



UNIVERSITÀ DEGLI STUDI DI PARMA  
Facoltà di Scienze Matematiche, Fisiche e Naturali  
Corso di Laurea in Informatica

**Studio di strutture dati per la  
rappresentazione di mappe di densità  
di proteine**

**Relatore:**  
Dott. Alessandro Dal Palù

**Candidata:**  
Elena Ghinelli

Anno Accademico 2006-2007

*A mia madre Mariella  
e a mio padre Ettore*

# Ringraziamenti

Mi sembra doveroso arrivata a questo punto della mia carriera scolastica ringraziare le persone che mi hanno permesso di raggiungere questo traguardo. Per primi voglio ringraziare i miei genitori che mi hanno permesso di arrivare a questo traguardo con la calma e la tranquillità necessaria, senza farmi mai pressioni e sostenendomi nei momenti più bui.

Il ringraziamento successivo va a tutti i compagni di università e soprattutto a tutti quei ragazzi di matematica che mi hanno sempre aiutata e che mi hanno convinto a non mollare quando mi sono demoralizzata. Un grazie sincera anche a tutti gli altri amici che mi hanno accompagnato nel percorso della mia vita.

Per concludere, un ringraziamento sincero al mio relatore Alessandro Dal Palù che mi ha permesso di svolgere un tirocinio che ho trovato estremamente stimolante e che mi ha aiutata a portare a termine questo lavoro, nonostante tutti gli altri impegni quotidiani.

A tutte queste persone un grazie di cuore!!

Elena

# Indice

<b>Prefazione</b>	<b>4</b>
<b>1 Introduzione</b>	<b>6</b>
1.1 Le mappe di densità . . . . .	6
1.2 Gli Octree . . . . .	8
1.3 Ricerca esaustiva . . . . .	9
1.4 Lo scopo della tesi . . . . .	10
<b>2 Studio preliminare</b>	<b>12</b>
2.1 Problema giocattolo . . . . .	12
2.1.1 Costruzione octree . . . . .	13
2.1.2 Un template . . . . .	15
2.2 L'algoritmo di ricerca . . . . .	15
2.3 Prove e risultati . . . . .	18
<b>3 Ricerca con templates multipli</b>	<b>20</b>
3.1 Rappresentazione di templates multipli in una struttura dati . . . . .	20
3.2 Octree con liste di puntatori . . . . .	24
3.2.1 Complessità in spazio . . . . .	24
3.2.2 Valutazione . . . . .	26
3.3 Octree con quantizzazione fissa . . . . .	26
3.3.1 Un'approssimazione delle strutture della mappa . . . . .	26
3.3.2 Complessità in spazio . . . . .	28
3.3.3 Valutazione . . . . .	28
3.4 Ricerca . . . . .	29
3.4.1 La ricerca approssimata . . . . .	31
<b>4 Implementazioni e risultati</b>	<b>32</b>
4.1 La classe template . . . . .	32
4.2 La classe mappa . . . . .	34

4.3	Risultati . . . . .	36
<b>5</b>	<b>Conclusioni</b>	<b>39</b>
5.1	Lavori futuri . . . . .	39
5.2	Utilizzi alternativi . . . . .	40
	<b>Bibliografia</b>	<b>41</b>

# Prefazione

Uno dei maggiori problemi in biochimica è la determinazione della struttura tridimensionale di una proteina.

Il metodo maggiormente usato per determinare la struttura tridimensionale di una proteina è la cristallografia a raggi-X. Nel processo di cristallografia a raggi-X per prima cosa si isola una proteina e se ne crea il cristallo. Una volta ottenuto il cristallo della proteina, si fa passare attraverso di esso un fascio di raggi-X e a seconda della diffrazione di questi raggi si ottiene un modello di punti su un piano. Questi punti rappresentano l'intensità dell'immagine della trasformata di Fourier della proteina. Infine una trasformazione di Fourier converte le intensità in una mappa di densità.

L'ultimo passo della cristallografia a raggi-X è interpretare la mappa di densità convertendola in una rappresentazione che sia utilizzabile dai biologi. È proprio l'interpretazione della mappa di densità di una proteina il punto di maggior debolezza, soprattutto nel caso in cui la mappa che si ottiene abbia una risoluzione bassa, infatti la sua interpretazione può richiedere settimane o addirittura mesi.

L'interpretazione della mappa di densità è il collo di bottiglia di questo problema, è proprio a questo punto che l'informatica cerca di costruire metodi sempre più efficienti per l'analisi delle mappe di densità.

Una proteina è formata da una catena di aminoacidi, dove gli aminoacidi sono particolari molecole aventi una conformazione simile.

Una volta finita anche l'analisi della mappa di densità si ottiene un modello tridimensionale per la struttura della proteina, trovando le coordinate cartesiane di ogni aminoacido appartenente alla proteina.

# Capitolo 1

## Introduzione

### 1.1 Le mappe di densità

**Definizione 1.1 (mappa di densità di una proteina)** *Una mappa di densità è un'immagine tridimensionale di una proteina.*

Una mappa di densità è il risultato intermedio del processo di cristallizzazione a raggi-X e rappresenta la densità elettrica delle molecole in una data porzione di spazio.

Solitamente le mappe di densità hanno una risoluzione che varia tra 1Å e 12Å. In mappe di densità con un'alta risoluzione (intorno ai 2Å) sono visibili gli atomi individualmente, purtroppo creare mappe di densità con questa risoluzione è molto costoso. Se si riuscisse ad ottenere mappa di densità con un'alta risoluzione il processo di determinazione delle posizioni degli amminoacidi della proteina nello spazio risulterebbe molto più veloce.

Una mappa di densità è semplicemente un tasso uniforme  $R$  nello spazio tridimensionale. Questo genera una partizione dello spazio in cubi con lato di lunghezza  $R$ . Ogni cubo contiene una densità che è descritta da un numero reale.

Di conseguenza ogni punto della mappa ha la sua densità.

**Definizione 1.2 ( $M[i][j][k]$ )**  $M[i][j][k]$  è la densità associata alla mappa nel cubo di coordinate  $i, j, k$ , nello spazio, cioè:

$$M[i][j][k] = P\langle i, j, k \rangle$$

Dove  $i, j, k \in \mathbb{N}$ .

Per come sono stati definiti i punti interni di una mappa si ha che una mappa di densità può essere vista come una versione discreta dello spazio tridimensionale.

**Definizione 1.3** ( $dim_x$ ) *Si definisce  $dim_x$  la dimensione del lato della mappa di densità sull'asse  $x$ .*

**Definizione 1.4** ( $dim_y$ ) *Si definisce  $dim_y$  la dimensione del lato della mappa di densità sull'asse  $y$ .*

**Definizione 1.5** ( $dim_z$ ) *Si definisce  $dim_z$  la dimensione del lato della mappa di densità sull'asse  $z$ .*

**Definizione 1.6 (dimensione)** *Si definisce dimensione di una mappa di densità di un template il prodotto fra  $dim_x$ ,  $dim_y$  e  $dim_z$ , cioè:*

$$dimensione = dim_x \cdot dim_y \cdot dim_z$$

La dimensione corrisponde al numero dei punti presenti nella mappa di densità.

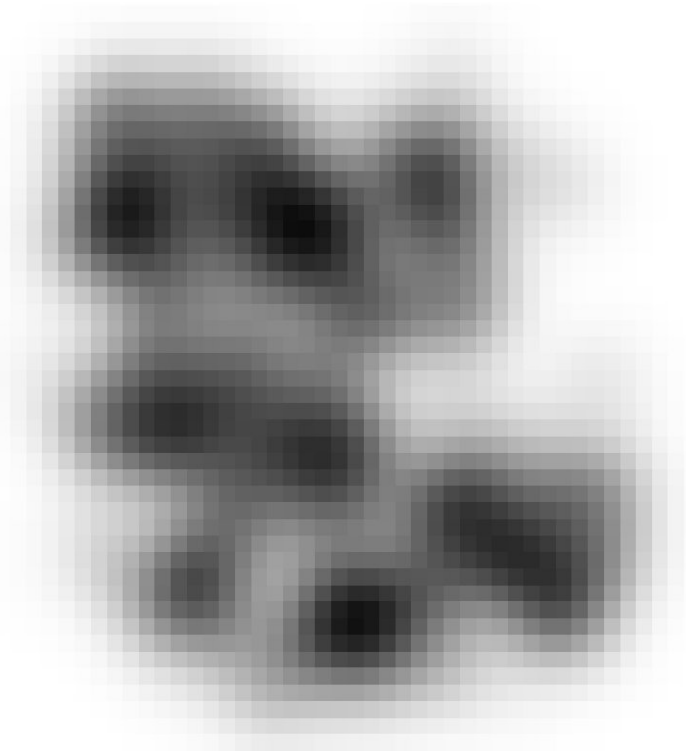


Figura 1.1: Immagine di un layer di una mappa di densità, le zone più scure sono quelle aventi maggiore densità, mentre quelle più chiare sono quelle a densità minore.

## 1.2 Gli Octree

**Definizione 1.7 (Octree)** *Un octree è una struttura dati ad albero dove ogni nodo interno ha otto figli.*

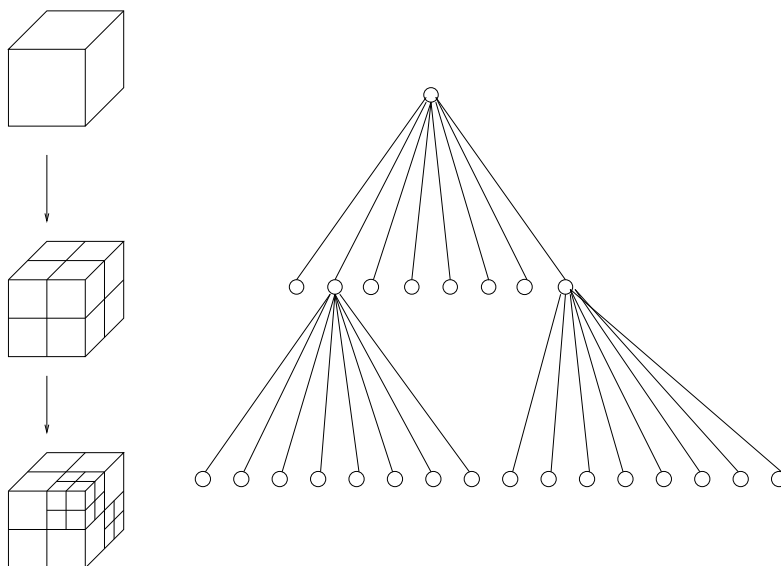


Figura 1.2: Octree e divisione dello spazio tridimensionale

Gli octree sono spesso utilizzati per rappresentare la suddivisione dello spazio tridimensionale in otto cubi più piccoli, come si può vedere dalla figura 1.2. Dalla figura si vede come ogni nodo dell'octree rappresenta un cubo in cui è stato diviso lo spazio iniziale. La divisione dello spazio viene ripetuta ricorsivamente fino a quando lo si ritiene necessario.

In questa tesi i cubi in cui viene diviso lo spazio sono enumerati nel seguente modo:

**Primo nodo:** il cubo in basso, dietro a sinistra.

**Secondo nodo:** il cubo in basso, dietro a destra.

**Terzo nodo:** il cubo in basso, davanti a sinistra.

**Quarto nodo:** il cubo in basso, davanti a destra.

**Quinto nodo:** il cubo in alto, dietro a sinistra.

**Sesto nodo:** il cubo in alto, dietro a destra.

**Settimo nodo:** il cubo in alto, davanti a sinistra.

**Ottavo nodo:** il cubo in alto, davanti a destra.

I livelli dell'albero si contano partendo dalla radice (livello 0). Avendo questa enumerazione per i livelli si ha che, se si considera un albero completo, ad ogni livello il numero dei nodi è:

$$num\_nodi(livello) = 8^{livello}$$

Solitamente i valori contenuti nei nodi rappresentano una proprietà o quantità associata al cubo corrispondente. Nel corso di questa tesi all'interno dei nodi si trova la densità dei cubi. Gli octree hanno svariati utilizzi come per esempio:

- La quantizzazione dei colori.
- La collision detection.
- La modellazione geometrica.

### 1.3 Ricerca esaustiva

Nella ricerca quello che si vuole fare è cercare in quali posizioni della mappa è possibile collocare un template, dove un template è un una catena amminoacidica, formata da alcuni amminoacidi consecutivi.

Ricerca le posizioni in cui è possibile collocare un template consiste nel confrontare i valori contenuti all'interno della mappa e quelli contenuti nel template per vedere se tutti i valori di quest'ultimo sono minori o uguali a quelli della mappa. Questa operazione è chiamata **templates matching**.

Solitamente il templates matching all'interno di una mappa di densità di una proteina si effettua eseguendo quello che può essere chiamato test puntuale, cioè si confrontano i valori contenuti all'interno del template con quelli contenuti nella mappa; solo i sotto pezzi della mappa dove tutti i punti del template hanno una densità minore o uguale a quella della mappa vengono accettati come posizioni ammissibili per il template, mentre gli altri vengono scartati. In questa situazione non è possibile confrontare più di un template alla volta con la mappa di densità.

Il test puntuale può ricordare la convoluzione, infatti si può considerare i valori contenuti all'interno della mappa come i valori appartenenti alla funzione

$f(n)$  e i valori contenuti nel template come appartenenti alla funzione  $g(n)$ . Lavorando nel discreto si ha che la convoluzione nel caso di una dimensione è definita come:

$$(g * f)(n) = \sum_m g(m)f(n - m)$$

Nel caso della convoluzione l'analisi di  $f * g$  porta a decidere se le due funzioni sono simili in una particolare posizione  $n$ .

Nel caso del test puntuale non viene effettuata la somma di tutti i contributi per valutare se un template può trovarsi in una determinata posizione, ma vengono esclusivamente confrontati i valori contenuti per capire se la funzione è minore o uguale alla funzione della mappa di densità.

## 1.4 Lo scopo della tesi

Lo scopo di questa tesi è lo studio di strutture dati per la rappresentazione di mappe di densità di proteine per la ricerca di posizioni di templates all'interno di una proteina.

**Definizione 1.8 (Template)** *Un template è la mappa di densità di un insieme di amminoacidi.*

D'ora in avanti con il termine mappa si indicherà la mappa di densità dell'intera proteina.

Il lavoro presentato nella tesi si colloca all'interno del problema globale di determinazione della struttura tridimensionale di una proteina presentato nella prefazione. Si vuole studiare in particolare una struttura dati per rappresentare le mappe e i templates che permetta di accelerare la fase di identificazione nella mappa di densità degli amminoacidi. Si vorrebbe aggiungere un'ulteriore miglioria alla situazione preesistente cercando di creare una struttura dati che permetta di eseguire contemporaneamente la ricerca delle posizioni di tutti i templates disponibili.

Nel corso della tesi si presentano i vari passaggi effettuati per costruire una struttura dati contenente un insieme di templates che andranno ricercati all'interno della mappa.

Il primo passaggio che si è fatto è quello di studiare principalmente una rappresentazione delle mappe di densità all'interno di un problema giocattolo (capitolo 2). In questa prima parte si è voluto lavorare con singoli templates per cercare di capire se era possibile rendere più veloce il templates matching, cercando di inserire i valori delle mappe di densità in strutture dati diverse da quelle usate per il test puntuale.

Il passo successivo è stato quello di ricercare una rappresentazione per i templates, che fosse compatta ma allo stesso tempo permettesse di distinguere i singoli templates (capitolo 3).

Successivamente si presentano le implementazioni della nuova struttura dati, della rappresentazione delle mappe di densità e i risultati che si sono ottenuti (capitolo 4).

Per concludere si presentano i vari utilizzi della struttura dati creata e i lavori futuri (capitolo 5).

# Capitolo 2

## Studio preliminare

### 2.1 Problema giocattolo

Come studio preliminare del problema descritto nel paragrafo 1.4 si è ricercata una prima rappresentazione sia per la mappa di densità di una proteina che per la mappa di densità di un singolo template, cercando di rendere queste rappresentazioni efficaci per la ricerca.

Da una prima analisi dei dati di input ci si è resi conto che eseguire la ricerca delle possibili posizioni del template all'interno della mappa di densità della proteina confrontando i valori dei singoli punti non era l'idea migliore (sezione 1.3). Si presentavano due problemi:

1. I valori presenti all'interno delle due mappe sono affetti da errori.
2. Eseguire la ricerca confrontando direttamente i valori delle due mappe può richiedere un tempo eccessivamente lungo.

Per risolvere il primo dei due problemi si è pensato di usare un'ulteriore approssimazione dei dati.

**Definizione 2.1 (Livelli di quantizzazione)** *I livelli di quantizzazione sono l'insieme dei valori che i dati appartenenti alla mappa di densità possono assumere. Si chiami  $q$  la cardinalità di questo insieme.*

In questa tesi si considera sempre che se  $q$  è il numero dei livelli di quantizzazione allora i livelli apparteranno all'insieme  $\{l \in \mathbb{N} : 0 \leq l \leq q - 1\}$ .

È necessario, a questo punto, far in modo che i valori contenuti nella mappa di densità assumano solo i valori appartenenti ai livelli di quantizzazione. Per fare ciò bisogna prima di tutto dare la seguente definizione.

**Definizione 2.2 (Massimo)** *MAX è il valore più grande contenuto nella mappa di densità, cioè:*

$$MAX = \max\{M[i][j][k] : i < \dim_x, j < \dim_y, k < \dim_z\}$$

Per fare in modo che i valori contenuti all'interno della mappa appartengano all'insieme dei livelli di quantizzazione si eseguono le seguenti operazioni sui singoli valori:

- Tra gli  $M[i][j][k]$  si cerca il massimo.
- Dopo di che si calcola  $M'[i][j][k]$  dove  $M'$  è la mappa di densità con i valori approssimati.

$$M'[i][j][k] = \lfloor M[i][j][k] / MAX \cdot q \rfloor$$

- Per concludere si esegue una piccola accortezza per essere sicuri che i valori appartengano ai livelli di quantizzazione, cioè:

$$M'[i][j][k] = q \rightarrow M'[i][j][k] = q - 1$$

Questo serve per gestire al meglio il caso in cui  $M[i][j][k] = MAX$ , infatti si avrebbe che  $M'[i][j][k] = q$ , ma se si guarda come è stato definito l'insieme è subito evidente che  $q \notin \text{livelli di quantizzazione}$ .

Dopo tutti i passaggi si ottiene che:

$$(\forall M'[i][j][k])(M'[i][j][k] \in \{l \in \mathbb{N} : 0 \leq l \leq q - 1\})$$

È importante sottolineare che tutti i punti della mappa non contengono più i valori reali che avevano all'inizio ma solo un'approssimazione di essi.

A questo punto si vuole creare una rappresentazione diversa della mappa di densità che permetta di non eseguire solo il test puntuale, ma una ricerca leggermente diversa che viene descritta più avanti nel corso di questo capitolo. Per rappresentare le mappe di densità si è scelto di usare gli octree, descritti in 1.2, sfruttando le caratteristiche delle mappe di densità descritte in 1.1.

### 2.1.1 Costruzione octree

Nella sezione 1.2 si sono descritte le caratteristiche generali degli octree, in questa sezione invece si presenta come in questa particolare situazione vengono calcolati i valori contenuti all'interno dell'albero.

Per trovare i valori dei nodi dell'albero si usa un procedimento ricorsivo, che è il seguente:

**Passo base:** come prima cosa vengono assegnati alle foglie i valori della mappa di densità, i valori che però vengono assegnati non sono gli  $M[i][j][k]$ , descritti nella sezione 1.1 ma gli  $M'[i][j][k]$  che sono stati quantizzati. Si vuole costruire un albero completo; affinché un albero sia completo è necessario che il numero delle foglie sia un potenza di 8.

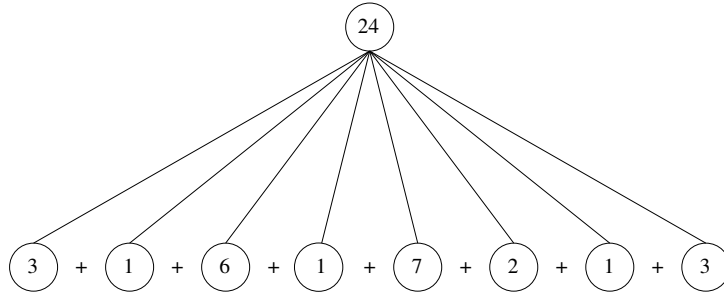


Figura 2.1: Costruzione di un nodo

**Passo ricorsivo:** Per poter assegnare i valori ai nodi del livello  $l$  è necessario che siano stati assegnati i valori ai nodi del livello  $l + 1$ . Si consideri quindi di aver costruito tutti i livelli fino al livello  $l + 1$  e si vuole trovare il valore contenuto all'interno dei nodi del livello  $l$ . Sia  $V(nodo)$  il valore del nodo e  $f_{nodo}^i$  l' $i$ -esimo figlio del nodo allora si ha che:

$$V(nodo) = \sum_{0 \leq i < 8} f_{nodo}^i$$

Quindi ogni nodo interno dell'albero è dato dalla somma dei valori contenuti nei suoi otto figli (figura 2.1).

Si procede in questo modo per tutti i livelli fino ad avere costruito l'albero fino alla radice.

Si noti come all'interno dell'albero ogni nodo contiene la densità all'interno del cubo che esso rappresenta. Questa caratteristica verrà usata durante la ricerca.

Per come è stato costruito l'albero è importante sottolineare che la radice contiene la densità totale della mappa, vedremo durante la ricerca quanto questa caratteristica sia utile.

## 2.1.2 Un template

Un template è una mappa di densità, quindi si può rappresentare un template nello stesso modo in cui si rappresenta una mappa, quindi  $T[i][j][k]$  rappresenta il punto di coordinate  $i, j, k$  della mappa di densità del template.

A ogni punto del template viene assegnato un valore usando la seguente funzione in tre variabili:

$$f(x, y, z) = e^{-((x-4)^2+(y-4)^2+(z-4)^2)}$$

La costruzione del template usando la funzione appena definita è solo un esempio che si è scelto di usare in questa particolare situazione, non una specifica caratteristica di tutti i templates.

La funzione appena definita ha le seguenti caratteristiche:

- Il suo valore massimo è uno.
- Simula una palla Guassiana centrata in 4.

I valori della funzione rappresentano le densità dei punti del template. Su di essi si costruisce un octree come descritto in 2.1.1.

Essendo che un template deve essere posizionato all'interno della mappa avrà una dimensione, definita in 1.1, minore rispetto alla mappa di densità della proteina.

## 2.2 L'algoritmo di ricerca

Durante la ricerca si vuole determinare in quali posizioni all'interno della mappa sia possibile posizionare un template. È necessario prima di tutto dare la definizione di posizione.

**Definizione 2.3 (Posizione di un template)** *La posizione di un template è una tripla  $\langle P_x, P_y, P_z \rangle \in \mathbb{N}^3$ , che permette di creare una corrispondenza tra i valori della mappa e quelli del template, cioè:*

$$T[i][j][k] \leftrightarrow M[i + P_x][j + P_y][k + P_z]$$

**Definizione 2.4 (Posizione ammissibile)** *Una posizione ammissibile per un template all'interno della mappa è rappresentata da una tripla  $\langle P_x, P_y, P_z \rangle \in \mathbb{N}^3$  tale che:*

$$\forall i, j, k : T[i][j][k] \leq M[i + P_x][j + P_y][k + P_z]$$

Per agevolare la ricerca si è estesa la dimensione della mappa di densità in modo che fosse potenza di otto. Così facendo l'albero che si ottiene è un albero completo.

**Proposizione 2.1** *La dimensione della mappa è maggiore rispetto alla dimensione del template, da cui ne deriva che anche il numero delle foglie della mappa è maggiore del numero di foglie del template, se si chiama  $l_p$  il numero dei livelli dell'albero della proteina e  $l_t$  il numero dei livelli dell'albero del template, allora  $l_p \leq l_t$ .*

**Dimostrazione 2.1** Siano  $n_p$  il numero delle foglie dell'albero della proteina e  $n_t$  il numero delle foglie dell'albero del template da questi si ottiene che:

$$l_p = \log_8 n_p + 1$$

$$l_t = \log_8 n_t + 1$$

Sapendo che:

$$n_p \leq n_t$$

Questo perché la dimensione della mappa è maggiore alla dimensione del template. Da cui si ottiene che:

$$l_p \leq l_t$$

□

Trovare le posizioni ammissibili del template all'interno della mappa equivale a cercare di posizionare l'albero del template all'interno dell'albero della proteina, come chiarisce meglio la figura 2.2.

Per poter eseguire la ricerca bisogna essere a conoscenza del numero dei li-

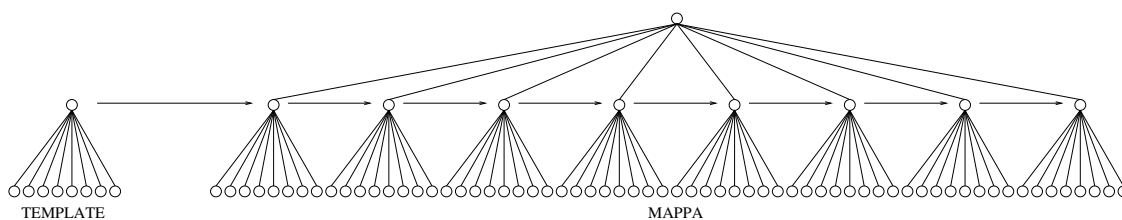


Figura 2.2: Ricerca posizioni ammissibili

velli dell'albero del template e del numero dei livelli dell'albero della mappa,

chiamandoli come prima  $l_p$  e  $l_t$ . È noto che per trovare il numero di livelli dei due alberi bisogna eseguire la seguente operazione:

$$l_p = \log_8 n_p + 1$$

$$l_t = \log_8 n_t + 1$$

dove  $n_p$  e  $n_t$  sono rispettivamente i valori delle foglie della mappa e del template.

Una volta calcolati il numero dei livelli è necessario calcolare il livello della mappa contenente le radici dei sottoalberi della mappa aventi ugual numero di livelli del template per fare questo si esegue la seguente operazione:

$$l_{rad} = l_p - l_t + 1$$

A questo punto può essere utile vedere lo pseudo codice dell'algoritmo di ricerca:

```

1 bool ricerca(mappa[], template[], livello, nodopartenzamappa, altezzatemplate)
2     tmp = nodopartenzamappa;
3     nodotemplate = 1;
4     for(int i=0; i<=livello; ++i)
5         nodotemplate += 8^i;
6     for(int i=0; i<=8^livello-1; ++i){
7         if(mappa[tmp]>=template[nodotemplate])
8             ++tmp;
9             ++nodotemplate;
10        else
11            return false;
12    }
12    ++livello;
14    nodopartenzamappa = primofiglio(nodopartenzamappa);
15    if(livello<=altezzatemplate)
16        confronto(mappa[], template[], livello, nodopartenzamappa,
17        altezzatemplate)
17    else
18        return true;
19 }
```

Figura 2.3: Un template e una mappa per il confronto

La funzione ricorsiva `ricerca` prende in input l'array `mappa[]` contenente i valori dei nodi dell'octree, nell'array nella posizione 0 è salvato il valore

della radice, subito dopo i valori contenuti nei nodi del livello sottostante, si procede per livelli fino ad aver inserito tutti i valori di tutti i nodi dell'albero. Lo stesso discorso vale per l'array `template` che contiene i valori del template. La funzione prende poi in input il livello del template che si sta analizzando, l'indice del nodo del template da cui si parte a eseguire i confronti e l'altezza dell'octree del template.

La funzione analizza gli alberi per livelli, cioè esegue una visita di tipo breadth first; il ciclo `for` della riga 6 serve proprio per confrontare i valori contenuti in tutti i nodi del livello che si sta analizzando. Nel caso il valore del template sia maggiore del valore della mappa (riga 7), la ricerca su quel sottoalbero della mappa viene interrotta e si restituisce che la posizione non è ammissibile per il template.

Se invece confrontando tutto il sottoalbero della mappa e l'albero del template si arriva all'ultima foglia senza fallimenti la funzione ritorna `true` (riga 18) e la posizione viene considerata ammissibile.

## 2.3 Prove e risultati

Per testare la validità di tutto quello che si è detto fin ora si è creato un semplice programma che, presa in input la mappa di densità di una proteina, esegue su di essa le seguenti operazioni:

- Quantizza i dati contenuti nella mappa secondo la definizione 2.1.
- Costruisce un octree su questi dati come descritto nella sezione 2.1.1.

Dopo aver costruito l'octree della mappa di densità si è simulato il template come descritto nella sezione 2.1.2 costruendo l'octree del template.

Una volta a conoscenza sia dell'octree della mappa che dell'octree del template si è passati a eseguire la ricerca delle posizioni ammissibili per il template nella mappa.

Durante la ricerca si è soprattutto cercato di valutare il numero dei test che venivano eseguiti, in caso di fallimento il livello in cui questo avveniva e il numero di successi che si ottenevano.

I risultati che si sono ottenuti hanno evidenziato i seguenti aspetti:

- Si è in grado di trovare posizioni ammissibili.
- In caso di fallimento si fallisce quasi sempre al primo livello.

La cosa maggiormente interessante è il fatto che in caso di fallimento si fallisce quasi sempre al primo livello dell'albero, questo significa che con un unico

test si è in grado di dare una risposta. Se invece di eseguisse lo stesso confronto usando solo i valori contenuti nelle foglie (ricerca puntuale) il numero di test richiesti prima di riscontrare il fallimento sarebbe molto maggiore. È d'altra parte vero che nel caso la posizione che si sta analizzando sia ammissibile, il numero di test richiesti nel caso del confronto tra gli octree è maggiore rispetto al confronto puntuale. Ma essendo il numero di test che non vengono effettuati nel caso di fallimento strettamente maggiore rispetto al numero dei test che vengono sprecati nel caso di successo, questo rende accettabile il numero di test in più. È evidente che il problema giocattolo ha mostrato che le idee che stanno alla base del problema che realmente si vuole risolvere hanno fondamento. A questo punto si può proseguire nello studio del problema iniziando a ricercare la struttura dati per risolvere il problema presentato nella Prefazione.

# Capitolo 3

## Ricerca con templates multipli

### 3.1 Rappresentazione di templates multipli in una struttura dati

Lo scopo principale di questa tesi è lo studio di una struttura dati che contenga tutti i templates di cui andranno cercate le posizioni ammissibili all'interno della mappa.

Si riassumono qui brevemente le caratteristiche che la struttura dati che si sta cercando deve avere:

- Sono contenuti gli alberi di tutti i templates.
- Templates che condividono gli stessi valori nei primi nodi sono memorizzati una volta sola e non due.
- La struttura dell'A-albero è già predisposta per una visita di tipo breadth first, questo perché un fallimento in un nodo che appartiene ai primi livelli dell'albero fa risparmiare più test. Essendo che una visita di tipo breadth first analizza prima i nodi appartenenti ai primi livelli. In questa situazione questo tipo di visita è più efficiente.
- Ogni nodo non punta ai suoi otto figli ma al nodo successore.

La struttura dati che si sta cercando deve permettere di eseguire la ricerca di tutti i templates in modo efficiente per ridurre i tempi della ricerca.

La struttura dati deve cercare di compattare il più possibile i dati contenuti al suo interno e mantenere la struttura ad octtree descritta nel paragrafo 2.1.1.

Si presentano a questo punto alcune definizioni che serviranno nella descrizione della struttura dati contenente i templates.

**Definizione 3.1 (A-nodo)** Un A-nodo è una lista dei valori contenuti nei nodi dei templates e a ogni valore è associato un riferimento ad un altro A-nodo.

**Definizione 3.2 (templates multipli)** Un template multiplo è un template che contiene al suo interno un insieme di templates singoli.

**Definizione 3.3 (A-albero)** Un A-albero è una struttura dati basata sugli octree avente come nodi gli A-nodi della definizione 3.1.

Una caratteristica degli A-alberi è data dal fatto che ogni A-nodo punta al suo successore, dove il successore rispetta la seguente definizione.

**Definizione 3.4 (A-nodo successore)** Il successore di un A-nodo  $x$  ha le seguenti caratterizzazioni:

- Se  $x$  non è l'ultimo A-nodo di un livello il suo successore è l'A-nodo che si trova immediatamente alla sua destra nell'albero.
- Se  $x$  è l'ultimo A-nodo di un livello il suo successore corrisponde al primo nodo a sinistra del livello successivo.

Si può implementare un A-nodo come un vettore dove i valori contenuti in esso sono gli indici del vettore, mentre i riferimenti sono dei puntatori, come si vede dalla figura 3.1.

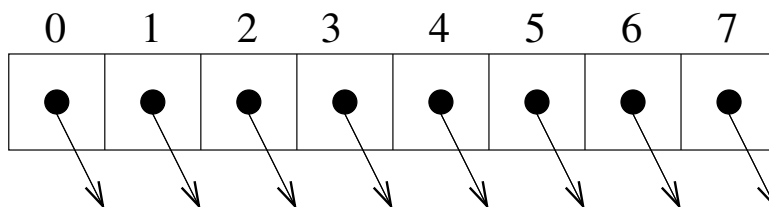


Figura 3.1: A-nodo

Ogni indice presente all'interno di un A-nodo rappresenta:

- Un insieme di templates.
- Per ciascun template dell'insieme c'è una corrispondenza tra A-nodo e nodo di un octree.
- Una visita breadth first dei templates visita gli stessi valori per ciascun template. I valori visitati sono gli indici degli array degli A-nodi visitati negli A-alberi.

L'idea che sta alla base per il raggruppamento degli octree è che due templates diversi (octree) sono uniti per le parti comuni. Si prevedono più templates diversi. A seconda del loro valore in un nodo corrispondente si segue un percorso diverso nell'A-albero. È necessario a questo punto spiegare come è strutturato un octree con templates multipli. Si parte prendendo in considerazione la radice:

- La radice è un A-nodo avente come indici i valori delle radici di tutti i templates.
- Ogni cella dell'A-nodo contiene un puntatore ad un altro A-nodo oppure `null` nel caso non esistano templates aventi quel valore alla radice.
- Nel caso esista più di un template contenente lo stesso valore alla radice esso viene contato una sola volta.

Ogni A-nodo contenuto all'interno dell'albero ha le stesse caratteristiche descritte per la radice.

Dopo aver descritto la struttura di un A-nodo è necessario descrivere la struttura di un A-albero. Per capire meglio una prima caratteristica di un A-albero ci si può aiutare con la figura 3.2. Nella figura si vede un A-albero contenente solo due templates. All'inizio l'A-albero scende per un'unica via, questo significa che:

- I due templates hanno gli stessi valori in quei nodi e
- All'interno di quegli A-nodi esiste un solo indice che ha il puntatore associato diverso da `null`.

L'A-nodo  $x$  invece punta a due A-nodi diversi, questo significa che i corrispondenti nodi  $x$  dei templates hanno valori diversi. Avendo i due templates valori differenti nel nodo  $x$  significa che all'interno dell'A-nodo  $x$  ci sono due indici diversi aventi un puntatore diverso da `null`.

Sempre dalla figura 3.2 si vede che i due templates all'interno dell'A-albero scendendo seguono il proprio albero. Questo anche nel caso che all'interno dei due templates successivamente esistano nodi corrispondenti aventi lo stesso valore. Il fatto che due templates scendano separati dopo che si è trovato il primo valore diverso serve per poter distinguere i templates all'interno dell'A-albero. A questo punto il discorso appena fatto per due alberi è estendibile per un numero maggiore di templates. Se si considerano più di due templates si ottiene che l'A-nodo che è la radice dell'A-albero ha più celle contenente dei puntatori diversi da `null` e ognuno di questi è la radice di almeno un template. Ogni A-nodo puntato dalla radice ha il suo array

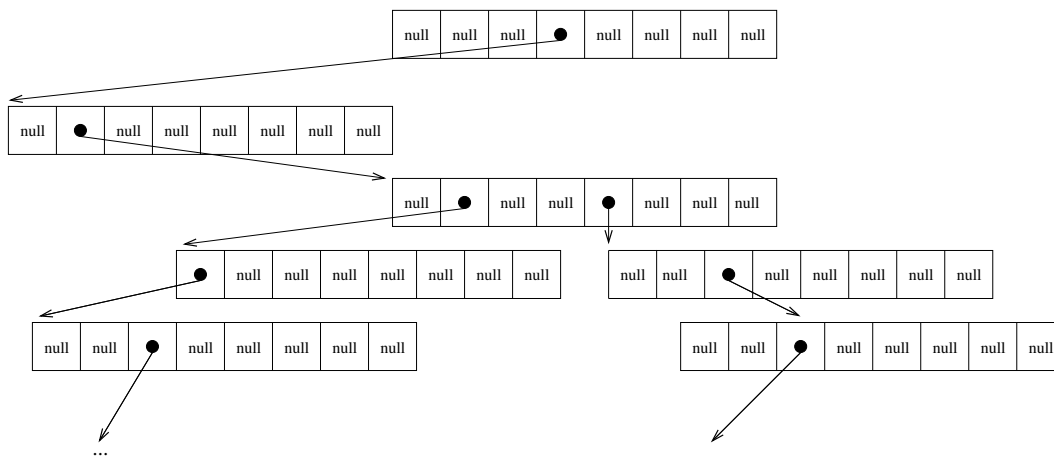


Figura 3.2: A-albero contenente due templates

contenente i puntatori. È quindi possibile che due o più templates si separino in qualsiasi nodo dell'albero. Se due templates arrivano fino all'ultimo A-nodo dell'albero senza essersi mai separati, sono uguali.

Normalmente negli alberi ogni nodo punta ai suoi figli, quindi il fatto che invece in un A-albero ogni A-nodo punta al suo successore è una peculiarità di questa struttura dati.

Questa caratteristica è messa in evidenza anche dalla figura 3.2, le frecce che partono da ogni nodo rappresentano i puntatori, in un octree normale ogni nodo ha otto puntatori che puntano agli otto figli. In questa situazione se si avesse un unico template da ogni A-nodo partirebbe un solo puntatore verso il successore del nodo.

Nella sezione 2.2 viene descritto il tipo di ricerca che si deve eseguire sulle mappe e si è mostrato come sia conveniente fare una ricerca di tipo breadth first. Anche sugli A-alberi è necessario eseguire una visita di tipo breadth first. In una visita di tipo breadth first, prima di scendere al livello sottostante, bisogna aver analizzato tutti i nodi di quel livello. Un A-albero è organizzato per offrire visite efficienti di tipo breadth first. Infatti se si considera un A-nodo  $x$  e si deve proseguire con la visita in ampiezza dell'A-albero gli A-nodi da analizzare sono proprio gli A-nodi puntati da  $x$ . Quindi rispetto agli alberi che si usano normalmente, che permettono un accesso diretto ai nodi da visitare durante una visita di tipo depth first, in una visita tipo breadth first bisogna appoggiarsi a strutture dati ausiliarie (per esempio una coda). Con gli A-alberi si può effettuare una visita in ampiezza dell'A-albero accedendo direttamente dall'A-nodo che è appena stato visitato all'A-nodo

da visitare subito dopo.

Oltre tutto il salvataggio del solo puntatore al nodo successivo diventa utile quando si vuole salvare un A-albero, in un albero a  $n$  vie, si salva direttamente il contenuto dei nodi senza puntatori. Infatti un A-albero è sempre un'estensione octree, quindi ogni A-nodo ha otto figli, ma con più vie per identificarlo.

## 3.2 Octree con liste di puntatori

A questo punto è il caso di guardare una prima possibile implementazione per la struttura descritta in 3.1. È quindi necessario spiegare meglio l'implementazione di un A-nodo. All'interno del vettore di un A-nodo bisogna salvare i puntatori agli A-nodi successivi. Ogni puntatore viene salvato nella cella avente come indice il valore del nodo nel singolo template. Per chiarire meglio qual è la situazione corrente si guardi la figura 3.3.

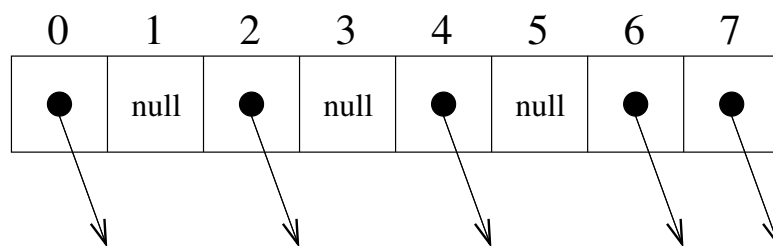


Figura 3.3: A-nodo

È possibile che alcune celle dell'array contengano il valore `null`. Infatti è possibile che alcuni valori non si trovino in nessun nodo.

### 3.2.1 Complessità in spazio

Dopo aver descritto una prima possibile implementazione degli A-alberi si analizza l'occupazione in memoria. Si inizia analizzando l'occupazione in memoria di un singolo nodo, per poi analizzare l'occupazione in memoria di un A-albero.

Per calcolare l'occupazione in memoria di un singolo nodo è necessario iniziare l'analisi dalle foglie. I valori contenuti nelle foglie di ogni albero sono quantizzate secondo i livelli di quantizzazione descritti in 2.1, quindi varieranno nell'intervallo  $[0, q-1]$ . Se si chiama  $D_p$  la dimensione in spazio di un

puntatore si ha che ogni foglia occupa:

$$D_p \cdot q$$

A questo punto si deve vedere l'occupazione in memoria di un A-nodo che appartiene al penultimo livello di un A-albero, si sa che il valore contenuto all'interno di un nodo è dato dalla somma dei valori degli otto figli, come visto in 2.1.1, da cui si ottiene che i valori dei nodi del penultimo livello variano nell'intervallo  $[0, 8q-1]$ , da cui l'occupazione in memoria è:

$$D_p \cdot 8q$$

Generalizzando la situazione si ha che se  $l_{tot}$  è il numero totale dei livelli dell'albero e si vuole calcolare il numero degli indici del livello  $k$  si ha che l'occupazione in memoria di generico nodo del livello  $k$  è:

$$D_p \cdot 8^{l_{tot}-k-1}q$$

È evidente il fatto che nei primi livelli di un A-albero l'occupazione in memoria di A-nodo è maggiore rispetto all'occupazione in memoria di un A-nodo che si trova negli ultimi livelli dell'A-albero. A questo punto è possibile anche vedere la complessità in spazio di un A-albero.

In questa situazione si assuma di contare lo spazio minimo richiesto per la rappresentazione di un template in un A-albero.

**Proposizione 3.1** *L'occupazione di spazio in memoria è  $O(nq \cdot \log n)$ , dove  $n$  è il numero dei campioni della mappa, cioè le foglie dell'A-albero e  $q$  sono i livelli di quantizzazione.*

**Dimostrazione 3.1** Si inizia contando dall'ultimo livello dell'albero, quindi dalle foglie, in questo caso abbiamo  $n$  nodi, con l'array che ha come massima dimensione  $q$ , quindi in questa situazione abbiamo  $nq$  elementi. A questi vanno sommati gli A-nodi appartenenti al livello superiore questi sono:

$$\frac{n}{8}$$

perché un A-nodo del livello in analisi corrisponde a otto nodi del livello successivo e il numero di elementi contenuto in ciascun A-nodo è  $8q$ , questo perché come si è visto nei calcoli dell'occupazione in memoria di un singolo nodo ogni volta che si passa al livello superiore l'ampiezza dell'intervallo a cui appartengono i valori aumenta di un fattore 8, da cui si ottiene:

$$\frac{n}{8} \cdot 8q = nq$$

Generalizzando quindi questi calcoli al livello  $l$  e sia  $l_{tot}$  il numero totale dei livelli dell'albero si ha che il numero degli A-nodi è  $\frac{n}{8^{l_{tot}-l}}$  mentre il massimo indice dell'array è  $8^{l_{tot}-l}q$ , quindi si ha che:

$$\frac{n}{8^{l_{tot}-l}} \cdot 8^{l_{tot}-l}q = nq$$

A questo punto è necessario calcolare il numero dei livelli dell'albero e si ottiene che:

$$numero\_livelli = \lceil \log_8 n \rceil \leq \log_8 n + 1$$

Si ha che  $\log_8 n + 1$  è il numero dei livelli dell'albero, quindi si ha come massima occupazione di spazio:

$$D_p \cdot nq \cdot (\log_8 n + 1)$$

Usando la notazione asintotica si ha che l'occupazione è  $O(nq \log n)$ .

□

### 3.2.2 Valutazione

Il difetto principale di questa struttura dati è il fatto che più si sale all'interno dell'albero e più aumenta la dimensione di ogni A-nodo, pur rimanendo uguale il numero dei templates che si stanno inserendo, quindi il rischio che si corre è quello di sprecare molta memoria per contenere dei puntatori a `null`. Essendo l'occupazione di memoria uno dei problemi che si vuole minimizzare con la struttura dati si vuole trovare un modo per evitare di lasciare delle celle vuote e allocate.

## 3.3 Octree con quantizzazione fissa

In questa sezione si presenta una nuova implementazione che cerca di compattare ulteriormente i dati all'interno dell'albero. Per cercare di compattare i dati si sfrutta l'approssimazione descritta nella sezione 3.3.1.

Infatti se si è disposti ad approssimare ulteriormente i dati contenuti nell'A-albero si porta l'A-albero ad una occupazione lineare nel numero di foglie.

### 3.3.1 Un'approssimazione delle strutture della mappa

Per rendere ancora più compatta la rappresentazione della mappa si può eseguire una approssimazione sui dati che vengono inseriti all'interno dell'A-albero. Di seguito si presenta l'approssimazione che viene usata nella successiva implementazione per i templates multipli, cercando di metterne in

evidenza i pregi.

Per eseguire l'approssimazione si sfrutta il fatto che i valori contenuti nei nodi interni è dato dalla somma dei valori contenuti negli otto figli come descritto nella sezione 2.1.1. Si può dare la seguente definizione:

**Definizione 3.5 (Valore di un nodo)** *Si definisce valore di un nodo la seguente uguaglianza:*

$$V(\text{nodo}) = \sum_{F \in \text{figli}(\text{nodo})} V(F)$$

*Dove figli(nodo) è l'insieme di tutti i nodi figli di quel nodo.*

In questa nuova implementazione invece di salvare all'interno dell'A-nodo corrispondente il  $V(\text{nodo})$  si esegue la seguente operazione:

$$V(\text{nodo})/8$$

Si esegue una divisione per i seguenti motivi:

- Essendo 8 il numero dei figli,  $V(\text{nodo})$  varia nell'intervallo  $[0, 8q - 1]$ .
- Dopo di che si vuole riportare  $V(\text{nodo})$  all'interno dell'intervallo  $[0, q - 1]$ .
- Per ottenere  $V(\text{nodo})$  all'interno dell'intervallo più piccolo bisogna eseguire la seguente operazione  $V(\text{nodo})/8$ .

A questo punto per presentare una caratteristica di questa implementazione, c'è bisogno di una definizione.

**Definizione 3.6 (Range)** *Un range è un insieme di valori appartenenti a un intervallo. Siano  $l$  un generico livello dell'A-albero,  $l_{\text{tot}}$  il numero totale dei livelli e  $i$  un generico valore contenuto in un A-nodo allora si ha che:*

$$\text{range} = [i \cdot 2^{l_{\text{tot}}-l-1}, (i+1) \cdot 2^{l_{\text{tot}}-l-1} - 1]$$

Il range è utile perché il valore di un A-nodo caratterizza tutti i nodi dei templates con valori nel range del puntatore dell'A-nodo. Se si prende in considerazione un qualsiasi A-nodo si ha che i valori contenuti all'interno degli A-nodi rappresentano un range di possibili valori. Nei primi livelli dell'albero i range di valori sono più grandi, più si scende nell'albero maggiormente i range si rimpiccioliscono fino ad arrivare alle foglie dove un valore non rappresenta un range ma proprio il valore.

### 3.3.2 Complessità in spazio

Anche in questa situazione si assume di contare lo spazio minimo richiesto per l'occupazione di un template.

**Proposizione 3.2** *Il costo di spazio nel caso peggiore di questa struttura è  $O(nq)$ . Dove  $n = 8^k$  con  $k \in \mathbb{N}$  è il numero delle foglie, cioè il numero di campioni presi dai templates.*

**Dimostrazione 3.2** Si ha che il numero totale degli A-nodi  $n_{tot}$  è:

$$n_{tot} = \sum_{i=0}^k 8^i$$

Lo spazio totale è la dimensione di un puntatore,  $D_p$ , per i livelli di quantizzazione:

$$(D_p \cdot \sum_{i=0}^k 8^i)q = D_p \cdot \frac{8^{k+1} - 1}{8 - 1} \cdot q = D_p \cdot \frac{8^{\log_8 n+1} - 1}{8 - 1} \cdot q$$

Passando quindi alla notazione asintotica si ha che il risultato scritto sopra un  $O(nq)$ .

□

### 3.3.3 Valutazione

Il pregio principale di questa implementazione rispetto a quella presentata nella sezione 3.2 è la minor occupazione di memoria. Un secondo pregio è che la dimensione degli array costante in tutto l'albero rende più facile l'implementazione. Infatti il numero dei livelli di quantizzazione viene deciso e reso costante all'inizio dell'implementazione, di conseguenza una volta che si conosce questo valore si conosce anche quanto spazio in memoria occupa ogni A-nodo. Contrariamente, nella prima implementazione per allocare in memoria lo spazio di un A-nodo oltre al numero dei livelli di quantizzazione bisogna anche conoscere in quale livello dell'A-albero ci si trova.

Si deve sottolineare il fatto che durante la ricerca si ragiona per range, questa caratteristica ha sia dei vantaggi che uno svantaggio:

**pro** Fare un test su un intervallo permette di testare contemporaneamente un insieme di valori molto vicini fra di loro.

**pro** La struttura è più insensibile alle fluttuazioni.

**contro** La ricerca può fallire più in basso.

Dire che la struttura è più insensibile alle fluttuazioni significa che la struttura dati è meno influenzata dagli errori presenti all'interno dei dati sperimentali. Questo è dovuto al fatto di quantizzare i dati, quindi il dato reale e il dato ottenuto sperimentalmente possono cadere nello stesso intervallo annullando quindi gli errori.

Il fatto che la ricerca possa fallire più in basso è dovuta dal fatto che i primi nodi hanno un intervallo di valori ampio quindi spesso si potrebbe pensare di poter scendere per quell'albero. Successivamente analizzando i nodi contenenti i figli ci si rende conto di aver preso una strada inesistente. Per chiarire meglio la situazione si può fare riferimento all'esempio 3.4. Nell'esempio si

livello	A-albero		Octree		Realitivi ranges
i	0	$\xleftrightarrow{si}$	0	$\longleftrightarrow$	$[0 \cdot 8^i, (1 \cdot 8^i) - 1]$
	$\downarrow$		$\downarrow$		$\downarrow$
i+1	2	$\xleftrightarrow{no}$	3	$\longleftrightarrow$	$[2 \cdot 8^{i-1}, (3 \cdot 8^{i-1}) - 1] \neq [3 \cdot 8^{i-1}, (4 \cdot 8^{i-1}) - 1]$

Figura 3.4: Esempio

vuole chiarire ulteriormente il fatto che si può fallire più in basso. Infatti al livello  $i$  i valori presenti nell'A-albero appartengono ad un intervallo che contiene il valore presente nell'octree. Successivamente quando si scende di un livello si vede che il valore contenuto nell'A-albero non è ammissibile rispetto a quello contenuto nell'octree e quindi la ricerca viene fermata.

### 3.4 Ricerca

Con questa nuova rappresentazione dei templates si presentano anche alcuni cambiamenti nell'algoritmo di ricerca rispetto all'algoritmo presentato in 2.2. Le differenze sono dovute soprattutto dal fatto che nell'algoritmo di 2.2 ogni nodo contiene un unico valore, mentre in questa situazione ogni nodo contiene un insieme di valori. Quando si va ad effettuare la ricerca in ogni A-nodo bisogna controllare l'esistenza di puntatori nelle celle dell'array aventi l'indice minore o uguale al valore contenuto nella mappa.

Per chiarire meglio come viene eseguita la ricerca si può presentare lo pseudo codice:

```

1  ricercapositioni(A-nodo, nodo){
2    if(A-nodo != ultimafoglia){
3      for(int i=0; i<=V(nodo); ++i){
4        if(A-nodo.puntatori[i] != NULL){
5          tmp = A-nodo
6          A-nodo = A-nodo.puntatori[i]
7          ricercapositioni(A-nodo, nodo+1)
8          A-nodo = tmp
9        }
10     }
11  }
12  else
13    if(A-nodo <= nodo)
14      return template
15    else return 0
16  return 0
17 }

```

Figura 3.5: Funzione che ricerca le posizioni ammissibili sugli A-alberi

La funzione `ricercapositioni` prende in input un oggetto di tipo `A-nodo` che rappresenta un `A-nodo` e `nodo` che è l'indice del nodo dell'octree in analisi.

Il ciclo `for` presente alla riga 3 serve per controllare che tutti i templates con valore minore uguale a quello della mappa all'interno dell'`A-nodo`. Il controllo alla riga 4 serve per verificare se in corrispondenza di quel valore all'interno dell'`A-nodo` esiste un puntatore, se esso esiste allora è possibile continuare la ricerca, altrimenti la ricerca per quella possibile via viene fermata.

Alla riga 14 si vede che se la ricerca ha avuto successo vengono restituiti i templates che possono essere posizionati all'interno di quella posizione della mappa. Infatti i primi valori che saranno passati alla funzione saranno proprio quelli delle due radici, quindi tutti i templates che avranno il valore della radice maggiore del valore della radice della mappa risultano testati e sono già stati eliminati. Si prosegue così per tutti i nodi che si riesce a raggiungere, fino a trovare i templates che sono ammissibili.

Questa implementazione presenta un ulteriore vantaggio che è la ricerca approssimata.

### 3.4.1 La ricerca approssimata

La ricerca approssimata permette di non eseguire la ricerca sull'intera altezza dell'albero ma si può decidere di fermarsi ad un determinato livello, invece di arrivare fino alle foglie.

Eseguendo la ricerca in questo modo si perde sicuramente in precisione, perché la ricerca diventa più grossolana sia in spazio che in densità ma è però più veloce. Può essere, però, molto utile soprattutto perché è noto che i dati possono essere affetti da errori, quindi quando si pensa di essere in presenza di dati che possono aver subito errori relativamente grandi, non eseguire la ricerca su tutto l'albero ma fermarsi prima permette di trovare come ammissibili posizioni che probabilmente non si sarebbero trovate proprio a causa degli errori sui dati.

Per eseguire la ricerca approssimata è necessario fissare una soglia.

**Definizione 3.7 (Soglia)** *La soglia è il livello al quale ci si ferma per la ricerca.*

Quindi se si sono testati tutti i nodi della soglia senza fallire allora la posizione viene considerata ammissibile.

La ricerca approssimata:

- È più veloce perché non vengono eseguiti tutti i test. Infatti se la posizione è ammissibile ci si ferma alla soglia e quindi si risparmia un numero di test pari al numero dei nodi che si trovano nei livelli successivi alla soglia.
- È più grossolano perché si decide fino a che precisione si vuole arrivare, cioè si decide la dimensione dell'intervallo oltre il quale non si vuole scendere per assumere che la posizione è ammissibile.

I metodi utilizzati per la ricerca di posizioni ammissibili, come quella presentata in 1.3, non permettono di effettuare la ricerca approssimata.

# Capitolo 4

## Implementazioni e risultati

### 4.1 La classe template

In questa sezione si vuole presentare, con maggiori dettagli, l'implementazione della classe `template` che è la classe che si occupa della creazione e gestione degli A-alberi descritti nella sezione 3.3.

Per l'implementazione della classe si usa una struct chiamata `nodo` che rappresenta la struttura di un A-nodo, la struct contiene i seguenti dati:

- `int indice`, è l'indice dell'A-nodo.
- `int* puntatori`, è l'array che contiene l'indice degli A-nodi puntati dall'A-nodo in analisi.
- `int padre`, è l'indice del nodo che precede il nodo in analisi, cioè il nodo in analisi è il successore, secondo la definizione 3.4, dell'A-nodo avente come indice il valore contenuto in `padre`.
- `int numTemplate`, contiene un valore diverso da -1 solo nel caso in cui l'A-nodo sia l'ultima foglia di un template e il valore rappresenta l'indice del template.
- `int valore`, se l'A-nodo è l'ultima foglia contiene il valore del nodo, altrimenti contiene il valore -1.

La classe `template` usa la struct `nodo` per la gestione degli A-nodi contenuti negli A-alberi. La classe `templateTree` è caratterizzata dai seguenti dati:

- `const static int q=8`, è il numero dei livelli quantizzazione che si sono scelti di usare durante l'implementazione.
- `vector<nodo> albero`, è il vettore che contiene i nodi dell'A-albero.

- `int templatesize`, è la dimensione di un lato del cubo dei templates.
- `double densitacarica`, è il rapporto tra la densità e la carica elettrica del primo template che viene caricato all'interno dell'A-albero.

Si è scelto di salvare gli A-nodi all'interno di un vettore per evitare la gestione diretta dei puntatori, in questo modo per riferirsi ad un A-nodo si usa l'indice del nodo all'interno del vettore e non il suo puntatore. Questa scelta spiega anche il fatto che tra i dati della struct `nodo` ci sia una variabile indice che contiene appunto l'indice del nodo all'interno del vettore `albero` e allo stesso modo l'array che dovrebbe contenere i puntatori ai nodi successivi non contiene i puntatori ma gli indici dei nodi.

La variabile `templatesize` serve per ricavare la dimensione del cubo contenente i templates, cioè anche il numero delle foglie di un octree, e l'altezza di un albero di un template. Oltretutto la scelta di salvare la dimensione di un solo lato è data da fatto che si è deciso che i templates dovevano essere dei cubi e quindi si ha che:

$$dim_x = dim_y = dim_z = templatesize$$

La variabile `densitacarica` una volta calcolata per il primo template viene usata per ottenere una quantizzazione omogenea sia tra i templates che vengono inseriti all'interno dell'A-albero, sia per la creazione dell'octree della mappa.

I metodi che vengono utilizzati dalla classe `templateTree` sono i seguenti:

- `TemplateTree()`, è il costruttore della classe e si occupa esclusivamente di inserire la radice nel vettore senza assegnarle valori.
- `vector<nodo> aggiungiAlbero(unsigned char* foglie, int dimensioneFoglie, int numeroTemplate, vector<nodo> albero, int livelli, int dimensione, unsigned char* tmptemplate)`, è la funzione che prende in input le foglie del nuovo template da inserire all'interno dell'A-albero, il numero delle foglie, l'indice del template che si sta inserendo nell'A-albero, il vettore contenente l'albero, il numero dei livelli di un template, la dimensione del template e un array vuoto che è solo stato allocato e che conterrà l'octree del template che si sta inserendo nell'A-albero. La funzione restituisce l'A-albero a cui è stato aggiunto un nuovo template.

La funzione prima di tutto si occupa di costruire il template da inserire nell'A-albero richiamando una funzione che si trova esternamente alla classe e che si occupa di costruire l'octree come descritto nella sezione 2.1.1. Successivamente la funzione si occupa di inserire l'octree all'interno dell'A-albero come descritto nella sezione 3.3.

- `void creafile(templateTree templates)`, è la funzione che si occupa di creare il file che contiene i dati di un A-albero e prende in input proprio un A-albero.
- `templateTree caricadafile(string nomefile, templateTree templates)`, la funzione prende in input il nome del file che contiene i dati da inserire nell'A-albero e restituisce un oggetto di tipo `templateTree` contenente i dati salvati nei file.

All'interno della classe `templateTree` i dati dei vari template vengono caricati con i valori reali  $T[i][j][k]$ , e questi vengono quantizzati nel seguente modo:

$$T'[i][j][k] = T[i][j][k] / \text{densita} \cdot \text{carica} \cdot \text{templateTree.densitacarica}$$

dove `densita` è la densità del template e `carica` è la sua carica elettrica.

## 4.2 La classe mappa

In questa sezione si presenta invece l'implementazione della classe `mappa`, che gestisce l'implementazione delle mappe di densità delle proteine. All'interno di questa classe i dati presenti sono i seguenti:

- `const static int q=8`, è il numero dei livelli di quantizzazione che si sono scelti di usare durante l'implementazione.
- `unsigned char* foglie`, contiene i dati di un pezzo di mappa avente le stesse dimensioni di un template.
- `int mappasizex`, equivale a  $dim_x$  definito nella sezione 1.1.
- `int mappasizey`, equivale a  $dim_y$  definito nella sezione 1.1.
- `int mappasizez`, equivale a  $dim_z$  definito nella sezione 1.1.
- `int dimalbero`, è la dimensione dell'albero di un template, cioè la dimensione che deve avere l'albero della mappa da confrontare con i template.
- `unsigned char* albero`, è l'array che contiene l'octree delle stesse dimensioni di un template e che andrà confrontato con un template.
- `vector<int> templok`, contiene gli indici dei template che possono essere inseriti in quella posizione della mappa.

- `int livellitemplate`, è il numero dei livelli dei template.

I metodi di questa classe sono i seguenti:

- `mappa(int dimLatoTempl, double denscarica)`, è il costruttore della classe e può chiamare la funzione `caricadati(ch)` se esiste già un file contenente i valori della densità della mappa, altrimenti chiama la funzione `simuladati(ch, densitacarica)`.
- `void simuladati(string ch, double densitacarica)`, è la funzione che prende in input il nome di un file `pdb` e il valore `densitacarica` calcolato all'interno della costruzione dell'A-albero dei templates e simula la densità della proteina caricata, quando la funzione termina l'array `foglie` contiene al suo interno i valori quantizzati della mappa simulata.
- `void caricadati(string ch)`, è la funzione che prende in input il nome del file che contiene al suo interno le densità della proteina, li legge, li quantizza e li salva nell'array `foglie`.
- `void caricafoglie(int posizionex, int posizioney, int posizionez, int latosize, int numshift)`, è la funzione che prende dall'array delle foglie i valori del cubo che parte dalla posizione  $P\langle\text{posizionex}, \text{posizioney}, \text{posizionez}\rangle$  e che ha la stessa dimensione del template, con questi valori costruisce l'octree che andrà confrontato con i templates.
- `void creanuovamappa(int posizionex, int posizioney, int posizionez, int dimLatoTempl)`, è la funzione che viene richiamata all'esterno della classe e richiama al suo interno la funzione `caricafoglie`. Quando la funzione termina l'octree avente le stesse dimensioni di quello di un template è stato costruito.
- `void confronto(int nodotemplattuale, int nodomappaindice, templateTree& alberoTemplate, int& numtest)`, è la funzione che esegue la ricerca delle posizioni ammissibili dei templates nella posizione seguendo l'algoritmo descritto nella sezione 3.4.

Per poter eseguire la ricerca di tutti i templates all'interno della mappa all'interno del `main` del programma è necessario eseguire le funzioni `creanuovamappa` e `confronto` all'interno di tre cicli `for` per permettere di testare tutte le posizioni all'interno della mappa. In questo modo dopo aver testato tutte le possibili posizioni all'interno della mappa si ottiene per ogni posizione la lista dei templates che possono essere posizionate in quella posizione.

### 4.3 Risultati

La prima cosa che si è testata è stato vedere se gli octree venivano costruiti correttamente, per controllarli si è voluto vedere un immagine degli octree (immagine 4.1). Il punto più grande dell'immagine rappresenta la radice, circondato dai suoi otto figli, i quali sono circondati dai loro otto figli. I co-

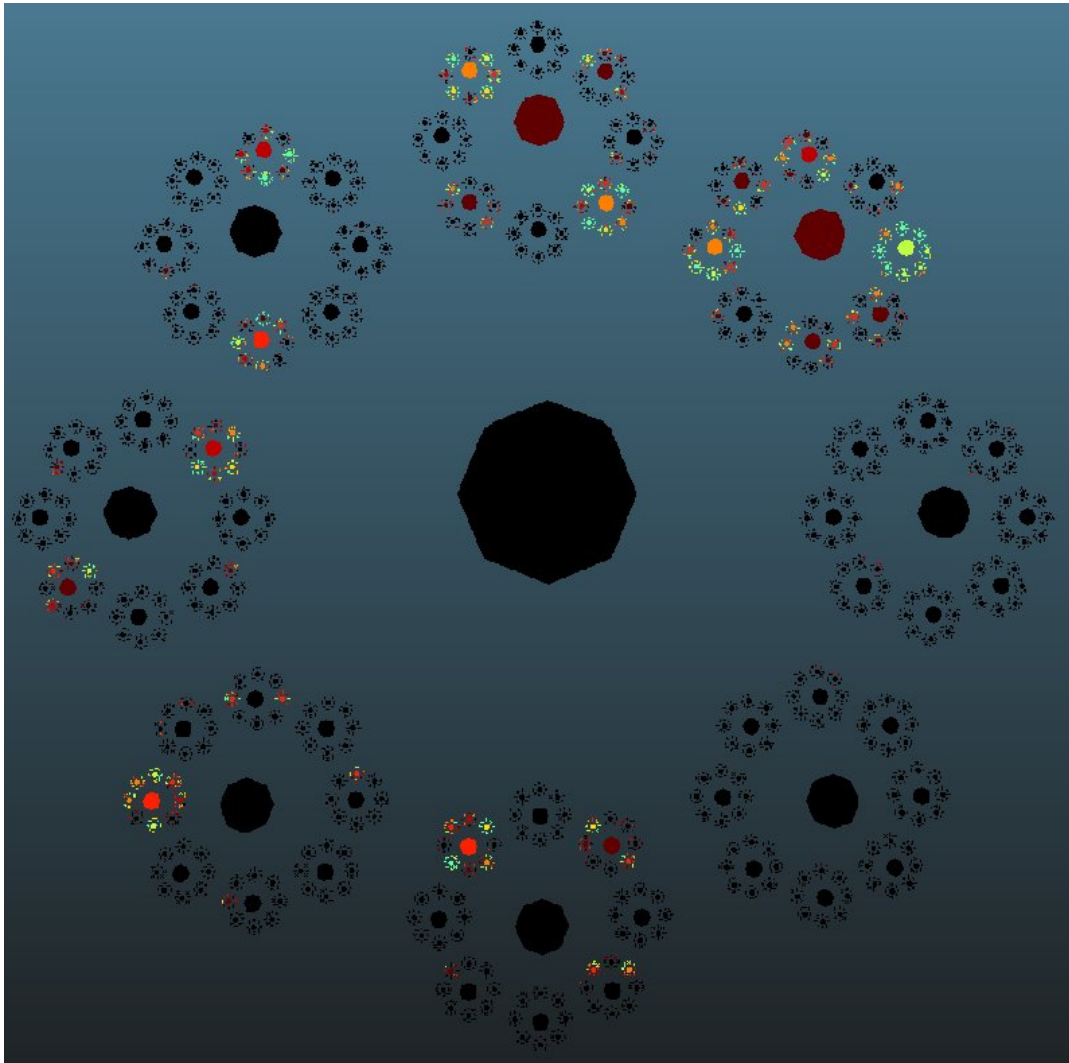


Figura 4.1: Un octree

lori dei nodi rappresentano i vari valori contenuti nei nodi, i colori più chiari rappresentano valori più alti mentre i colori più scuri rappresentano valori più bassi.

Per testare l'efficacia della struttura dati creata si è scelto di prendere una proteina dalla banca dati pdb ([www.rcsb.org](http://www.rcsb.org)) e da questa si è creato i templates, formati da alcuni amminoacidi. Della stessa proteina si è simulata la mappa di densità e all'interno di essa si sono ricercate le posizioni ammissibili per i templates.

Per tutti i test si è usato il seguente file 1E0M.pdb. Di questo file appartenente alla banca dati pdb si è simulata la densità per poter creare la mappa. Usando sempre lo stesso file si sono simulati anche i templates che sono stati inseriti all'interno dell'A-albero, i templates sono stati generati usando catene formate da alcuni amminoacidi di cui si sono simulate le densità per poter costruire la loro mappa di densità.

Il primo risultato che si è ottenuto è stato vedere che le posizioni ammissibili venivano trovate e ricoprivano per intero la proteina. Si è quindi provata la correttezza dell'implementazione.

Come secondo test si sono contati il numero di confronti tra i valori contenuti nei nodi effettuando la ricerca con la struttura dati e il numero di confronti che vengono effettuati facendo il confronto con il test puntale descritto nella sezione 1.3, i risultati che si sono ottenuti usando la stessa mappa e gli stessi templates sono i seguenti:

**A-albero:** il numero di test è circa 2000000.

**test puntale:** il numero di test è circa 200000000.

Dai risultati ottenuti è subito evidente il fatto che cercando le posizioni ammissibili usando la struttura dati si ha un guadagno di un fattore 100 rispetto alla ricerca puntale.

Il guadagno che si ottiene nella ricerca delle posizioni ammissibili è uno dei maggiori punti di forza della nuova struttura dati.

L'ultimo test che si è voluto fare è stato quello di provare ad eseguire la ricerca approssimata usando come soglia il penultimo livello. In questo caso il numero dei test eseguiti è stato di circa mezzo milione in meno rispetto alla ricerca normale. Oltretutto usando la ricerca approssimata sono state trovate alcune posizioni ammissibili in più. Il fatto di riuscire a riconoscere un numero maggiore di posizioni ammissibili, come già detto è dovuto agli errori di approssimazione sui dati. Per concludere può essere utile la tabella 4.1 contenente il numero dei test eseguiti con i tre metodi.

<b>Tipo di confronto</b>	<b>Numero di test</b>
Test puntuale	184528120
Ricerca completa	1993520
Ricerca approssimata	1422755

Tabella 4.1: Confronto tra i possibili tipi di ricerca delle posizioni ammissibili

# Capitolo 5

## Conclusioni

### 5.1 Lavori futuri

Alla fine di questa tesi si è in grado di determinare le posizioni ammissibili per i templates all'interno della mappa. Per poter determinare la struttura tridimensionale della proteina bisogna collocare un template in una delle sue posizioni ammissibili, questa operazione potrà essere fatta in un contesto di ricerca vincolata con backtracking; oppure si potrà dare un punteggio ad ogni coppia posizione-template e in base al punteggio decidere in che posizione e quale template posizionare.

Una proteina può essere suddivisa in vari templates e lo scopo è quello di ricostruire la loro disposizione.

Per posizionare un template è necessario eseguire la sua sottrazione per sottrarre all'interno della mappa la densità della proteina. Una volta che si hanno i nuovi valori della mappa si deve eseguire una nuova ricerca delle possibili posizioni dei restanti templates all'interno della mappa. La ricerca che viene effettuata presenta dei vincoli da rispettare, infatti, conoscendo la catena amminoacidica della proteina, i templates che precedono o seguono il template che è stato posizionato non potranno essere posizionati lontani da esso all'interno della mappa. Quindi una volta posizionato il primo template la cosa più logica sarà cercare se è possibile posizionare nelle sue vicinanze i templates che si trovano vicino ad esso. Nel caso in cui non si trovino posizioni ammissibili per i vicini questo significa che anche il primo template si trova in una posizione sbagliata.

Alla fine della ricerca delle posizioni dei templates, si otterrà la struttura tridimensionale a cui la mappa di densità si riferisce, dedotta dal posizionamento dei templates.

## 5.2 Utilizzi alternativi

La struttura dati che è stata creata presenta alcuni utilizzi alternativi in campi in cui la ricerca è molto attiva, infatti può essere utilizzata nei seguenti campi:

- Analisi mediche.
- Drug discovery.

Quando si parla di applicazione alle analisi mediche si intende che la struttura dati può essere usata per ricercare masse con densità e forme specifiche in esami come TAC o risonanze magnetiche.

Con il termine drug discovery si intende lo studio dei nuovi farmaci, in questa situazione non si vuole ricostruire la struttura tridimensionale, ma si vuole capire qual è la forma del sito attivo della proteina per poter in questo modo costruire le molecole che si andranno a legare. In questo caso la struttura dati può essere usata per cercare le zone della mappa di densità dove la proteina presenta una superficie compatibile con una molecola (template). I test di compatibilità possono essere effettuati con piccole modifiche all'algoritmo di ricerca che è stato presentato. L'efficienza riportata suggerisce un'applicazione nello screening di potenziali principi attivi.

# Bibliografia

- [1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduzione agli algoritmi e strutture dati. McGraw-Hill, 1990.
- [2] Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli. The Density Constraint. Conferenza 2007.
- [3] Frank DiMaio, Jude Shavlik, George N Phillips. A probabilistic approach to protein backbone tracing. 2006.
- [4] Frank DiMaio, Ameet Soni, George N. Phillips, Jr, Jude W. Shavlik. Improved methods for template-matching in electron-density maps using spherical harmonics 2007.
- [5] <http://en.wikipedia.org/wiki/Octree>
- [6] <http://it.wikipedia.org/wiki/Convoluzione>