4
Implementando Eficientemente uma 2^{d}–*Tree* Aberta

O capítulo anterior tratou da modelagem e otimização estatística das buscas numa 2^{d}–*tree* aberta, supondo a existência de um método de indexação que cumpra os quatro requisitos estabelecidos na seção 3.1. Este capítulo apresenta um método de indexação eficiente que satisfaz tais exigências. Tal método se baseia na ordem de Morton, apresentada na seção 2.4.1, e é chamado de *Códigos de Morton*. Uma *hash table* (seção 2.5.3) é usada para armazenar os nós indexados da 2^{d}–*tree* aberta, a qual é chamada de *hashed* 2^d–*tree*.

Notação Serão usados símbolos de operações de máquina tais como | (OU), & (E), deslocamento à direita $\ll$ e deslocamento à esquerda $\gg$. A notação $|_{i=1}^{d}$ representa um "outório", isto é, uma seqüência indexada de operações de OU. A notação de bits $0^n := 0...0$ (n vezes) é a repetição do elemento binário 0 no número de vezes igual à sua potência que também vale para o bit igual a 1.

4.1
Códigos de Morton

A ordem de Morton (seção 2.4.1) é uma ordenação do espaço, portanto, dada uma precisão, é possível transformar qualquer posição do espaço multidimensional em um único índice bem definido, que será chamado de *chave*. Uma das suas características importantes é a sua forma hierárquica, pois com ela é possível criar um único índice para cada nó de uma 2^{d}–*tree*, que também será chamado de *chave* (seção 2.4.2). Em outras palavras, junta a indexação hierárquica da 2^{d}–*tree* com a ordenação do espaço.

4.1.1
Codificando o Espaço

Para gerar a chave $k_m(p)$ de precisão m de uma dada posição p no hipercubo unitário no espaço d-dimensional $p = (x_1, \ldots, x_d) \in [0,1]^d$, é preciso converter cada uma das coordenadas de p para a base binária: $\lfloor x_i \times 2^m \rfloor = (b_m^i, \ldots, b_1^i)_2$, onde $b_j^i = 0$ ou 1 com $i \in \{1, \ldots, d\}$ e $j \in \{1, \ldots, m\}$. Agora, com todas as coordenadas de p na base binária, basta fazer a intercalação dos

bits de cada coordenada: $k(p) = (1\, b_m^d \ldots b_m^1 \ldots b_2^d \ldots b_2^1 \ldots b_1^d \ldots b_1^1)_2$. O *nível da chave* é a precisão m da qual ela foi gerada. A adição do '1' bit, chamado de *bit de nível*, na posição mais significante da chave tem a função de gravar na chave o seu nível, evitando o seu armazenamento de forma separada e permitindo que ele seja calculado a partir dela. Um exemplo bidimensional com precisão de dois bits e com o tamanho de cinco bits:

$$(x, y) = (2, 1) = (\underset{x_2 x_1}{10_2}, \underset{y_2 y_1}{01_2}) \sim \underset{1 x_2 y_2 x_1 y_1}{1\,1001_2} = 25.$$

A Figura 4.1 explica como é feita a geração de uma chave de precisão 8 bits e de tamanho 25 bits no espaço tridimensional. Esse processo de intercalação de bits pode ser acelerado usando operações em inteiros como dilatação e contração, que serão apresentadas no próximo capítulo, mais precisamente na seção 5.1.

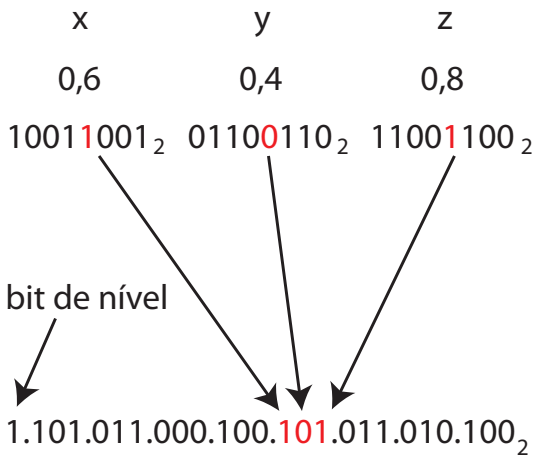


Figura 4.1: Intercalação dos bits para a geração da chave de um ponto 3D com precisão de 8 bits.

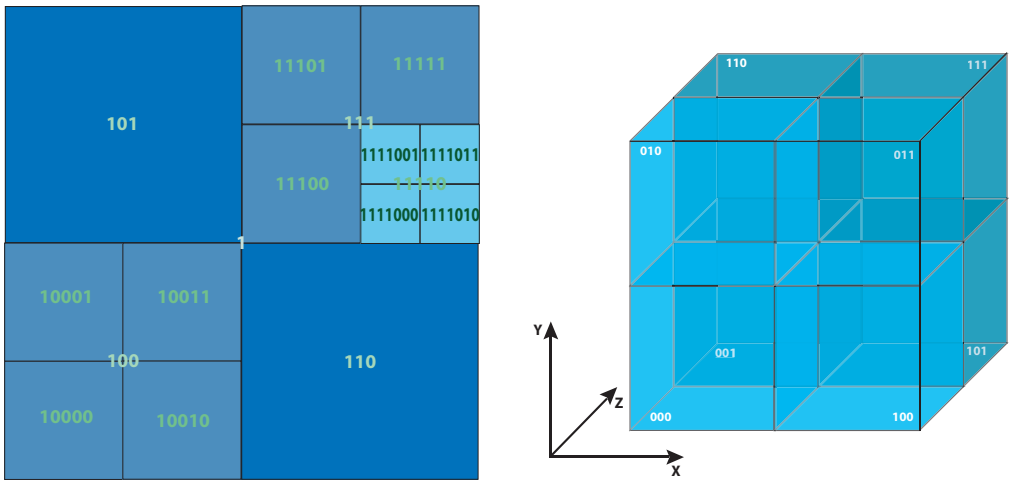
4.1.2 Codificando uma 2^d–*tree*

A chave $k(n)$ de um nó n pode ser gerada recursivamente a partir da hierarquia da 2^d–*tree*: a chave do nó raiz é 1, a chave dos filhos de um nó n qualquer é a concatenação da sua chave $k(n)$ com os d bits do código direcional dos filhos: $(k(n) \ll d) \mid (\text{código da orientação do filho})_{\text{d bits}}$. O código da orientação dos filhos, no espaço bidimensional, pode ser visto na tabela abaixo:

Direções		Códigos			
NO	NE	01	11	1	3
SO	SE	00	10	0	2

Para o espaço tridimensional, o código da orientação dos filhos pode ser visto na Figura 4.2(b). O nível de um nó n é $\lfloor \frac{1}{d} \log_2 k(n) \rfloor$, e a chave do pai de n é

obtida pelo truncamento dos d bits menos significantes da sua chave $k(n)$, da seguinte forma: $k(n) \gg d$. A Figura 4.2(a) exemplifica uma *quadtree* com seus nós codificados pelo método dos códigos de Morton.



a) Quadtree codificada com a orientação SO, NO, SL e NL

b) Sufixos dos códigos direcionais em 3D

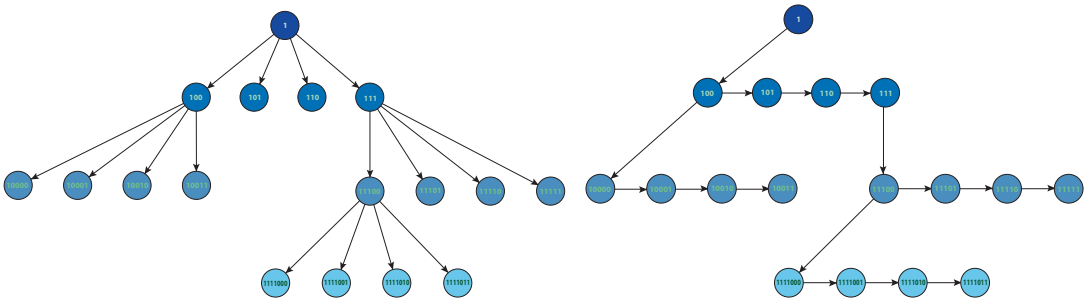
Figura 4.2: Exemplos em 2D e 3D para a geração da chave para uma 2^d–*tree*.

4.1.3 Função de *Hash*

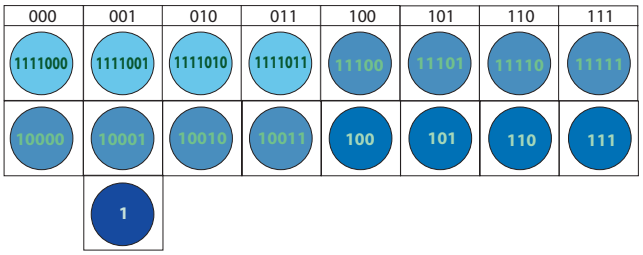
Já que a 2^d–*tree* é armazenada numa *hash table*, a função de *hash* tem um grande papel na eficiência das buscas e do armazenamento. O exemplo de função de *hash* eficiente, e que é usada nesta tese, é a que atribui para uma chave $k(n)$ de um nó n os seus h bits menos significantes:

$$h(n) = k(n) \mod 2^h.$$

Isso tende a homogeneizar a *hash table*, de forma que as entradas são regularmente distribuídas, mesmo para o caso de um 2^d–*tree* desbalanceada, tornando assim eficientes as buscas e armazenamento de um nó pela sua chave numa 2^d–*tree*. A Figura 4.3 mostra três maneiras diferentes de se armazenar uma *quadtree*: a Figura 4.3(a) armazena a *quadtree* da Figura 4.2(a) numa árvore com a implementação clássica; a Figura 4.3(b) armazena a quadtree da Figura 4.2(a) numa árvore com a implementação filho/irmão, visando economizar memória; e a Figura 4.3(c) armazena a *quadtree* da Figura 4.2(a) numa *hash table* (seção 2.5).



a) Representação Hierárquica b) Representação Filho/irmão



c) Representação Hashed

Figura 4.3: Três representações de *quadtree* da Figura 4.2(a). A *hash table* usa os três significantes bits da chave como função de *hash*: $k(n) \mod 2^3$.

4.1.4
Os quatro requisitos e os códigos de Morton

Agora que o método de códigos de Morton foi apresentado como um método de indexação para uma 2^{d}–*tree*, é preciso provar a sua eficiência, satisfazendo os quatro requisitos definidos na seção 3.1:

1. *Único*: Sejam dois nós n_1 e n_2 de uma 2^{d}–*tree* e suas respectivas chaves $k(n_1)$ e $k(n_2)$. Se $k(n_1) = k(n_2)$, então são iguais cada um dos d bits dos códigos direcionais dos filhos; logo, basta seguir a partir da raiz e verificar para cada d bits qual é o filho descendente percorrendo a árvore e encontrando o mesmo nó $n_1 = n_2$ no final.

2. *Hierárquico*: Seja $k(n)$ a chave de um nó n de uma 2^{d}–*tree*. A chave do seu nó pai n_{pai} é calculada em tempo constante utilizando apenas uma operação de deslocamento à direita de d bits: $k(n_{pai}) = k(n) \gg d$. A chave do seu i-ésimo filho n_i, com $i \in \{1, \ldots, d\}$ é calculada em tempo constante utilizando apenas uma operação de deslocamento à esquerda de d bits e uma operação de OU: $k(n_i) = (k(n) \ll d) \mid (i)$.

3. *Localizado*: O cálculo em tempo constante da chave $k(n_{adj})$ do nó adjacente n_{adj} de um nó n pertencente a uma 2^{d}–*tree* é feito utilizando operações em inteiros, como adição em códigos de Morton (seção 5.4), que utiliza $2d$ operações de OU, $2d$ operações de E e d operações de adição: $k(n_{adj}) = k(n) \oplus_{Morton} \Delta(k)$.

4. *Espacial*: Dada uma posição $p = (x_1, \ldots, x_d) \in [0,1]^d$ no hipercubo unitário do espaço d-dimensional e uma precisão m, a chave $k(p)$ da posição p de precisão m é calculada em tempo constante utilizando d multiplicações, d operações de comparação inteira, e uma operação de tempo constante de intercalação de d coordenadas que pode ser acelerada utilizando operações em inteiros como dilatação e contração (seção 5.1): $k(p) = (1b_m^d \ldots b_m^1 \ldots b_2^d \ldots b_2^1 \ldots b_1^d \ldots b_1^1)_2$, onde $(b_m^i, \ldots, b_1^i)_2 = \lfloor x_i \times 2^m \rfloor$.

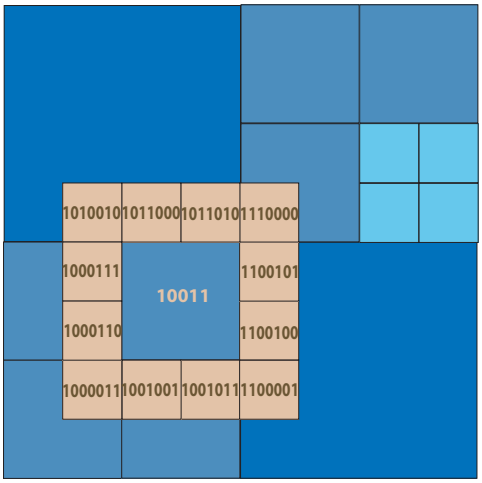


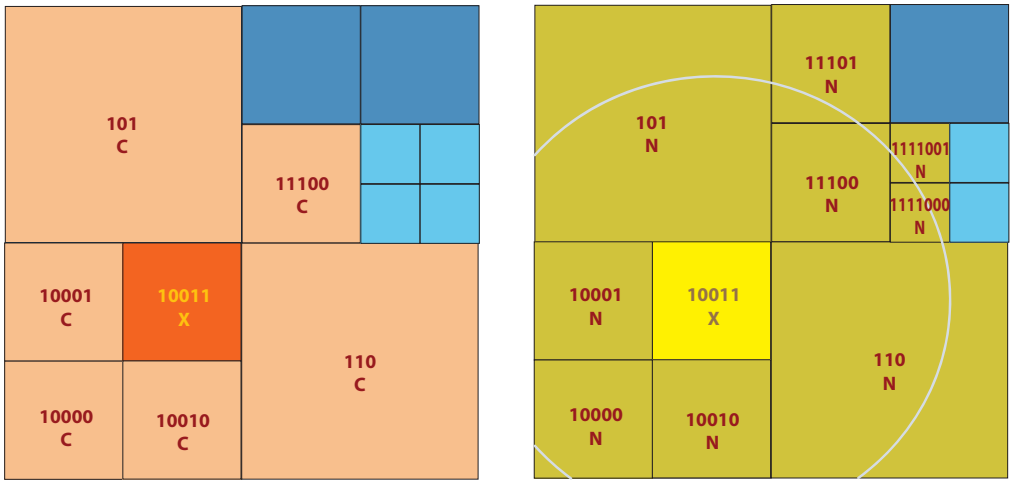
Figura 4.4: Chaves dos nós adjacentes de mesmo nível.

4.2 Implementando as Buscas

A Figura 4.4 é um exemplo de chaves dos nós adjacentes de mesmo nível que serão usadas nos procedimentos das busca por vizinhos e num raio da *quadtree* da Figura 4.2(a). O cálculo das chaves dos nós adjacentes será definido na seção 5.3.

4.2.1 Busca Direta

O algoritmo 1 descreve em detalhes o método otimizado do procedimento de busca direta apresentado na seção 3.2.1, para uma implementação de uma 2^d–*tree* aberta que armazena seus nós em uma *hash table*, e que utiliza o método de indexação de códigos de Morton apresentado acima.



a) Busca por vizinhos adjacentes b) Busca por vizinhos num raio

Figura 4.5: Procedimentos de busca da *quadtree* da Figura 4.2(a).

4.2.2
Busca de Vizinhos Adjacentes e em Torno de Um Raio

A Figura 4.5(a) é um exemplo do procedimento de busca por vizinhos da *quadtree* da Figura 4.2(a). O algoritmo 2 descreve em detalhes o método otimizado do procedimento de busca de vizinhos adjacentes apresentado na seção 3.2.2, para uma implementação de uma 2^d–*tree* aberta que armazena os seus nós numa *hash table*, e que utiliza o método de indexação de códigos de Morton apresentado acima.

Algorithm 1 busca(ponto p): busca a folha que contém p.

1: calcula a chave k_{max} de p no nível máximo
2: calcula a chave k de p no nível $\hat{l}$ usando k_{max} :
$k = k_{max}(p) \gg d \cdot (l_{max} - \hat{l})$
3: acessa o nó n que corresponde à chave k na *hash table*
// Caso 2: n não é uma folha
4: $count = 0$
5: **while** n existe na *hash table* **do**
6: $count = count + 1$
7: adiciona um no nível de k usando k_{max} :
$k = k_{max}(p) \gg d \cdot (l_{max} - \hat{l} + count)$
8: acessa a folha de n na *hash table* com k usando
9: **end while**
10: recupera o último acesso válido
// Caso 3: n está abaixo da folha
11: $count = 0$
12: **while** n não existe na *hash table* **do**
13: $count = count + 1$
14: diminui um no nível de k
15: acessa o pai de n na *hash table* com k
$k = k_{max}(p) \gg d \cdot (l_{max} - \hat{l} - count)$
16: **end while**
17: **return** n

Algorithm 2 adjacente(n): busca os nós vizinhos de n .

1: coloca no conjunto s os vizinhos adjacentes n_i de mesmo nível que n (seção 5.3)
2: **for all** nós n_i dentro de s **do**
3: na *hash table*, busque o nó n_i
// Case 1: n_i é uma folha
4: **if** n_i existe na *hash table* e é uma folha **then**
5: adicione n_i ao conjunto resultante r
// Case 2: n_i não é uma folha
6: **else if** n_i existe *hash table* **then**
7: insira em s os filhos de n_i adjacentes a n
// Case 3: n_i está abaixo da folha
8: **else**
9: chame busca(n_i) usando o nível $\min\{\hat{l}, l_n\}$
10: adicione o nó encontrado no conjunto resultante r
11: **end if**
12: **end for**
13: **return** o conjunto resultante r