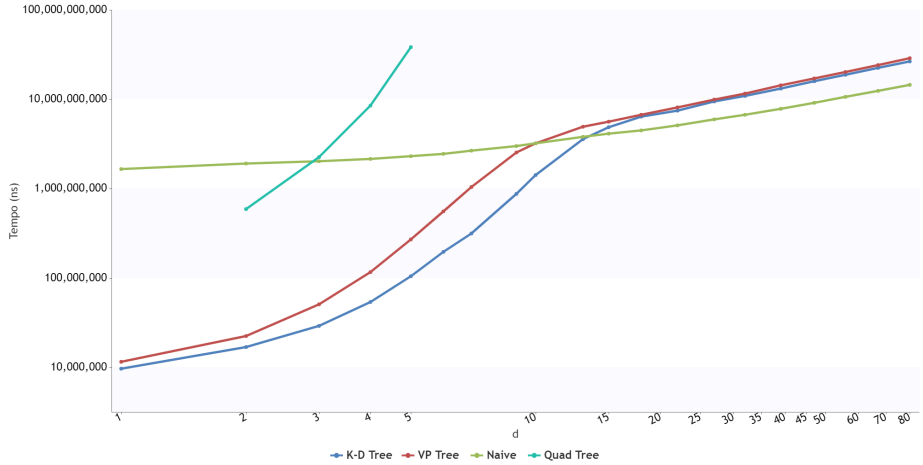
Iniziando dai Quad Tree si può subito notare che ad ogni unità di d aggiunta, il tempo di esecuzione raddoppia e si osserva quindi un tempo esponenziale rispetto a d, confermando i risultati ottenuti in sezione 2.1.8 del capitolo 2.

Passando all'algoritmo Naive, non si ottiene nessuna sorpresa, il tempo di esecuzione aumenta linearmente con d.

Infine, per quanto riguarda i K-D Tree e i VP Tree, come nel caso dell'aumento di m considerato in sezione 4.2.3, durante una ricerca nell'albero si ha la necessità di esplorare una porzione sempre maggiore di spazio fino a $d = 9$ in cui la ricerca degenera in una ricerca esaustiva e quindi in entrambi gli algoritmi in ogni nodo verranno eseguite entrambe le chiamate ricorsive, sia quella di destra e sia quella di sinistra. Per $d > 9$ le strutture non riescono ad indicizzare i punti in maniera ottimale e gli algoritmi procedono prendendo tempo $O(n^2 d log(m))$ che diventa $O(n^2 d)$ (ossia lo stesso tempo dell'algoritmo Naive) se m



(a) Scala lineare.



(b) Scala doppiamente logaritmica.

Figura 4.4: Tempo di esecuzione degli algoritmi Naive, Quad Tree, K-D Tree, VP Tree, al variare di d tra 1 e 80. $m = 5$, $n = 4000$.

viene considerata costante.

Si evidenzia quindi *curse of dimensionality* sia per i VP Tree e sia per i K-D Tree che anche in questo caso sono del tutto equivalenti tra di loro a meno di un fattore costante a favore dei K-D Tree.

4.2.5 Risultati per spazi con condizioni periodiche

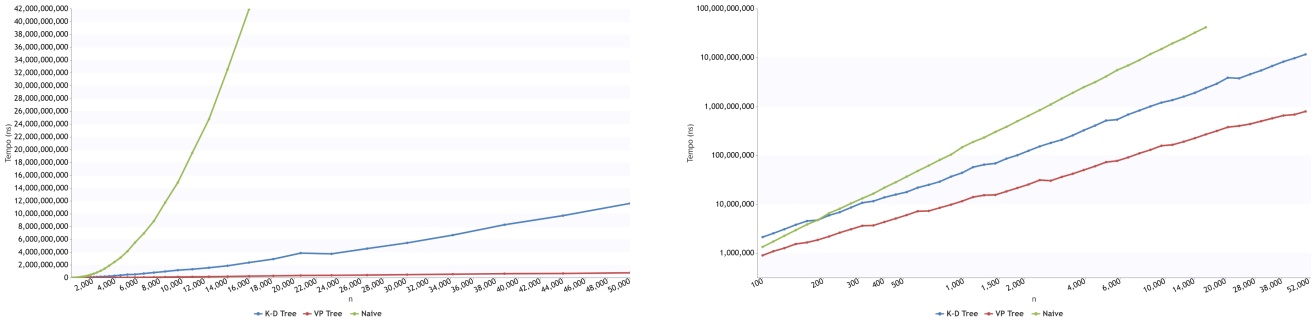
Nelle figure 4.5, 4.6 e 4.7 sono rappresentati i risultati effettuati effettuando gli stessi identici test effettuati per ottenere i risultati delle rispettive figure 4.1, 4.3 e 4.4 con l'unica differenza che gli algoritmi eseguiti in questo caso sono stati: Naive, VP Tree, K-D Tree tutti e 3 nelle varianti per spazi con condizioni periodiche. Per gli algoritmi Naive e VP Tree l'unica diversità è la metrica utilizzata nella distanza mentre per quanto riguarda i K-D Tree (che nella variante base funzionano solo in spazi euclidei) si è utilizzata la modifica discussa nella sezione 3.1 del capitolo 3 che permette di simulare uno spazio periodico aumentando opportunamente il numero di query necessarie.

Si osservi fin da ora che per quanto riguarda l'algoritmo Naive e l'algoritmo che utilizza i VP Tree, nelle figure 4.5, 4.6 e 4.7 non vi è alcuna discordanza dei comportamenti asintotici rispetto a quanto già analizzato nelle figure 4.1, 4.2 e 4.4. I VP Tree e l'algoritmo Naive hanno quindi un comportamento asintotico indipendente dalla metrica utilizzata. Si evidenzia ancora una volta che questo rappresenta

un risultato notevole per i VP Tree che pur basando tutto solo sulla dicotomia *vicino/lontano* riescono ad essere più generali dei K-D Tree che sfruttano più informazioni.

Rimane ora da considerare solo il caso dei K-D Tree modificati. Per quanto riguarda il test effettuato al variare di n (si veda la figura 4.5) e il test effettuato al variare di m (figura 4.6) il comportamento asintotico dei K-D Tree rimane sostanzialmente quello analizzato nelle rispettive sezioni 4.2.2 e 4.2.3 ma si noti che ora la costante moltiplicativa è a favore dei VP Tree che risultano più veloci nella pratica.

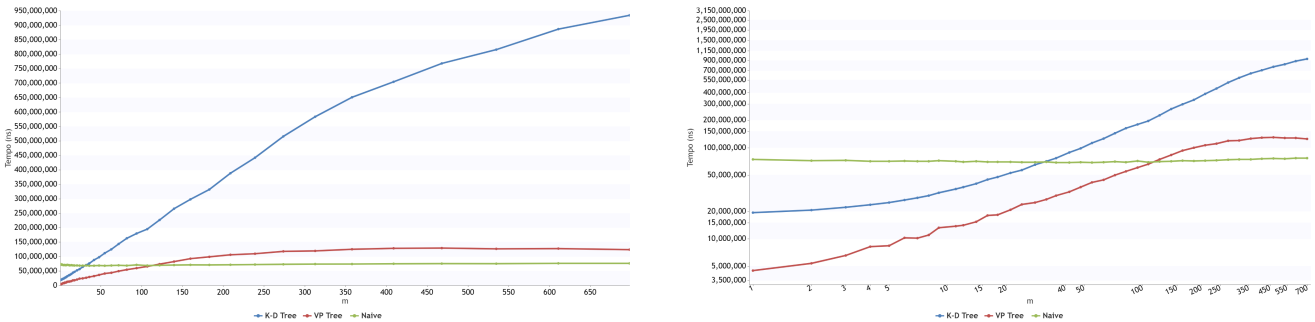
Il vero limite dei K-D Tree modificati si può osservare facendo variare d (figura 4.7). Si può subito notare il tempo esponenziale rispetto al numero di dimensioni d, questo è dovuto alla simulazione dello spazio periodico che richiede $O(3^d)$ ricerche nell'albero per ognuno degli n punti. I risultati della sezione 3.1.2 vengono quindi confermati.



(a) Scala lineare.

(b) Scala doppiamente logaritmica.

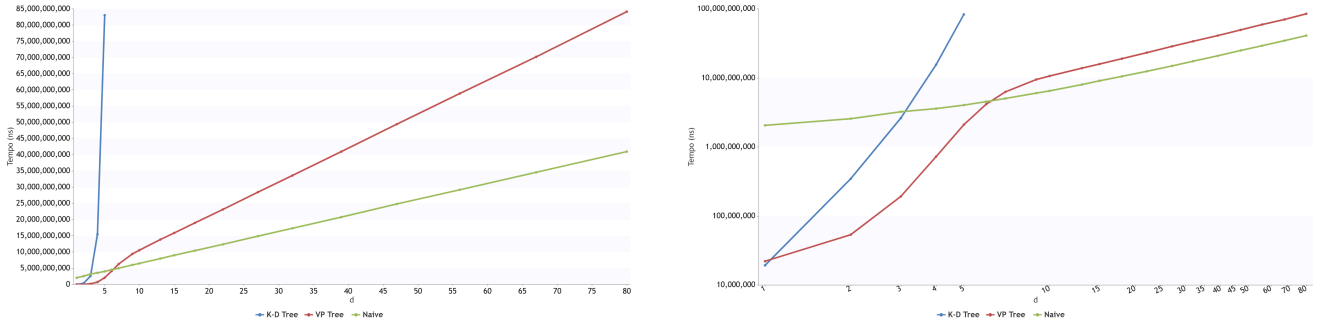
Figura 4.5: Tempo di esecuzione degli algoritmi Naive, K-D Tree, VP Tree, al variare di n tra 100 e 50000. $m = 5$, $d = 2$.



(a) Scala lineare.

(b) Scala doppiamente logaritmica.

Figura 4.6: Tempo di esecuzione degli algoritmi Naive, K-D Tree, VP Tree, al variare di m tra 1 e 698. $n = 700$, $d = 2$.



(a) Scala lineare.

(b) Scala doppiamente logaritmica.

Figura 4.7: Tempo di esecuzione degli algoritmi Naive, K-D Tree, VP Tree, al variare di d tra 1 e 80. $m = 5$, $n = 4000$.

5

Conclusioni e sviluppi futuri

Nel presente lavoro sono state presentate dettagliatamente alcune soluzioni al problema della scelta del vicinato insieme ai risultati principali che esso porta con sè.

Sono stati implementati e testati tutti gli algoritmi proposti nei capitoli 1, 2 e 3 confermando tutti i risultati teorici ottenuti. In particolare per quanto riguarda la generalizzazione a m vicini dei Quad Tree è stata verificata l'analisi sui tempi di esecuzione fatta in sezione 2.1.6 nel capitolo 2. Anche i Quad Tree sono quindi utilizzabili per risolvere il problema del vicinato e, per opportuni valori, molto più velocemente dell'algoritmo Naive (ma meno velocemente degli algoritmi che utilizzano VP Tree e K-D Tree).

Degli algoritmi analizzati si conclude che:

- Per spazi euclidei i K-D Tree si sono mostrati i più veloci.
- In spazi non euclidei come gli spazi con condizioni di periodicità, i K-D Tree non funzionano più (e vanno riadattati) e i VP Tree si comportano meglio degli altri algoritmi.
- Con valori di d particolarmente elevati oppure se $m \in \Theta(n)$, i K-D Tree e i VP Tree degenerano in una ricerca esaustiva e conviene utilizzare l'algoritmo Naive.

Le soluzioni viste soffrono del noto problema *curse-of-dimensionality* (che si riesce ad evitare nel caso dell'approssimazione del vicinato [26]). In letteratura sono stati fatti degli sforzi per evidenziare la presenza di questo problema (si veda ad esempio [26]) tuttavia manca ancora una dimostrazione formale che ne testimoni l'esistenza. Le strutture analizzate in questo documento non sono le uniche esistenti: esistono decine di strutture dati ad albero ciascuna con le proprie varianti e i propri metodi di costruzione. Anche con queste strutture si raggiungono risultati simili a quelli analizzati in questa tesi pertanto i K-D Tree insieme ai VP Tree rimangono una delle più vecchie e classiche opzioni per implementare efficientemente un algoritmo per la scelta del vicinato. Seppur la letteratura è così vasta non esiste ad oggi una soluzione ottimale in tutti i contesti e non esiste un confronto dettagliato di tutte queste strutture. Rimane quindi aperto il problema di trovare una soluzione universale che prenda tempo $\Theta(n \log(n) + ndm)$ e spazio polinomiale in n, indipendentemente dal tipo di spazio, dalla dimensionalità dei punti d, dalla numerosità del vicinato m e dalla distribuzione degli n punti, oppure di dimostrare formalmente che una tale soluzione non può esistere e quindi il limite inferiore mostrato in sezione 1.1.2

necessita di essere ulteriormente raffinato. Ogni soluzione proposta in letteratura offre delle intuizioni e degli spunti interessanti, ad esempio Vaidya [2] mostra un modo assolutamente non banale per limitare superiormente il tempo totale di splitting dei nodi in un albero, i K-D Tree offrono una via intelligente per dividere lo spazio e mantenere il bilanciamento dell'albero, i VP Tree trovano il modo di dedurre le informazioni esclusivamente dalla dicotomia vicino/lontano dipendente solo dalla metrica utilizzata, altre strutture invece si comportano meglio in più dimensioni ma peggio nel caso medio e altre ancora risolvono il problema dinamicamente ovvero prendono in considerazione la rimozione e l'aggiunta di punti dall'insieme di input V (aspetto non discusso nel presente documento). Per lavori futuri rimane quindi da indagare per ogni struttura quale sia la variante migliore e quale metodo di ricerca porti al miglior tempo di esecuzione, ad esempio nel caso dei VP Tree la selezione del punto di vantaggio ottimale è ancora oggetto di discussione e ricerca. Rimane infine aperta la ricerca di una nuova struttura dati che possa prendere in considerazione tutte le ottime intuizioni formatesi negli anni che speranziosamente possa oltrepassare *curse-of-dimensionality*.

Bibliografia

[1] Kasturi Varadarajan. All nearest neighbours via quadtrees. Technical report, The University of Iowa, 2013.

[2] Pravin M. Vaidya. An o(n log n) algorithm for the all-nearest.neighbors problem. *Discret. Comput. Geom.*, 4:101–115, 1989.

[3] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, sep 1975.

[4] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, sep 1977.

[5] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998.

[6] Raphael A. Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 2004.

[7] Martin Skrodzki. *Neighborhood Data Structures, Manifold Properties, and Processing of Point Set Surfaces*. PhD thesis, 2019.

[8] Peter Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. volume 93, 01 1993.

[9] Jeffrey K. Uhlmann. Metric trees. *Applied Mathematics Letters*, 4(5):61–62, 1991.

[10] Ada Fu, Polly Chan, Yin-ling Cheung, and Yiu Moon. Dynamic vp-tree indexing for n-nearest neighbor search given pairwise distances. *The Vldb Journal - VLDB*, 9:154–173, 07 2000.

[11] Tzi cker Chiueh. Content-based image indexing. In *In Proceedings of the 20th VLDB Conference*, pages 582–593, 1994.

[12] Tolga Bozkaya and Meral Ozsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Trans. Database Syst.*, 24(3):361–404, sep 1999.

[13] Stephen M. Omohundro. Five balltree construction algorithms. ICSI Technical Report TR-89-063 Berkeley, 1989.

[14] Yury A. Malkov and Dmitry A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *CoRR*, abs/1603.09320, 2016.

[15] Kenneth L. Clarkson. Fast algorithms for the all nearest neighbors problem. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 226–232, 1983.

[16] Michael Ben-Or. Lower bounds for algebraic computation trees. *Proc. 15th Annual Symp. on Theory of Computing*, 5:80–86, 01 1983.

[17] Christos H Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, 1994.

[18] Anna Lubiw and András Rácz. A lower bound for the integer element distinctness problem. *Information and Computation*, 94(1):83–92, 1991.

[19] Joshua Brown, Terry Bossomaier, and Lionel Barnett. Review of data structures for computationally efficient nearest-neighbour entropy estimators for large systems with periodic boundary conditions. Journal of Computational Science Volume 23, November 2017, Pages 109-117.

[20] M. Kuhn and K. Johnson. *Applied Predictive Modeling*. SpringerLink : Bücher. Springer New York, 2013.

[21] Kneighborsclassifier: Machine learning in python. `https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html`, [Online; accessed 08-June-2022].

[22] Kneighborsregressor: Machine learning in python. `https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html`, [Online; accessed 08-June-2022].

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[24] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is np-complete. *Information Processing Letters*, 5(1):15–17, 1976.

[25] Ian Sommerville. *Software Engineering*. Addison-Wesley, Harlow, England, 9 edition, 2010.

[26] Allan Borodin, Rafail Ostrovsky, and Yuval Rabani. *Lower Bounds for High Dimensional Nearest Neighbor Search and Related Problems*, pages 253–274. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.