Pontifícia Universidade Católica
do Rio de Janeiro

Rener Pereira de Castro

Otimização Estatística de Buscas para Estruturas Hierárquicas Espaciais

PUC-Rio - Certificação Digital Nº 0410421/CA

Tese de Doutorado

Tese apresentada ao Programa de Pós–graduação em Matemática Aplicada do Departamento de Matemática da PUC-Rio como requisito parcial para obtenção do título de Doutor em Matemática Aplicada

Orientador : Prof. Geovan Tavares dos Santos
Co–Orientador: Prof. Thomas Lewiner

Rio de Janeiro
Março de 2008

Pontifícia Universidade Católica
do Rio de Janeiro

Rener Pereira de Castro

Otimização Estatística de Buscas para Estruturas Hierárquicas Espaciais

Tese apresentada ao Programa de Pós–graduação em Matemática Aplicada do Departamento de Matemática do Centro Técnico Científico da PUC-Rio como requisito parcial para obtenção do título de Doutor em Matemática Aplicada. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Geovan Tavares dos Santos
Orientador
Departamento de Matemática — PUC-Rio

Prof. Thomas Lewiner
Co–Orientador
Departamento de Matemática — PUC-Rio

Prof. Jorge Stolfi
Instituto de Informática — UNICAMP

Prof. Claudio Esperança
COPPE — UFRJ

Prof. Luiz Henrique de Figueiredo
Laboratório Visgraf — IMPA

Prof. Waldemar Celes Filho
Departamento de Informática — PUC-Rio

Prof. Hélio Côrtes Vieira Lopes
Departamento de Matemática — PUC-Rio

Prof. Sinésio Pesco
Departamento de Matemática — PUC-Rio

Prof. José Eugênio Leal
Coordenador Setorial do Centro Técnico Científico — PUC-Rio

Rio de Janeiro, 7 de Março de 2008

PUC-Rio - Certificação Digital Nº 0410421/CA

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador.

Rener Pereira de Castro

Bacharel em Matemática pela Universidade Federal de Uberlândia–UFU (Uberlândia, Minas Gerais) (2002). Mestre em Matemática pelo Instituto Nacional de Matemática Pura e Aplicada–IMPA (Rio de Janeiro, Rio de Janeiro) (2004). Durante o doutorado foi membro do laboratório Matmidia do Departamento de Matemática da PUC–Rio, desenvolvendo projetos científicos patrocinados pela Petrobras. Tem experiência na área de Matemática Aplicada, com ênfase em Computação Gráfica e atualmente suas principais áreas de interesse são: estruturas de dados espaciais, aprendizado estatístico, modelagem geométrica.

PUC-Rio - Certificação Digital Nº 0410421/CA

Ficha Catalográfica

Castro, Rener Pereira de

Otimização Estatística de Buscas para Estruturas Hierárquicas Espaciais / Rener Pereira de Castro; orientador: Geovan Tavares dos Santos; co–orientador: Thomas Lewiner. — Rio de Janeiro : PUC-Rio, Departamento de Matemática, 2008.

v., 68 f: il. ; 29,7 cm

1. Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Matemática.

Inclui referências bibliográficas.

1. Matemática – Tese. 2. 2^d–*tree*. 3. Estrutura de Dados Espaciais. 4. *Hash Table*. 5. Modelagem Geométrica. 6. Geometria Computacional. 7. Matemática Discreta. I. Santos, Geovan Tavares dos. II. Lewiner, Thomas. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Matemática. IV. Título.

CDD: 510

Agradecimentos

PUC-Rio - Certificação Digital Nº 0410421/CA

Deus, aos meus pais, irmãos, família e que souberam lidar com minha ausência e que sem o apoio deles esse sonho não seria realizado.

À Daniela por sua compreensão, sua paciência, seu amor e por me proporcionar conforto e paz indispensáveis para a realização desse trabalho.

Ao meu orientador Professor Geovan Tavares pelo apoio durante toda a realização desse trabalho.

Ao meu co-orientador Thomas Lewiner pelo incansável incentivo e motivação para a conclusão desse trabalho.

Aos professores Hélio Lopes, Sinésio Pesco e Marcos Craizer os quais compartilharam grandes idéias durante esse tempo.

Aos meus amigos Etereldes (Manjator), Fabiano (Véia Murrinha), Thiago (MochI) e Afonso (Mamba) por proporcionarem boas gargalhadas e muitas descontrações e que ao longo desses anos eles se tornaram a minha outra família.

Aos meus colegas da PUC–Rio, que me fizeram adorar este lugar.

Ao pessoal do Departamento de Matemática, pela ajuda de todos os dias, em particular, à Creuza e ao Otávio.

À CAPES e à PUC–Rio, pelos auxílios concedidos, sem os quais este trabalho não poderia ter sido realizado.

Resumo

Castro, Rener Pereira de; Santos, Geovan Tavares dos; Lewiner, Thomas. **Otimização Estatística de Buscas para Estruturas Hierárquicas Espaciais**. Rio de Janeiro, 2008. 68p. Tese de Doutorado — Departamento de Matemática, Pontifícia Universidade Católica do Rio de Janeiro.

Este trabalho surgiu da seguinte observação: os clássicos algoritmos de busca em 2^d–*tree* começam da raiz para acessar dados armazenados nas folhas. Entretanto, como as folhas são os nós mais distantes da raiz, por que começar as buscas pela raiz? Com representações clássicas de 2^d–*trees*, não existe outra forma de acessar uma folha. Existem 2^d–*trees*, porém, que permitem acessar em tempo constante qualquer nó, dado sua posição e seu nível. Para o algoritmo de busca, a posição é conhecida, mas o nível não. Para estimar o nível de um nó qualquer, um método de otimização estatística do custo médio das buscas é proposto. Como os piores custos de busca são obtidos quando se começa da raiz, este método melhora ambos: o consumo de memória pelo uso de 2^d–*trees* que permitem acessar em tempo constante qualquer nó, e o tempo de execução através da otimização proposta.

PUC-Rio - Certificação Digital Nº 0410421/CA

Palavras–chave

2^d–*tree*. Estrutura de Dados Espaciais. *Hash Table*. Modelagem Geométrica. Geometria Computacional. Matemática Discreta.

Abstract

Castro, Rener Pereira de; Santos, Geovan Tavares dos; Lewiner, Thomas. **Statistical Optimization of Spatial Hierarchical Structures Searchs**. Rio de Janeiro, 2008. 68p. PhD Thesis — Department of Mathematics, Pontifícia Universidade Católica do Rio de Janeiro.

This work emerged from the following observation: usual search procedures for 2^d–*trees* start from the root to retrieve the data stored at the leaves. But since the leaves are the farthest nodes to the root, why start from the root? With usual 2^d–*trees* representations, there is no other way to access a leaf. However, there exist 2^d–*trees* which allow accessing any node in constant time, given its position in space and its depth in the 2^d–*tree*. Search procedures take the position as an input, but the depth remains unknown. To estimate the depth of an arbitrary node a statistical optimization of the average cost for the search procedures is introduced. Since the highest costs of these algorithms are obtained when starting from the root, this method improves on both, the memory footprint by the use of 2^d–*trees* which allow accessing any node in constant time, and execution time through the proposed optimization.

PUC-Rio - Certificação Digital Nº 0410421/CA

Keywords

2^d–*tree*. Data Structure. Hash Table. Geometry Modeling. Computational Geometry. Discrete Mathematics.

Sumário

1 Introdução **9**

2 Estruturas Hierárquicas **13**
2.1 Decomposição 13
2.2 Representação 17
2.3 Buscas 20
2.4 Indexação 20
2.5 Armazenamento 28

3 Otimização de Buscas **36**
3.1 Requisitos para uma $\mathbf{2^d}$–$\boldsymbol{Tree}$ Aberta 36
3.2 Busca com $\mathbf{2^d}$–$\boldsymbol{Tree}$ Abertas 37
3.3 Modelagem das Buscas e Otimização Estatística 40

4 Implementando Eficientemente uma $\mathbf{2^d}$–$\boldsymbol{Tree}$ Aberta **44**
4.1 Códigos de Morton 44
4.2 Implementando as Buscas 48

5 Operações Eficientes em Inteiros **51**
5.1 Dilatação e Contração 51
5.2 Adição em Inteiros Dilatados 56
5.3 Adição em Códigos de Morton 57
5.4 Chaves Adjacentes 58

6 Resultados **60**
6.1 Modelo de Teste 60
6.2 Modelo de Comparação 60
6.3 Modelo de Buscas Diretas 61
6.4 Modelo de Objetos Geométricos 61

7 Conclusão e Trabalhos Futuros **65**

Referências Bibliográficas **66**

PUC-Rio - Certificação Digital Nº 0410421/CA

PUC-Rio - Certificação Digital Nº 0410421/CA

Os problemas significativos que enfrentamos não podem ser resolvidos no mesmo nível de pensamento em que estávamos quando os criamos.

Albert Einstein.

1
Introdução

A geometria local dos objetos discretos é geralmente definida considerando a geometria dos elementos próximos. Portanto, o processamento geométrico depende fortemente de procedimentos de busca, identificando proximidades entre objetos. Algumas representações de objetos discretos, tais como reticulados regulares ou malhas, representam explicitamente as respostas para esses procedimentos. Nesse sentido, eles fornecem um extremo do balanceamento da relação tempo/memória: tempo mínimo e memória máxima. No outro extremo estão as representações de objetos como nuvens de pontos, que não armazenam nenhuma relação de vizinhança.

Em ambas as situações, estruturas de localização em malhas ou em nuvem de pontos independem da representação dos objetos em si, permitindo calcular propriedades locais de uma maneira mais eficiente. Exemplos dessas aplicações: cálculo de normais e curvaturas de nuvens de pontos; triangulação de superfícies através de algoritmos locais, tais como técnicas de mínimos quadrados móveis ou algoritmos globais usando triangulações de Delaunay; teste de colisão em animação; representação de interação física em simulações; e geração de uma representação em multi-resolução de objetos discretos.

As estruturas clássicas de busca são na sua maioria hierárquicas, por reduzirem localizações para uma complexidade média logarítmica. Elas incluem 2^d–*trees*, kd-*trees*, multi-grades, triangulação hierárquica e partições binárias do espaço. Esta tese propõe uma melhora nos procedimentos de busca para 2^d–*trees*, baseada numa modelagem estatística na distribuição dos nós, trabalhando numa representação otimizada do espaço, de tal forma a fornecer uma melhora no tempo de execução desses procedimentos.

Metodologia

Este trabalho foi desenvolvido a partir da seguinte observação: procedimentos usuais de busca para 2^d–*trees* começam da raiz para acessar um dado armazenado nas folhas, porém, já que as folhas são os nós mais distantes da raiz, por que começar da raiz? Com a representação usual das 2^d–*trees*, não

PUC-Rio - Certificação Digital Nº 0410421/CA

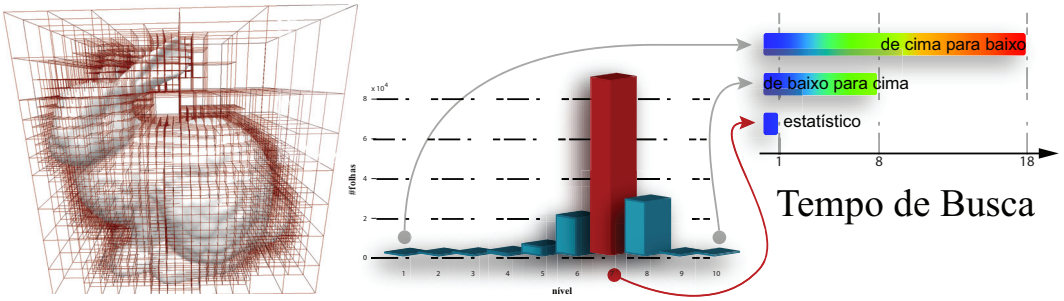


Figura 1.1: Já que as folhas são os nós mais distantes da raiz, porque começar da raiz ao procurar uma folha? Um nível ótimo inicial reduz consideravelmente os custos das buscas: (esquerda) um refinamento de *octree* que separa os vértices do modelo *Stanford bunny*; (meio) o histograma do número de folhas da *octree* por nível; (direita) custo de busca de uma folha começando de diferentes níveis.

existe outra maneira de acessar as folhas, embora uma 2^d–*tree* permita acesso constante aos seus nós, dada a sua posição no espaço e a sua profundidade. Procedimentos de busca recebem a posição como entrada, mas a profundidade é geralmente desconhecida. Este trabalho propõe estimar a profundidade de um nó arbitrário, através de uma otimização estatística do custo médio dos procedimentos de busca. A Figura 1.1 mostra que um nível ótimo inicial reduz consideravelmente os custos das buscas. Como o custo maior desses algoritmos ocorre quando se começa da raiz, esse método sempre reduz ao mesmo tempo a memória e o tempo de execução, através da otimização proposta.

Desenvolvimento Histórico das 2^d–*Tree*

O princípio da decomposição hierárquica tem sua origem incerta: foi visto primeiramente como uma maneira de agregar blocos de zeros em matrizes esparsas, mas se desenvolveu para aplicações em dados geométricos. Hoare (11) indicou a Dijkstra o atributo da decomposição de um nível de uma matriz em blocos de zeros. Morton (28) usou o significado da decomposição hierárquica para a indexação de dados geográficos. Warnock (32), em sua tese de doutorado, que marcou a literatura da computação gráfica, descreve a implementação de um algoritmo de eliminação de linha e superfície escondida usando uma decomposição recursiva da área de uma imagem. A área de imagem era repetidamente subdividida em retângulos que são sucessivamente menores, procurando assim por áreas simples para exibir. Klinger (25) e Klinger & Dyer (26), aplicaram as idéias de Warnock em reconhecimento de padrões e em processamento de imagens, enquanto Hunter (22) usou-as em animação.

O projeto Robô SRI (29) usou uma decomposição de três níveis do

PUC-Rio - Certificação Digital Nº 0410421/CA

espaço para representar um mapa no mundo dos robôs. A monografia de Knuth, intitulada *Art of Computer Programming* (27), cobre muitos tipos de programação de algoritmos e suas análises. No volume 3, mais precisamente, possui muitas estruturas de dados para busca. Dentre elas, existe a *binary search tree*, onde a maioria das estruturas de dados de busca conhecidas foram baseadas. Finkel e Bentley (16) casaram a *binary search tree* com o método da malha fixa, dando origem a uma composição de diretórios na forma de árvore, com células de tamanhos não-uniformes, e a nomearam de *quadtree*. Logo depois surge a *kd-tree* no trabalho de Bentley em (4), uma estrutura de dados multidimensional para busca e para um melhor balanceamento da árvore na qual é armazenada. Friedman, Bentley e Finkel em (13) criaram a *kd-tree adaptativa*, que faz uma partição adaptativa do espaço levando em conta a média dos pontos: cada nó é representado por um escalonamento de um semi-espaço.

Posteriormente, Fuchs, Kedem e Naylor em (9) criaram o *particionamento binário do espaço*, que é uma generalização da *kd-tree*, onde cada nó é representado por uma equação linear de um semi-espaço. Depois surgiram as chamadas *quadtrees* sem ponteiros, sendo uma delas chamada *df-expressions* (denotando *depth-first*), discutida por Kawaguch e Endo em (23) e por Kawaguch, Endo e Yokota em (24), representada na forma de uma lista de letras na ordem de percorrimento. As letras identificam se um nó da *quadtree* é um pai, uma folha vazia ou uma folha que contém algum conteúdo. Irene Gargantini em (7, 14) introduziu outra versão de *quadtree* sem ponteiros, chamada de *quadtree linear*, formada apenas da coleção dos nós folhas. Cada nó é representado por um par de números, sendo um o nível do nó na *quadtree*, e o outro chamado *código local*, que é formado pela concatenação de 4 dígitos, que correspondem ao código direcional da posição do nó ao longo do seu caminho até a raiz da *quadtree*.

A variação em três dimensões da *quadtree*, chamada de *oct-tree* ou *octree*, foi desenvolvida independentemente por vários pesquisadores. Reddy e Rubin em (31) propuseram a *octree* como uma de três representações de objetos sólidos. A segunda representação é a decomposição em paralelepípedos retangulares por planos perpendiculares aos eixos x, y e z. A terceira representação quebra o objeto em paralelepípedos retangulares, que não são necessariamente alinhados com os eixos coordenados, possuindo tamanhos e orientações arbitrárias. Cada paralelepípedo é subdividido recursivamente em outros novos paralelepípedos, no sistema coordenado definido pelo paralelepípedo inicial. Yau e Srihari em (33) estenderam a *octree* para dimensões quaisquer, num processo de desenvolvimento de algoritmos para manipular imagens médicas.

PUC-Rio - Certificação Digital Nº 0410421/CA

Hoje em dia as 2^d–*trees* são usadas em diversas áreas de pesquisa, sendo seu estudo de grande importância para o desenvolvimento da ciência.

Contribuição

Esta tese propõe aperfeiçoar os procedimentos de busca de uma 2^d–*tree* aberta, exigindo apenas quatro requisitos para o seu armazenamento e sua indexação. Para um armazenamento numa *hash table* e uma indexação baseada nos códigos de Morton, serão definidos métodos e algoritmos que satisfazem esse requisitos. Comparado com a representação clássica de uma 2^d–*tree*, o novo método reduz, tanto o tempo de execução por um fator médio de 4 vezes, como o consumo de memória utilizada, em média, metade da memória exigida pela representação clássica. No caso de uma 2^d–*tree* aberta, o método mantém o mesmo consumo de memória, mas melhora, em média, 15 vezes o tempo de execução. Embora o método aqui proposto se baseie numa modelagem estatística simples, melhora aqueles anteriores, mesmo nos piores casos. Para uma 2^d–*tree* aberta, o acesso do nó pela posição espacial e seu nível possui na média um tempo de execução *constante*. A idéia principal para acessar uma folha numa dada posição é, através de uma estimativa estatística, procurar por seus vizinhos. Parte desse trabalho foi publicado em (6), que se resume apenas ao caso tridimensional.

PUC-Rio - Certificação Digital Nº 0410421/CA

2
Estruturas Hierárquicas

Para localizar um objeto de uma maneira eficiente, é preciso *decompor* o espaço em pequenas partes, utilizando uma de suas *representações*, *indexar* cada parte de uma única forma e *armazená-la* numa estrutura apropriada. Por exemplo, o sistema de endereços dos correios decompõe uma cidade em blocos, representados pelos bairros, indexados pelo CEP de forma única, sendo essas informações armazenadas no endereço de cada carta.

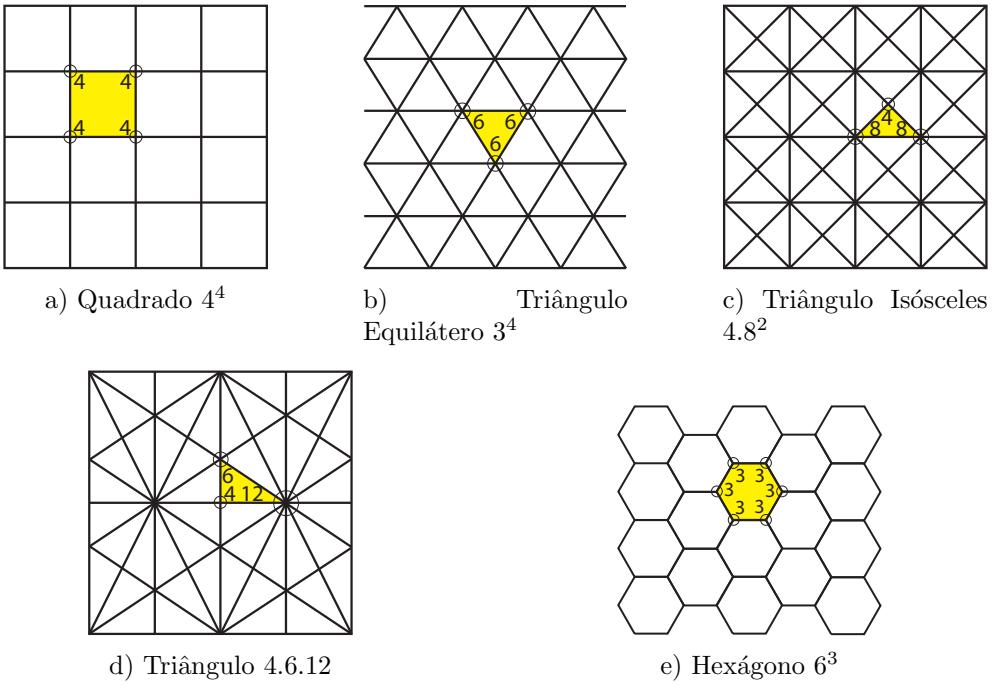
Este capítulo descreve algumas estruturas de decomposição, representação, indexação e de armazenamento espacial. Seu principal objetivo é dar uma visão geral simplificada das diferentes estruturas espaciais existentes. Para uma descrição mais detalhada, consultar os livros clássicos de Samet (17, 18) sobre estruturas de dados e suas aplicações.

PUC-Rio - Certificação Digital Nº 0410421/CA

2.1
Decomposição

Existem infinitas maneiras de decompor um espaço. Em particular, uma maneira simples de decompor o plano é escolher uma pequena peça, e a partir dela tentar preencher todo o plano, encaixando e colando infinitas peças que possuem a mesma forma que a peça original, sem sobrepor e sem deixar buracos. Esse tipo de decomposição é chamado de *ladrilhamento*, e a pequena peça inicialmente escolhida para decompor o plano é chamada de *ladrilho*. Como os ladrilhos mais comuns são os que possuem formas poligonais, considera-se sempre que um ladrilho é um polígono. O *grau* do vértice de um ladrilho é o número de arestas que se conectam a ele dentro do ladrilhamento. Os ladrilhos são descritos pelo uso de uma notação, baseada no grau de cada vértice, da seguinte forma: sejam $g_0, g_1, \ldots, g_n$ os n graus na ordem crescente dos n vértices de um ladrilho; então sua notação é definida como sendo $[g_0.g_1.\ \ldots\ .g_n]$. Se para algum $i \in \{0, \ldots, n\}$, tem-se $g_i = g_{i+1} = \cdots = g_{i+m}$; então sua notação também pode ser escrita como sendo $[g_0.g_1.\ \ldots\ .g_i^m.g_{i+m+1}.\ \ldots\ .g_n]$. A Figura 2.1, por exemplo, ilustra alguns ladrilhamentos, em particular o ladrilhamento da Figura 2.1(c), que tem a forma de um triângulo isósceles, onde o primeiro vértice tem grau 4, enquanto

os outros dois vértices têm grau oito cada um; então nesse caso sua notação é dada por $[4.8^2]$.



a) Quadrado 4^4 b) Triângulo Equilátero 3^4 c) Triângulo Isósceles 4.8^2

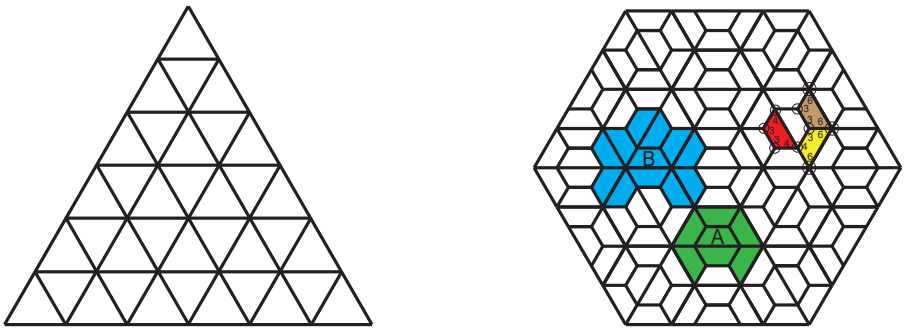
d) Triângulo 4.6.12 e) Hexágono 6^3

Figura 2.1: Exemplos de ladrilhamento do plano.

Um ladrilhamento em que o polígono formado da união de um ladrilho com os ladrilhos com os quais ele compartilha uma aresta ou um vértice, é o mesmo polígono para qualquer ladrilho. Nesse caso esse ladrilhamento é dito ser *isoedral*. Por exemplo, considere os dois ladrilhamentos da Figura 2.2. Um consiste de triângulos equiláteros (Figura 2.2(a)) e é descrito por $[6^3]$. O outro consiste de trapézios (Figura 2.2(b)) e é descrito por $[3^2.4^2]$, por $[3^2.6^2]$, ou ainda por $[3.4.6^2]$. É fácil verificar que os triângulos eqüiláteros são isoedrais, pois, para qualquer ladrilho escolhido, o polígono formado pela união desse ladrilho com os ladrilhos com os quais ele compartilha um vértice ou uma aresta, sempre é um hexágono. Isso já não acontece com os trapézios, como pode ser visto pelos ladrilhos A e B.

Um ladrilhamento é dito *regular* se o seu ladrilho é um polígono regular. O conjunto de ladrilhos formado por uma seqüência de ladrilhos de mesma forma, contidos inteiramente uns nos outros, é chamado de *ladrilho hierárquico*, cuja forma não precisa ser necessariamente igual a do ladrilho inicial. O *nível* de um ladrilho hierárquico é a quantidade de ladrilhos existente na seqüência. Um ladrilho hierárquico de nível $k > 0$ é dito ser *similar*, se ele possui a mesma forma de seu ladrilho hierárquico de nível 0. Para que um ladrilho hierárquico possua um nível infinito, é necessário que ele seja similar. Um *ladrilhamento hierárquico* é um ladrilhamento formado de ladrilhos hierárquicos. Um *ladrilha-*

PUC-Rio - Certificação Digital Nº 0410421/CA



a) Isoedral b) Não isoedral

Figura 2.2: Ladrilhamento isoedral e não-isoedral.

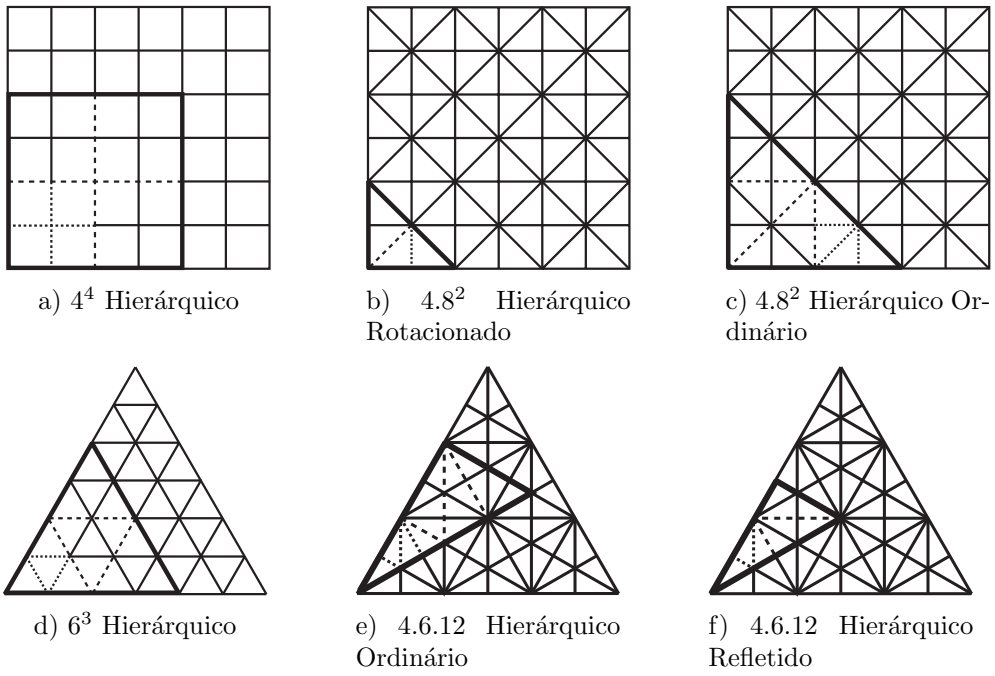
mento hierárquico similar é um ladrilhamento hierárquico formado de ladrilhos hierárquicos similares.

O critério mais importante a se discutir é a distinção entre hierarquias *limitadas* e *ilimitadas* de ladrilhamento. Um ladrilhamento hierárquico é dito *ilimitado* se ele contém um ladrilho hierárquico de nível infinito. Um ladrilhamento hierárquico é *limitado* se ele não é similar. O ladrilhamento hexagonal $[3^6]$ da Figura 2.1(e) é limitado, pois todos os seus ladrilhos hierárquicos possuem nível 0. Bell *et al.* (3), afirmam que apenas quatro ladrilhamentos são ilimitados, os quais estão ilustrados na Figura 2.3. Deles $[4^4]$, consistindo do ladrilho quadrado, e o $[6^3]$, consistindo de ladrilho triângulo equilátero, são os bem conhecidos ladrilhamentos regulares (2). Seus ladrilhos hierárquicos são a Figura 2.3(a) e a Figura 2.3(d), respectivamente. Os ladrilhamentos $[4^4]$ e $[6^3]$ podem gerar um número infinito de ladrilhos hierárquicos diferentes, onde cada ladrilho hierárquico de nível n consiste de n^2 ladrilhos.

Os demais ladrilhamentos triangulares não-regulares e ilimitados $[4.8^2]$, correspondentes à Figura 2.1(c), e o [4.6.12], correspondentes à Figura 2.1(d), são junções dos centróides dos ladrilhos de $[4^4]$ e $[6^3]$, respectivamente, de ambos os seus vértices e pontos médios de suas arestas. O ladrilhamento $[4.8^2]$ possui um ladrilhamento ilimitado hierárquico ordinário, que corresponde à Figura 2.3(c), e um ladrilhamento ilimitado hierárquico rotacionado, que corresponde à Figura 2.3(b), exigindo uma rotação de 135° graus entre níveis. Já o ladrilhamento [4.6.12] tem um ladrilhamento ilimitado hierárquico ordinário, que corresponde à Figura 2.3(e), e um ladrilhamento ilimitado hierárquico refletido, que corresponde à Figura 2.3(f), que exige uma reflexão do ladrilho básico entre níveis.

No caso dos tipos de hierarquias $[4.8^2]$ e [4.6.12], o ladrilhamento não é similar sem uma rotação ou uma reflexão, quando a hierarquia não é ordinária. Isso pode ser visto observando-se o uso de pontos na Figura 2.3 para delimitar o ladrilho no seu ladrilho hierárquico. Do mesmo modo, linhas tracejadas são

PUC-Rio - Certificação Digital Nº 0410421/CA



a) 4^4 Hierárquico

b) 4.8^2 Hierárquico Rotacionado

c) 4.8^2 Hierárquico Ordinário

d) 6^3 Hierárquico

e) 4.6.12 Hierárquico Ordinário

f) 4.6.12 Hierárquico Refletido

Figura 2.3: Ladrilhamentos ilimitados, os pontos são para delimitar o ladrilho no seu ladrilho hierárquico.

PUC-Rio - Certificação Digital Nº 0410421/CA

usadas para delimitar os componentes dos ladrilhos de segundo nível. Para as hierarquias ordinárias de $[4.8^2]$ e de [4.6.12], cada ladrilho hierárquico de nível n consiste de n^2 ladrilhos. Para a hierarquia reflexiva de [4.6.12], cada ladrilho hierárquico de nível n consiste de $3n^2$ ladrilhos, enquanto para a hierarquia rotacionada de $[4.8^2]$, cada ladrilho hierárquico de nível n consiste de $2n^2$ ladrilhos.

Ladrilhos são ditos *adjacentes*, quando compartilham ao menos um vértice ou uma aresta. Dois ladrilhos são ditos *vizinhos* se eles são adjacentes. Um ladrilhamento é *uniformemente adjacente* se as distâncias entre o centróide de um ladrilho e os centróides de todos os seus vizinhos são iguais. O *número de adjacência* de um ladrilhamento é o número de distâncias diferentes de centróides internos entre qualquer ladrilho e seus vizinhos. No caso de $[4^4]$, existem apenas duas distâncias adjacentes; por sua vez $[6^3]$ tem três distâncias adjacentes. Um ladrilhamento é dito ter *orientação uniforme*, se os ladrilhos podem ser transformados uns nos outros por uma translação do plano, que não envolve rotação e nem reflexão. O ladrilhamento $[4^4]$ possui orientação uniforme; já o ladrilhamento $[6^3]$, não.

Para representar dados no plano Euclidiano, os ladrilhos ilimitados são preferenciais. Para uma decomposição única do plano, os ladrilhamentos $[4.8^2]$ e [4.6.12] não são aplicados. Comparando os ladrilhamentos quadrado $[4^4]$ e triangular $[6^3]$, vemos que eles se diferem em termos de *adjacência* e *orientação*. Então, o ladrilhamento $[4^4]$ é mais adequado que o ladrilhamento $[6^3]$, quando

a orientação uniforme e a distancia de adjacência mínima são propriedades consideradas importantes para um ladrilhamento. Além disso, o ladrilhamento $[4^4]$, devido à sua geometria simples, é mais usado na prática.

2.2 Representação

Métodos de decomposição geram diferentes maneiras de se representar um espaço. Normalmente a representação é escolhida para uma tarefa específica, e a escolha é altamente influenciada pelo tipo de operação a ser feita no dado. Para representar eficientemente dados multidimensionais, é preciso agregar dados próximos e similares. Uma maneira eficiente é efetuar *subdivisões* no espaço separando-o em regiões de forma hierárquica ou não. O balanço entre a eficiência do agrupamento de uma subdivisão *versus* quantidade de informação a ser armazenada é discutido nesta seção.

2.2.1 Representação Não-hierárquica

A maneira mais simples de se representar um dado multidimensional é por um *reticulado regular*, que corresponde à decomposição do espaço obtida a partir do ladrilhamento $[4^4]$. Um reticulado regular subdivide igualmente o espaço em subespaços que possuem a mesma forma e o mesmo tamanho. Essa representação é um extremo da relação subdivisão eficiente *versus* informação armazenada, pois não sendo adaptativa, não garante uma boa subdivisão, mas, por outro lado, requer quase nenhuma informação além do dado.

2.2.2 Representação Hierárquica

Uma *subdivisão hierárquica* do espaço é uma seqüência de regiões $(\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_m)$ tal que a seqüência é encaixante: $\mathcal{R}_i \supset \mathcal{R}_{i+1}\ \forall i \in 0, \ldots, m-1$. Isto significa que toda região $\mathcal{R}_{i+1}$ da seqüência está contida na região antecessora $\mathcal{R}_i\ \forall i = 0, \ldots, m-1$. Por exemplo, um ladrilho hierárquico (seção 2.1) é um tipo de subdivisão hierárquica. A Figura 2.4 é um exemplo de uma subdivisão hierárquica composta de uma seqüência de cinco regiões quadradas.

Cada região da seqüência é chamada de *nó*. A região zero da seqüência, que é a região original antes de ser subdividida, é chamada de *nó raiz*, ou simplesmente *raiz*. A última região da seqüência, aquela que está contida em todas as outras, mas que não contém nenhuma outra, é chamada de *nó folha*, ou simplesmente *folha*.

PUC-Rio - Certificação Digital Nº 0410421/CA

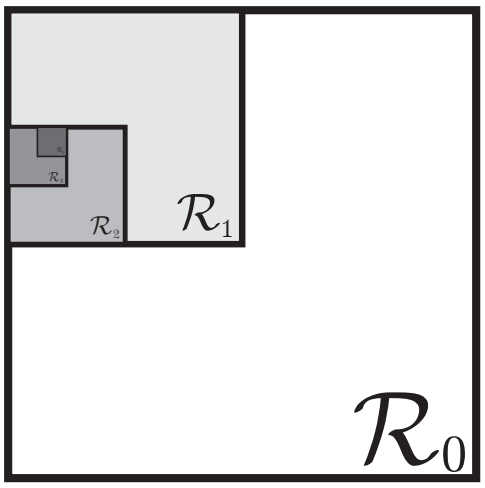


Figura 2.4: Uma subdivisão hierárquica composta de uma seqüência de cinco regiões quadradas.

O nó que corresponde à posição *zero* da seqüência representa o *nível zero* da subdivisão hierárquica, o nó que corresponde à posição *um* da seqüência representa o *nível um* da subdivisão hierárquica, o nó que corresponde à posição *dois* da seqüência representa o *nível dois* da subdivisão hierárquica, e assim por diante. O *nível máximo*, ou apenas *nível* de uma subdivisão hierárquica, é a quantidade de nós da seqüência.

Existem relações que dependem de uma referência: seja S_i com $i \in \{0, \ldots, m\}$ algum nó da seqüência, então o nó S_j com $j \in \{0, \ldots, i-1\}$ pertence à *ascendência* do nó S_i. Por outro lado, o nó S_k com $k \in \{i+1, \ldots, m\}$ pertence à *descendência* do nó S_i. O nó S_{i-1} é chamado de *nó pai*, ou simplesmente *pai*, em relação ao nó S_i $\forall i = 1, \ldots, m$. Já o nó S_{i+1} é chamado de *nó filho*, ou simplesmente *filho*, em relação ao nó S_i $\forall i = 0, \ldots, m-1$.

Uma *representação hierárquica* é uma representação constituída de subdivisões hierárquicas. O *nível* de uma representação hierárquica é o maior dos níveis máximos de suas subdivisões hierárquicas. Um ladrilhamento hierárquico (seção 2.1), por exemplo, é um tipo de representação hierárquica.

2.2.3 Particionamento binário do espaço

Particionamento binário do espaço ou *bsp* é uma representação hierárquica do espaço d-dimensional, em que as suas subdivisões hierárquicas são constituídas de seqüências de subconjuntos convexos subdivididos através de hiperplanos. Existem muitas maneiras de escolher os hiperplanos de subdivisão, por isso, essa representação hierárquica necessita de muita informação armazenada, pois cada nó deve conter informações do seu hiperplano de sub-

PUC-Rio - Certificação Digital Nº 0410421/CA

divisão, que nesse caso possui posição e direção arbitrárias. A representação *bsp* é o outro extremo da relação subdivisão eficiente *versus* informação armazenada, pois subdivide o espaço da maneira mais adaptável possível, mas exige um grande armazenamento de informação.

Para uma nuvem de pontos, uma maneira eficiente de se construir uma *bsp*, para agrupar pontos similares, encontra-se em (5): em cada passo da subdivisão o hiperplano de subdivisão possui a posição que corresponde à mediana dos pontos pertencentes à região a ser subdividida, e a direção do hiperplano é a direção da primeira componente principal dos mesmos pontos. Essa escolha faz com que a *bsp* tenha aproximadamente o mesmo número de pontos por nó folha.

2.2.4 KD-*Tree*

A representação hierárquica *kd-tree* é um caso particular da *bsp*, onde são usados apenas hiperplanos de subdivisão que são perpendiculares aos eixos coordenados, com o objetivo de diminuir a quantidade de informação armazenada. Existem muitas maneiras de escolher os hiperplanos de subdivisão, mas necessariamente, devem ser perpendiculares aos eixos, exigindo menos informação a ser armazenada, pois cada nó deve conter informações do seu hiperplano de subdivisão, que nesse caso possui uma posição arbitrária, porém, sua direção é conhecida, devendo ser perpendicular a um dos eixos coordenados. A *kd-tree* é também uma representação eficiente, um pouco menos que a *bsp*, pois limita a maneira de subdividir o espaço com o objetivo de reduzir o armazenamento de informação. É um bom equilíbrio da relação subdivisão eficiente *versus* informação armazenada.

Para uma nuvem de pontos, a maneira clássica de se construir uma *kd-tree* é: em cada passo da subdivisão, a posição do hiperplano de subdivisão é a mediana dos pontos pertencentes à região a ser subdividida, e as escolhas dos eixos coordenados para alinhar o hiperplano devem formar um ciclo ordenado. Essa escolha faz com que a *kd-tree* tenha aproximadamente o mesmo número de pontos por nó folha. A escolha do ponto mediano é necessária apenas para o balanceamento, e não uma exigência da *kd-tree*.

2.2.5 2^{d}–*Tree*

A 2^{d}–*tree* é uma representação hierárquica do espaço d-dimensional, em que as suas subdivisões hierárquicas são constituídas de seqüências de subespaços $(\mathcal{Q}_0, \mathcal{Q}_1, \ldots, \mathcal{Q}_m)$ tais que todo subespaço $\mathcal{Q}_{i+1}$ da seqüência tem

PUC-Rio - Certificação Digital Nº 0410421/CA

seu volume menor numa razão de 2^d do que o volume do seu subespaço antecessor $\mathcal{Q}_i$:

$$V(\mathcal{Q}_{i+1}) = \frac{V(\mathcal{Q}_i)}{2^d} \quad \forall i = 0, \ldots, m-1.$$

As posições dos subespaços são fixas e independem do espaço, exigindo pouca informação a ser armazenada.

2.3 Buscas

As buscas têm o objetivo de acessar os dados similares agregados pelas representações hierárquicas. Existem vários tipos de busca para as estruturas hierárquicas. Na sua maioria, são procedimentos que procuram informações de proximidade, retornando o endereço de armazenamento daquele pedido. Em alguns casos, o procedimento procura vários endereços de armazenamento que estão próximos daquele dado de entrada.

A forma clássica de efetuar esses procedimentos de busca numa estrutura hierárquica se baseia, na sua maioria, em percorrer a hierarquia, começando pela raiz, e perguntando a cada nó se tal procedimento de busca procede na sua descendência. Para o caso positivo, é perguntado para os nós filhos daquele nó, se tal procedimento de busca procede na sua descendência, e assim por diante, até encontrar um nó folha que o procedimento de busca seja positivo. O nó folha é o fim da hierarquia, e onde geralmente estão as informações buscadas.

2.4 Indexação

A indexação de uma representação é uma enumeração de dados, associando os similares a um mesmo índice. Um fator importante na escolha da representação de dados multidimensionais está no fato de que a estrutura resultante minimiza o número de índices que não correspondem a nenhum dado. Isto tem um impacto crítico no desempenho de operações como armazenamento, atualização e buscas. Ter uma estrutura, porém, que satisfaça essas exigências nem sempre é possível. Como as técnicas para dados *uni*dimensionais são mais conhecidas, dados multidimensionais devem ser primeiramente mapeados em dados *uni*dimensionais, e então representá-los usando uma estrutura adequada. Como num espaço unidimensional existe uma *ordem* natural dos elementos, deve-se definir uma *ordem* coerente no espaço multidimensional, para que o mapeamento esteja bem definido preservando localização espacial. A primeira subseção descreve alguns tipos mais conhecidos de ordenações do plano.

No caso de uma representação hierárquica, em que as subdivisões hierárquicas geram regiões (nós) similares, é possível definir um único índice

PUC-Rio - Certificação Digital Nº 0410421/CA

para cada nó, chamado de *código local*, que o identifica dentro da estrutura hierárquica. Assim é possível buscar um nó apenas pelo seu código local e não pela hierarquia, fornecendo acesso direto ao nó. De uma maneira informal, permite-se chamar um nó diretamente, sem ter que referenciar a sua família. A segunda subseção descreve alguns tipos mais conhecidos de códigos locais para uma *quadtree* ($2^2 - tree$).

2.4.1 Tipos de Ordenação

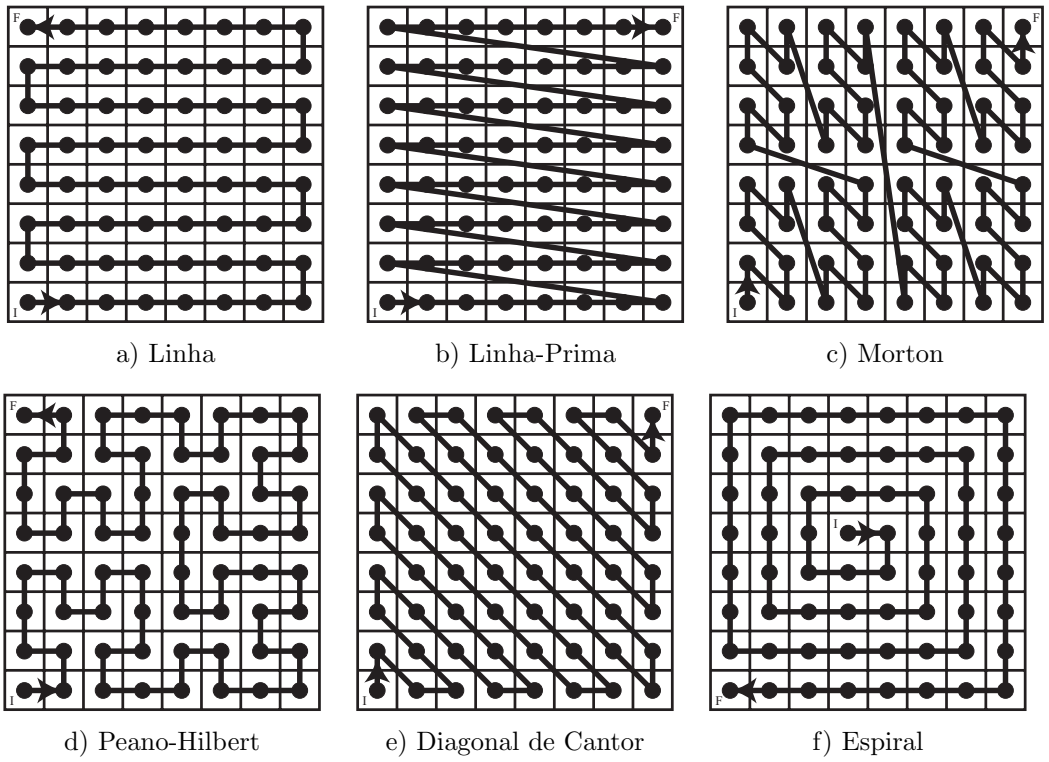


Figura 2.5: Exemplos de ordenação.

O primeiro propósito dos métodos de ordenação é o de aperfeiçoar o armazenamento, e processar seqüências de dados multidimensionais, mapeando-as em uma única dimensão. Goodchild e Grandfield (8) discutem vários métodos de ordenações espaciais, sendo que alguns estão ilustrados na Figura 2.5 e nas tabelas abaixo. Cada um tem características diferentes. A ordem *linha* (Figura 2.5(a)), também conhecida como *raster-scan*, e a ordem *linha-prima* (Figura 2.5(b)) são similares, bem como a ordem *Morton* (Figura 2.5(c), (28)) e a ordem *Peano-Hilbert* (Figura 2.5(d), (30, 10)). A primeira diferença entre as ordens linha e linha-prima, e entre as ordens Morton e Peano-Hilbert, é que as primeiras têm a propriedade de que todo elemento é vizinho do elemento anterior na seqüência, tornando assim o grau de localidade ligeiramente superior. Ambas as ordens Morton e Peano-Hilbert esgotam hierarquicamente

PUC-Rio - Certificação Digital Nº 0410421/CA

(0,3)	(1,3)	(2,3)	(3,3)
(0,2)	(1,2)	(2,2)	(3,2)
(0,1)	(1,1)	(2,1)	(3,1)
(0,0)	(1,0)	(2,0)	(3,0)

Tabela 2.1: Posições de exemplo

12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

Tabela 2.2: Ordenação Linha

15	14	13	12
8	9	10	11
7	6	5	4
0	1	2	3

Tabela 2.3: Ordenação Linha Prima

5	7	13	15
4	6	12	14
1	3	9	11
0	2	8	10

Tabela 2.4: Ordenação Morton

5	6	9	10
4	7	8	11
3	2	13	12
0	1	14	15

Tabela 2.5: Ordenação Peano-Hilbert

9	10	14	15
3	8	11	13
2	4	7	12
0	1	5	6

Tabela 2.6: Ordenação Diagonal de Cantor

6	7	8	9
5	0	1	10
4	3	2	11
15	14	13	12

Tabela 2.7: Ordenação Espiral

um sub-quadrante antes de passar para o próximo. A ordem Peano-Hilbert possui propriedade de percorrer o espaço continuamente. A ordem Morton é simétrica, ao contrário da ordem Peano-Hilbert. A geração do índice da posição na seqüência, de um elemento do espaço, não é tão fácil para a ordem Peano-Hilbert como é para as demais ordens.

As outras ordens da Figura 2.5 são as ordens *diagonal de Cantor* (Figura 2.5(e)) e a *espiral* (Figura 2.5(f)). A ordem diagonal de Cantor percorre a imagem, saindo de um ponto da origem, visitando os elementos numa ordem similar à ordem linha-prima, com a diferença de que os elementos são visitados na ordem em que aumentam as suas distâncias $l1$ (para $p_1 = (x_1, y_1)$ e $p_2 = (x_2, y_2)$ temos $d_m = |x_1 - x_2| + |y_1 - y_2|$). Então, esta ordem é boa para ordenar um espaço que é ilimitado em duas direções a partir do primeiro quadrante. Por sua vez, a ordem espiral é atraente quando o espaço é ilimitado em quatro direções saindo da origem.

As ordens mais interessantes para estruturas de dados hierárquicas são as ordens Morton e Peano-Hilbert, por possuírem a propriedade de exaurirem o quadrante de forma hierárquica, podendo também ser usadas para ordenar um

PUC-Rio - Certificação Digital Nº 0410421/CA

espaço que teria sido agregado em quadrados. Das duas ordens, a mais usada é a ordem Morton, por causa de sua simplicidade no processo de conversão entre o índice e a posição espacial do elemento multidimensional.

2.4.2 Código Local

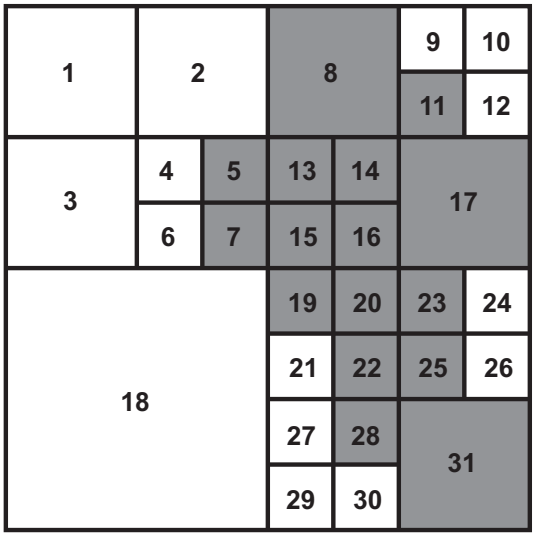


Figura 2.6: Exemplo de uma *quadtree*

Um *código local* é uma seqüência de dígitos ou símbolos que forma um índice de um nó numa representação hierárquica. Cada símbolo representa uma orientação da posição do nó, ao longo do seu caminho até a raiz. Em alguns casos, o nível do nó, ou é gravado no código local e calculado a partir dele, ou armazenado separadamente, podendo ter um número de dígitos fixos ou variados. Na Figura 2.6 está um exemplo de uma *quadtree* (2^2-*tree*), com índices numerados de acordo com o percorrimento em profundidade da árvore. Serão dados abaixo três exemplos de códigos locais numa *quadtree*.

Código Local Simples Considere uma *quadtree* de nível n, tendo sua raiz denotada por R e P um nó de nível m com $m \leq n$. Seja a seqüência de nós $[P_0, \ldots, P_m]$ o caminho de nós numa *quadtree*, que vai da raiz R até o nó P com $P_0 = R$, $P_m = P$ e $P_i = PAI(P_{i+1})$, para $i = 0, \ldots, m-1$, onde a função PAI retorna o nó pai do nó de entrada da função. Considere o caminho de orientações dos filhos numa *quadtree* que vai da raiz R até o nó P, sendo a seqüência de códigos de orientação $< C_0, \ldots C_{m-1} >$ com $C_i = FILHO(P_i)$, para $i = 0, \ldots, m-1$, onde a função $FILHO$ retorna 0, 1, 2 ou 3, que são as orientações $NO(0)$, $NE(1)$, $SO(2)$ e $SE(3)$ de P_i com relação a P_{i-1}. O *código local simples* do nó P pertencente à *quadtree* de nível

PUC-Rio - Certificação Digital Nº 0410421/CA

n é dado por:

$$\begin{cases} C_i = FILHO(P_i) & \forall i = 1, \ldots, m \\ C_P^s = \sum_{i=0}^{m-1} 4^{m-i-1} C_i \end{cases}$$

O nó 13 da Figura 2.6, possui a seqüência de códigos de orientação dada por $< NE, SO, NO >$, ou seja $< 1, 2, 0 >$, e então o seu código local simples é igual a $C_{13}^s = 4^2 \cdot 1 + 4^1 \cdot 2 + 4^0 \cdot 0 = 24$. A *quadtree* da Figura 2.7 é a *quadtree* da Figura 2.6 indexada por códigos locais simples. Essa definição de código local simples exige que o nível do nó seja armazenado separadamente, pois os nós 3 e 18 da Figura 2.6, possuem o mesmo código local simples igual a 2, logo sem os seus níveis não seria possível diferenciá-los.

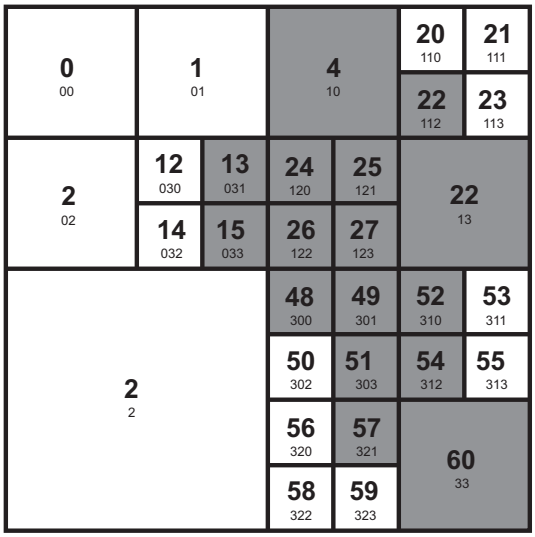


Figura 2.7: *Quadtree* codificada com o código local simples

Código Local de Tamanho Fixo Para uma formulação de código local de tamanho fixo, isto é, com quantidade de códigos de orientação sendo a mesma para qualquer nó, é preciso definir um novo código que represente uma ausência de orientação, sendo diferente das orientações usadas para a orientação dos filhos na *quadtree*. Para um nó P de nível m, pertencente à *quadtree* de nível n ($n \geq m$), seja a seguinte formulação de código local:

$$\begin{cases} C_i = FILHO(P_i) & \forall i = 0, \ldots, m-1 \\ C_i = 4 & \forall i = m, \ldots, n \\ C_P^{fl} = \sum_{i=0}^{n-1} 5^{n-i-1} C_i \end{cases}$$

Essa formulação de código local é chamada de *código local de tamanho fixo* ou *código local FL* (do inglês FL = *Fixed Length*) e foi definida por

PUC-Rio - Certificação Digital Nº 0410421/CA

Gargantini em (15). O nó 13 da Figura 2.6 possui a seqüência de orientações dada por $< 1, 2, 0 >_{13}$, então seu código local FL é igual a $C_{13}^{fl} = 5^2 \cdot 1 + 5^1 \cdot 2 + 5^0 \cdot 0 = 35$. A *quadtree* da Figura 2.8 é a *quadtree* da Figura 2.6 indexada por códigos locais FL.

Nessa formulação, os códigos locais sempre possuem tamanhos fixos iguais a n; o número 4 representa a ausência de códigos de orientação para nós folhas que possuem níveis menores que o nível n da *quadtree*. Observe que não é preciso armazenar o nível do nó; ele pode ser calculado contando o número de divisões sucessivas desse código local por 5, cujos restos das divisões são os códigos de orientação decodificados, antes de encontrar um resto igual a 4. Quando são armazenados apenas os nós folhas, a *quadtree* é chamada de *quadtree linear FL*, podendo ainda reconstruir explicitamente a *quadtree* original a partir dos códigos locais FL (15).

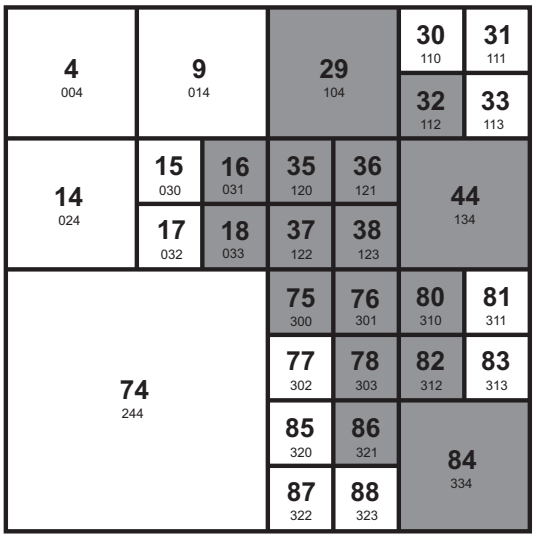


Figura 2.8: *quadtree* codificada com o código local FL

Código Local de Tamanho Variado Para a operação de decodificação do código local, são feitas divisões sucessivas, e os restos das divisões são os códigos de orientação decodificados. É importante que esse processo seja rápido e simples de se calcular. A decodificação do código local deve ser de tal modo que os dígitos sejam obtidos na ordem em que se percorre a *quadtree*, começando da raiz até as folhas. Infelizmente, da maneira que foram definidos os códigos anteriores, obtém-se os dígitos na ordem inversa. Uma codificação que satisfaz essa exigência de decodificação pode ser obtida alterando a forma do cálculo do código local de um nó P de nível m, pertencente à *quadtree* de nível n ($n \geq m$),

PUC-Rio - Certificação Digital Nº 0410421/CA

para:

$$\begin{cases} C_i = FILHO(P_i) + 1 \quad \forall i = 0, \ldots, m-1 \\ C_P^{vl} = \sum_{i=0}^{m-1} 5^i C_i \end{cases}$$

Essa formulação de código local é chamada de *código local de tamanho variado* ou *código local VL* (do inglês VL = *Variable Length*), e foi definida por Gargantini em (7). O nó 13 da Figura 2.6, possui a seqüência de orientações dada por $< 2, 3, 1 >$, então seu código local VL é igual a $C_{13}^{vl} = 5^0 \cdot 2 + 5^1 \cdot 3 + 5^2 \cdot 1 = 42$. A *quadtree* da Figura 2.9 é a *quadtree* da Figura 2.6 indexada por códigos locais VL. Uma representação de nós de uma *quadtree* usando qualquer tipo de código local tem o potencial de economizar espaço, quando comparado com a representação explícita de uma *quadtree*; mais espaço ainda pode ser economizado se armazenarmos apenas os códigos locais dos nós folhas. A lista de códigos locais VL dos nós folhas de uma *quadtree* é chamada de *quadtree linear VL*, podendo ainda reconstruir explicitamente a *quadtree* original, a partir dos códigos locais VL (7).

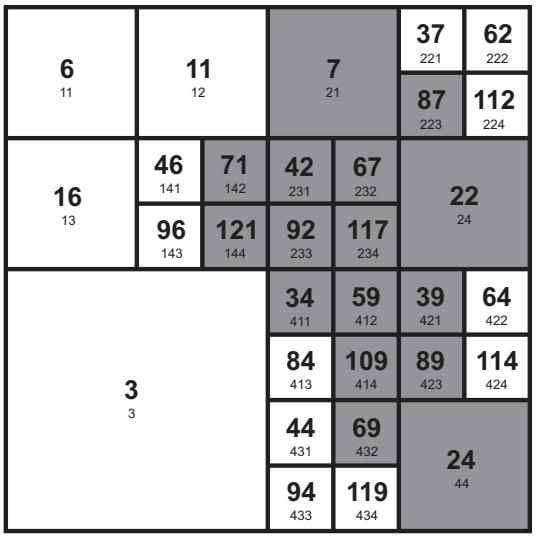


Figura 2.9: *Quadtree* codificada com o código local VL

Tamanho Fixo *versus* **Tamanho Variado** O código local FL tem várias desvantagens quando comparado com o código local VL. Uma delas é que os códigos locais são grandes, pois todos possuem sempre n dígitos, ao contrário dos códigos locais VL, cujos números de dígitos são variáveis. Outra desvantagem é que refinando a *quadtree* em mais um nível, é preciso recodificar os nós existentes multiplicando todos os códigos por 5; já os códigos locais VL não se alteram com o refinamento da *quadtree*. E também, quando é preciso obter o caminho da raiz da árvore até um nó específico, a decodificação do

PUC-Rio - Certificação Digital Nº 0410421/CA

código local VL já está na ordem certa de percorrimento, enquanto que a decodificação do código local FL está na ordem inversa.

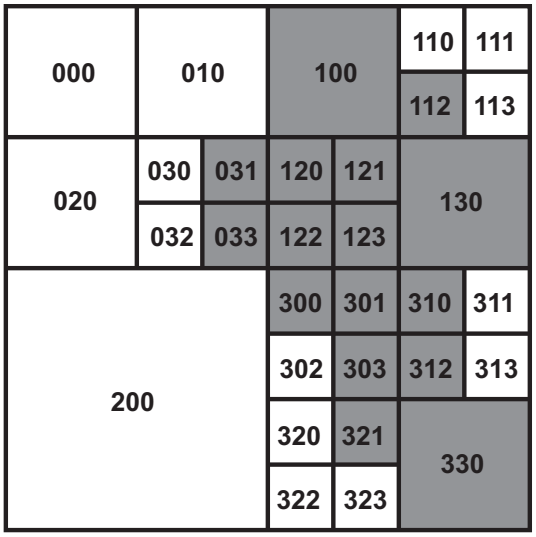


Figura 2.10: *Quadtree* codificada com o código local FD

Código Local de Profundidade Fixa Para evitar o uso de operações de módulo e divisões inteiras no processo de decodificação, Gargantini em (14) introduziu o chamado *código local de profundidade fixa* ou *código local FD* (do inglês FD = *Fixed Depth*), também conhecido como *quadnode*. Ele consiste da seqüência de códigos das orientações NO, NE, SO e SE representados pelos números 0, 1, 2 e 3, respectivamente, similar aos códigos locais simples, VL e FL, mas sem a codificação. Para uma *quadtree* de nível n, o código local FD de um nó de nível m ($m < n$) possui n dígitos, cujos $n - m$ últimos são preenchidos por 0. Sua formulação para um nó P de nível m, pertencente à *quadtree* de nível n ($n \geq m$), é a seguinte:

$$\begin{cases} C_i = FILHO(P_i) & \forall i = 0, \ldots, m-1 \\ C_i = 0 & \forall i = m, \ldots, n \\ C_P^{fd} = \sum_{i=0}^{n-1} 10^{n-i-1} C_i \end{cases}$$

O nó 13 da Figura 2.6 possui a seqüência de orientações dada por $< 1, 2, 0 >$; então seu código local FD é igual a $C_{13}^{fd} = 120$. A *quadtree* da Figura 2.10 é a *quadtree* da Figura 2.6 indexada por códigos locais FD. As operações de decodificação são evitadas, pois o código já está na base 10, e os códigos orientação são os dígitos do código local FD. Sua única desvantagem é que o nível não é conhecido, tendo que ser armazenado separadamente. Por

PUC-Rio - Certificação Digital Nº 0410421/CA

exemplo, para um nó com código local FD igual a 0, não se sabe o seu nível; o mesmo acontece para o nó 1 da Figura 2.6.

2.5 Armazenamento

Esta seção descreve alguns dos tipos mais conhecidos de armazenamento de estruturas hierárquicas. O armazenamento na forma de árvore possui um acesso em média logarítmico, mas possui uma sobrecarga associada à sua hierarquia. Já num armazenamento numa *hash table*, o acesso varia entre linear e constante.

2.5.1 Vetor

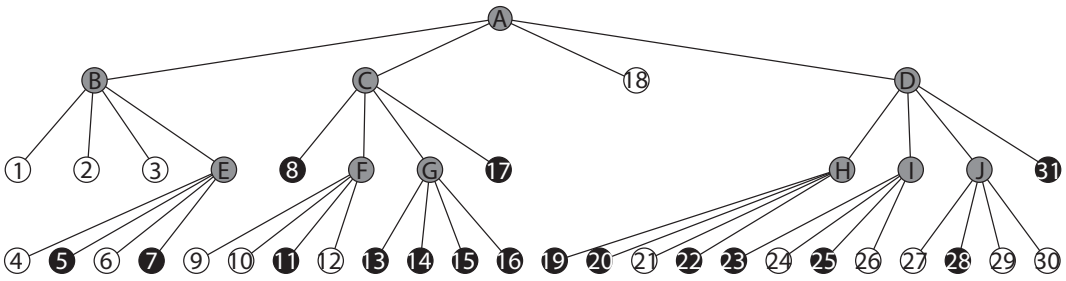
É maneira mais simples e conhecida de se armazenar dados. A maioria das linguagens possui um formato nativo de armazenamento em vetor.

2.5.2 Árvore

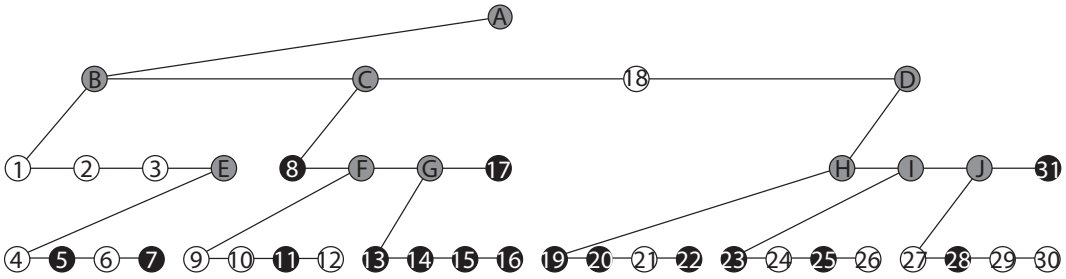
Uma *árvore* é uma estrutura de armazenamento hierárquica que imita a estrutura de árvore da natureza, com o conjunto de raiz e nós conectados, mas com a diferença de possuírem sua raiz no topo e os ramos para baixo. O vocabulário é similar ao das representações hierárquicas. O *nó raiz*, ou apenas *raiz*, é o nó no topo da árvore. O *nó pai*, ou apenas *pai*, de um nó qualquer, é aquele que está diretamente conectado acima dele na hierarquia. *Nó filho*, ou apenas *filho*, de um nó qualquer, está diretamente conectado abaixo dele na hierarquia. Nós com o mesmo nó pai são chamados de *nós irmãos*, ou apenas *irmãos*. Os nós chamados de *nós folhas*, ou apenas *folhas*, são aqueles que não possuem nós filhos. Note que cada nó da árvore pode ser considerado um nó raiz para a sub-árvore que contém todos os nós que estão conectados abaixo dele na hierarquia. Um *ramo* é uma seqüência de nós tal que o primeiro é o pai do segundo, o segundo é o pai do terceiro, e assim por diante. O *tamanho de um ramo* é o número de nós da seqüência. A *profundidade* ou *nível* de uma árvore é o tamanho do maior ramo que ela possui. As operações de busca e armazenamento numa árvore possuem tempos de acesso com uma complexidade média que varia entre linear no pior caso, e logarítmica no melhor caso, dependendo do número de filhos.

A implementação de uma árvore pode ser feita de várias formas. Na maneira tradicional, cada nó pai possui um ponteiro para cada filho, tendo opcionalmente um ponteiro para o pai. Chamaremos esse tipo de *implementação*

PUC-Rio - Certificação Digital Nº 0410421/CA



a) Implementação clássica



b) Implementação filho-irmão

Figura 2.11: Armazenamento da *quadtree* da Figura 2.6 na forma de árvore com diferentes implementações.

clássica. Outra forma ocorre quando cada nó pai possui um ponteiro para uma lista encadeada de filhos, tendo opcionalmente um ponteiro para o pai, que chamaremos de *implementação filho-irmão*. Na Figura 2.11 encontra-se exemplos de um armazenamento da *quadtree* da Figura 2.6 na forma de árvore com diferentes implementações. A Figura 2.11(a) é a implementação clássica, e a Figura 2.11(b) é a implementação filho-irmão.

Árvore Binária A *árvore binária* é uma árvore que possui no máximo dois filhos por nó. Assim cada nó pode ser identificado como sendo o filho esquerdo ou o filho direito. A *árvore binária de busca* é uma árvore binária que possui uma ordem pré-determinada no dado armazenado em cada nó. Como, por exemplo, o filho esquerdo armazena um valor sempre menor do que o armazenado no filho direito do mesmo pai. As operações de busca e armazenamento numa árvore binária possuem tempos de acesso com uma complexidade média que varia entre linear igual a $O(n)$ no pior caso, e logarítmica igual a $O(\log_2(n))$ no melhor caso, onde n é o nível da árvore.

Árvore AVL A árvore *AVL* é uma árvore binária auto-balanceada, no sentido de que o nível de duas sub-árvores de qualquer nó se diferem de no máximo um. O fator de balanço de um nó é o nível da sua sub-árvore da direita, menos o nível da sua sub-árvore da esquerda. Um nó com fator de balanço 1, 0, ou -1 é considerado balanceado. Um nó com qualquer outro fator de balanço

PUC-Rio - Certificação Digital Nº 0410421/CA

é considerado desbalanceado, exigindo assim uma modificação na árvore. O fator de balanço de cada nó é armazenado no próprio nó, ou calculado a partir dos níveis das suas sub-árvores. A árvore AVL tem esse nome devido aos seus dois inventores: Adelson-Velsky & Landis (1). As operações de busca e armazenamento numa árvore AVL possuem tempos de acesso com uma complexidade logarítmica igual a $\Theta(\log_2(n))$, onde n é o nível da árvore.

2.5.3 *Hash Table*

A *hash table* é uma estrutura de dados similar a uma agenda de telefone, podendo rapidamente determinar se um item pertence ou não a um subconjunto, olhando a primeira letra do nome. Sua principal idéia é a de diminuir as buscas necessárias durante o acesso ou armazenamento dos dados, pela divisão do conjunto de itens em subconjuntos menores, de tal forma que cada busca pertença apenas a um subconjunto. Um subconjunto é determinado mapeando todos os itens num conjunto de inteiros por uma função, armazenando os itens nas posições definidas por esses inteiros em um vetor.

Mais especificamente, seja $\mathcal{U}$ um conjunto de n itens a ser armazenados em um vetor $T[0, 1, \ldots, m-1]$ de tamanho m. A *função de hash* é a função

$$\begin{array}{rccc} h: & \mathcal{U} & \rightarrow & \{0, 1, \ldots, m-1\} \\ & x & \mapsto & h(x) \end{array}$$

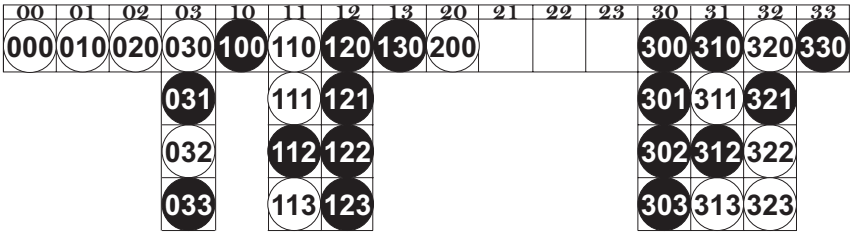
que mapeia cada item $x \in \mathcal{U}$ num recipiente $T[h(x)]$ de posição $h(x)$ do vetor $T[0, 1, \ldots, m-1]$. Esse vetor de recipientes é chamado de *hash table*.

Se $n = m$, então a função de *hash* pode ser definida como $h(x_i) = i$. Tal função de *hash bijetora*, que mapeia todos os itens na *hash table*, preenchendo todos os recipientes, sem criar colisões, é chamada de *função de hash perfeita*. Tais funções só podem ser geradas para um conteúdo estático, limitando as suas aplicações. Para os outros casos, existirão itens x e $y \in \mathcal{U}$ tais que $h(x) = h(y)$. Quando isso acontece, é dito que x e y *colidem*. Os métodos mais comuns para resolver as *colisões* são o *método do encadeamento* e o *método do endereçamento aberto*. Uma discussão detalhada da diferença dos dois métodos pode ser encontrada em (27).

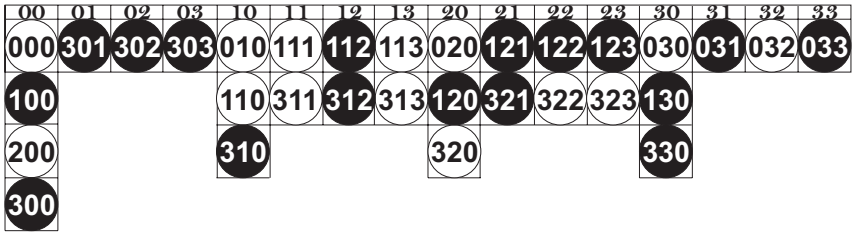
Método do Encadeamento Uma *hash table* onde cada recipiente $T[h(x_i)]$ não possui apenas um item, mas uma lista encadeada de itens $(h(x_i^1), \ldots, h(x_i^j))$ que possuem a mesma função de *hash* $h(x_i^1) = \cdots = h(x_i^j)$, é chamada de *hash table encadeada*. Para verificar se um item x está na *hash table*, é preciso varrer toda a lista encadeada $T[h(x_i)]$. Então o tempo total de

PUC-Rio - Certificação Digital Nº 0410421/CA

remoção e inserção de um item x numa *hash table* encadeada é de $O(l(x))$, onde $l(x)$ é o tamanho da lista encadeada $T[h(x)]$.



a) A função de *hash* é igual aos dois *primeiros* dígitos do código local FD



b) A função de *hash* é igual aos dois *últimos* dígitos do código local FD

Figura 2.12: Armazenamento da *quadtree* da Figura 2.6 na forma de *hash table* com diferentes funções de *hash*.

A Figura 2.12 representa o armazenamento apenas dos nós folhas da *quadtree* da Figura 2.6 na forma de uma *hash table* encadeada com funções de *hash* diferentes, onde os itens são os nós da *quadtree* com os seus respectivos códigos locais FD sem os níveis. A função de *hash* da Figura 2.12(a) é igual aos dois primeiros dígitos do código local FD de cada nó. Já a função de *hash* da Figura 2.12(b) é igual aos dois últimos dígitos do código local FD de cada nó. A existência de recipientes vazios e várias listas encadeadas de colisões contendo quatro entradas na *hash table* da Figura 2.12(a) mostra um exemplo de uma função de *hash* que não é eficiente, ao contrário do que acontece na *hash table* da Figura 2.12(b).

No pior caso, todos os n itens seriam registrados num mesmo inteiro, e por conseqüência disto seriam armazenados num mesmo recipiente. Existiria então apenas uma longa lista encadeada de n entradas, logo o tempo de acesso teria uma complexidade linear igual a $O(n)$. Para calcular o tempo esperado de uma busca sem sucesso de um item x não pertencente ao conjunto $\mathcal{U}$ de n itens, que está armazenado em uma *hash table* encadeada $T[0, 1, \ldots, m-1]$ de tamanho m, supõe-se usualmente que a função de *hash* é completamente aleatória, isto é, $\forall x$ e $y \in \mathcal{U}$, tais que $x \neq y$, a probabilidade de serem mapeados num mesmo valor $h(x) = h(y)$ é igual a $1/m$: $x \neq y \Rightarrow P[h(x) = h(y)] = 1/m$. Se o item buscado x não pertence ao conjunto $\mathcal{U}$, o valor esperado de $l(x)$ pode

PUC-Rio - Certificação Digital Nº 0410421/CA

ser calculado definindo a variável indicadora

$$C_{x,y} = \begin{cases} 0 & \text{se } h(x) \neq h(y) \\ 1 & \text{se } h(x) = h(y) \end{cases} .$$

E então temos que

$$x \neq y \Rightarrow P[h(x) = h(y)] = E[C_{x,y}] = \frac{1}{m}.$$

Como o tamanho da lista encadeada $T[h(x)]$ é igual à quantidade de itens que colidem com x, logo

$$l(x) = \sum_{y \in T} C_{x,y}.$$

Portanto, o valor esperado de $l(x)$ é

$$E[l(x)] = \sum_{y \in T} E[C_{x,y}] = \sum_{y \in T} \frac{1}{m} = \frac{n}{m}.$$

Essa razão é chamada de *fator de balanço* de uma *hash table*, e como ela independe do item escolhido, é definida como sendo

$$\alpha = \frac{n}{m}.$$

A análise feita implica que o tempo médio esperado para uma busca sem sucesso numa *hash table* encadeada é $\Theta(\alpha)$. Então, enquanto a quantidade n de itens difere do tamanho m da *hash table* de uma constante, o tempo de busca é constante. Uma busca com sucesso possui um tempo esperado menor do que uma busca sem sucesso, logo o tempo esperado para uma busca com sucesso numa *hash table* encadeada é constante. O mesmo acontece para a inserção e para a remoção.

Listas encadeadas não são as únicas estruturas de armazenamento que podem ser usadas para um armazenamento de uma *hash table* encadeada, qualquer estrutura de armazenamento pode ser usada para armazenar as listas encadeadas. Por exemplo, se é possível definir uma ordem no conjunto $\mathcal{U}$, cada lista encadeada pode ser armazenada numa árvore binária de busca. Isso reduziria o tempo médio de busca no pior caso para $\Theta(\log_2 l(x))$, e sobre a suposição de uma função de *hash* aleatória, o tempo de busca esperado seria de $\Theta(\log_2 \alpha)$.

Outra possibilidade é armazenar as listas encadeadas em outras *hash tables*. Para ser eficiente, deve-se garantir que o fator de balanço para as *hash tables* secundárias é sempre uma constante menor do que 1, fazendo assim com

PUC-Rio - Certificação Digital Nº 0410421/CA

que o fator de balanço da *hash table* principal seja também constante.

Método do Endereçamento Aberto Outro método capaz de resolver as colisões é chamado de *endereçamento aberto*. Ao invés de depender de estruturas de armazenamento secundárias, este método resolve as colisões investigando por outro lugar na *hash table*. Considere uma seqüência de funções de *hash* $\{h_0, h_1, \ldots, h_{m-1}\}$, tais que para qualquer item $x \in \mathcal{U}$ a *seqüência investigadora* $\{h_0(x), h_1(x), \ldots, h_{m-1}(x)\}$ é uma permutação de $\{0, 1, \ldots, m-1\}$. Isto significa que diferentes funções de *hash* da seqüência mapeiam x em diferentes posições na *hash table*. Note que sempre se deve ter $n \geq m$ e então o fator de balanço é sempre menor do que um: $\alpha \leq 1$. Ao inserir um item x, se a primeira posição $h_0(x)$ estiver ocupada na *hash table*, outra posição $h_1(x)$ é sugerida, e assim por diante, até encontrar uma posição $h_i(x)$ vazia, no caso em que a *hash table* não estiver completamente cheia.

A busca de um item x numa *hash table* com endereçamento aberto é feita da seguinte forma: começando com a primeira função de *hash* h_i $(i = 0)$ da seqüência $\{h_0, h_1, \ldots, h_{m-1}\}$, se o item x estiver na posição $h_i(x)$ da *hash table*, $x = T[h_i(x)]$, o item x foi encontrado e a busca acaba. Caso contrário, se essa posição estiver vazia, $T[h_i(x)] = \varnothing$, é porque o item x não está na *hash table*. E, por fim, se a posição $h_i(x)$ não está vazia, $T[h_i(x)] \neq \varnothing$, é feito o mesmo processo com a segunda função de *hash* da seqüência, e assim por diante. Se todas as funções de *hash* foram testadas e não se tem um veredito, é porque o item x não está na *hash table*, e a mesma está completamente cheia.

A inserção de um item x numa *hash table* com endereçamento aberto se comporta da mesma forma, mas com uma única diferença: quando a posição $h_i(x)$ estiver vazia, $T[h_i(x)] = \varnothing$, o item x é inserido nessa posição, $T[h_i(x)] = x$. E, ao final, quando todas as funções de *hash* forem testadas e o item x não foi inserido, conclui-se que a *hash table* está completamente cheia.

A remoção de um item numa *hash table* com endereçamento aberto é um pouco mais complicada. Não se pode simplesmente remover um item da *hash table*, pois é preciso saber que o recipiente foi preenchido em algum momento para garantir a sincronia, ao inserir ou buscar itens das funções de *hash* da seqüência investigadora. Portanto, quando um item é removido de um recipiente, deve ser marcado como sendo um recipiente *desperdiçado*. É claro que ao inserir e remover muitos itens, a *hash table* com endereçamento aberto ficaria com muitos recipientes vazios, mas que foram anteriormente marcados como desperdiçados, impedindo uma nova inserção. Por isso, para garantir um bom desempenho, é preciso respeitar duas regras: primeira, quando o número de itens armazenados numa *hash table* de tamanho m exceder $\frac{m}{4}$,

PUC-Rio - Certificação Digital Nº 0410421/CA

deve-se construir uma nova *hash table* com o tamanho dobrado; segunda, se o número de recipientes marcados ultrapassar $\frac{m}{2}$, deve-se construir uma nova *hash table* que utilize recipientes marcados.

Para calcular o tempo esperado de uma busca sem sucesso de um item x não pertencente ao conjunto $\mathcal{U}$ de n itens, que está armazenado em uma *hash table* com endereçamento aberto $T[0, 1, \ldots, m-1]$ de tamanho m, é preciso observar que o valor inicial $h_0(x)$ é igual a algum inteiro do conjunto $\{0, 1, \ldots, m-1\}$ pela definição da seqüência investigadora. Usando o modelo anterior, isto implica que a probabilidade do recipiente $T[h_0(x)]$ estar ocupado é igual a $\frac{n}{m}$. Se a primeira investigação for ignorada, a seqüência investigadora resultante $\{h_1(x), \ldots, h_{m-1}(x)\}$ é igual a alguma permutação do conjunto menor $\{0, 1, \ldots, m-1\} \setminus \{h_0(x)\}$ (sem o elemento $h_0(x)$). Isto implica que se o recipiente $T[h_0(x)]$ estiver ocupado, a busca segue recursivamente. Para usar essa recursão, estará suposto que o recipiente $T[h_0(x)]$ nunca será investigado. Então, a recorrência do número esperado de investigações, em função de m e n, implica:

$$E[T(m,n)] = 1 + \frac{n}{m} E[T(m-1, n-1)].$$

O caso trivial é $T(m, 0) = 1$. Se a *hash table* estiver vazia, a primeira investigação sempre retorna um recipiente vazio. Será provado por indução que $E[T(m,n)] \leq \frac{m}{(m-n)}$:

$$\begin{aligned} E[T(m,n)] &= 1 + \tfrac{n}{m} E[T(m-1, n-1)] \\ &\leq 1 + \tfrac{n}{m} \cdot \tfrac{m-1}{m-n} && \text{[hipótese de indução]} \\ &\leq 1 + \tfrac{n}{m} \cdot \tfrac{m}{m-n} && [m-1 < m] \\ &= \tfrac{m}{m-n} \, . \end{aligned}$$

Reescrevendo em termos do fator de balanço $\alpha = \frac{n}{m}$, tem-se que $E[T(m,n)] \leq \frac{1}{(1-\alpha)}$. Então, o tempo médio esperado de uma busca sem sucesso numa *hash table* com endereçamento aberto é constante, pois $\alpha \leq 1$, a não ser que a *hash table* esteja quase toda completa. O mesmo acontece para uma busca com sucesso e para uma inserção.

Na prática, a seqüência de funções de *hash* que investigam a existência de um item x na *hash table* é criada de acordo com uma das seguintes heurísticas:

Investigação Linear: Dada uma função de *hash* $h(x)$, define-se h_i como sendo $h_i(x) = (h(x) + i) \bmod m$. Uma seqüência simples, mas as colisões tendem a agrupar os itens deixando muitos buracos na função de *hash*. Para garantir que a seqüência investigadora $\{h_1(x), \ldots, h_{m-1}(x)\}$ seja uma permutação do conjunto $\{0, 1, \ldots, m-1\}$, é preciso ter m relativamente primo com i.

PUC-Rio - Certificação Digital Nº 0410421/CA

Investigação Quadrática: Dada uma função de *hash* $h(x)$, define-se h_i como sendo $h_i(x) = (h(x)+i^2) \mod m$. Infelizmente, para alguns valores de m a seqüência investigadora $\{h_1(x), \ldots, h_{m-1}(x)\}$ não é uma permutação do conjunto $\{0, 1, \ldots, m-1\}$. Isto pode ser evitado tomando m como sendo um número primo. Embora a investigação quadrática não sofra dos mesmos problemas que a investigação linear, ela possui o seguinte problema de agrupamento: se dois itens possuem o mesmo valor inicial, as respectivas seqüências investigadoras serão as mesmas.

Hashing **Duplo:** Dadas duas funções de *hash* $h(x)$ e $h'(x)$, define-se h_i como sendo $h_i(x) = (h(x) + i \cdot h'(x)) \mod m$. Para garantir que a seqüência investigadora $\{h_1(x), \ldots, h_{m-1}(x)\}$ seja uma permutação do conjunto $\{0, 1, \ldots, m-1\}$, é necessário que os valores da função de *hash* $h'(x)$ e o tamanho da função de *hash* m sejam primos entre si, bastando simplesmente tomar m como sendo um primo. O duplo *hashing* evita os problemas de agrupamento que as investigações lineares e quadráticas possuem.

Enfim, o método do endereçamento aberto é mais apropriado quando se tem uma pequena quantidade de itens para armazenar, pois, por não possuir um armazenamento externo, o número de elementos deve ser menor ou igual ao tamanho da *hash table*. E, na prática, para uma *hash table* com poucos recipientes vazios, as seqüências de funções de *hash* têm mais dificuldade de armazenar um item, podendo até não conseguir preencher toda a *hash table*.

Nas análises a seguir, será considerado que a complexidade média do acesso de um item numa *hash table* é de ordem constante.

PUC-Rio - Certificação Digital Nº 0410421/CA

3
Otimização de Buscas

Este trabalho surgiu da seguinte observação: como as folhas são os nós mais distantes da raiz, por que começar a partir da raiz quando se está à procura de uma folha? A resposta para uma representação clássica de uma 2^d–*tree* é que não existe outra forma de acessar uma folha. Existem 2^d–*trees* que permitem acessar em tempo constante qualquer nó, dada sua posição e seu nível. A posição é conhecida quando se procura por um nó, mas a profundidade não. A estimativa do nível será p objetivo da otimização.

Será apresentado neste capítulo um novo método para otimização do tempo de execução de buscas numa 2^d–*tree*. A otimização independe da dimensão e da implementação. A única exigência é o uso de uma 2^d–*tree* que permita acessar em tempo constante os seus nós, e um eficiente método de indexação que satisfaça os requisitos estabelecidos na primeira seção. Buscas diretas, buscas de vizinhos e buscas num certo raio são otimizadas através de uma modelagem do custo de tais buscas. A estrutura de dados e os algoritmos usados para validar essas otimizações serão apresentados e detalhados no próximo capítulo. Este capítulo generaliza o método proposto em (6) pela independência do armazenamento e da indexação, estabelecendo apenas quatro requisitos para a otimização.

PUC-Rio - Certificação Digital Nº 0410421/CA

3.1
Requisitos para uma 2^d–*Tree* Aberta

Uma 2^d–*tree* que permita acessar em tempo constante os seus nós será chamada de *2^d–tree aberta*. Exemplos de tais estruturas existem na literatura. Gargantini em (7) define a *octree linear*, na qual apenas os nós folhas são armazenados em um vetor ordenado. Já Glassner em (21) descreve outra implementação de uma *octree linear*, mas com muitas inovações: uma delas é o uso de uma *hash table* (seção 2.5.3), ao invés de um vetor ordenado. Mais recentemente, Warren e Salmon em (12) discutem uma paralelização de uma *octree*, cuja implementação armazena seus nós em uma *hash table* em paralelo. A implementação de uma 2^d–*tree* aberta, usada para validar as otimizações aqui propostas, é uma *hashed* 2^d–*tree*.

A 2^d–*trees* abertas possuem um armazenamento memória eficiente e fornecem ao mesmo tempo, um acesso em tempo constante e hierárquico. Mas seu calcanhar de Aquiles é a indexação, pois é preciso que cada nó possua um único índice (seção 2.4.2), que é chamado de *chave*, de forma a ser armazenado numa estrutura com o tempo de acesso constante. Dependendo do método usado para indexar os nós, a 2^d–*tree* aberta pode não ser tão eficiente como uma 2^d–*tree* clássica, que armazena seus nós numa árvore implementada com ponteiros explícitos.

Para ser eficiente, um método de indexação para uma 2^d–*tree* aberta deve satisfazer os quatro requisitos abaixo:

1. *Único*: cada nó da 2^d–*tree* deve ter uma chave diferente.
2. *Hierárquico*: cálculo em tempo constante da chave dos nós pai e filho a partir da chave de um dado nó.
3. *Localizado*: cálculo em tempo constante da chave dos nós adjacentes.
4. *Espacial*: cálculo em tempo constante da chave a partir de uma posição e de um nível.

Para tal método de indexação, este capítulo irá modelar métodos otimizados de busca, analisando estatisticamente e comparando com os métodos clássicos de busca (seção 2.3).

3.2 Busca com 2^d–*Tree* Abertas

Será considerado o seguinte modelo de aplicações para fixar a terminologia: uma 2^d–*tree* armazena dados associados a posições do espaço. Os dados são acessíveis unicamente a partir das folhas. Uma busca procura a folha contendo uma determinada posição. Será usada uma 2^d–*tree* com indexação satisfazendo os requisitos da seção 3.1. As implementações desses algoritmos, no caso de uma *hashed* 2^d–*tree* com uma indexação de Morton, serão detalhadas no próximo capítulo. O seguinte algoritmo descreve como acessar uma folha (ou mais geralmente um nó qualquer) a partir da sua posição e um nível estimado $\hat{l}$. A próxima seção descreverá como estimar o nível de uma folha, supondo o uso de um método de indexação eficiente satisfazendo os quatro requisitos.

PUC-Rio - Certificação Digital Nº 0410421/CA

3.2.1
Busca Direta

O procedimento de busca direta surge da seguinte idéia: a fim de encontrar a (única) folha contendo um ponto p, o algoritmo gera a chave $k_l(p)$ de p no nível estimado $\hat{l}$. Ao buscar o nó $n_l(p)$ correspondente àquela chave na estrutura, três situações podem ocorrer:

1. O nó $n_l(p)$ é uma folha: o algoritmo então retorna $n_l(p)$.

2. O nó $n_l(p)$ existe, mas não é uma folha: isso significa que o nível estimado l é pouco profundo e o nível l é incrementado até que $n_l(p)$ se torne uma folha.

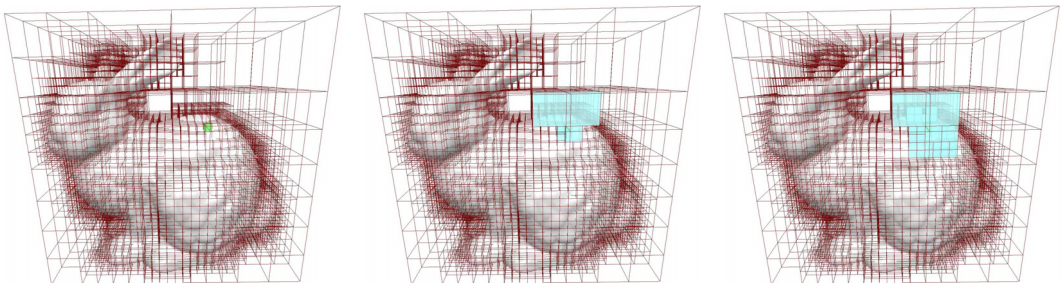
3. Não existe um nó que corresponda a $n_l(p)$: isso significa que o nível estimado l é muito profundo, e l é decrementado até $n_l(p)$ se tornar um nó válido e, portanto, uma folha.

Observe que se o nível estimado é zero, a busca começa da raiz e prossegue para o segundo caso, correspondendo neste caso à busca hierárquica clássica (seção 2.3). Mais ainda, a chave não precisa ser recalculada no terceiro caso, pois a chave de $n_{l-1}(p)$ pode ser deduzida a partir da chave de $n_l(p)$ (segundo requisito). Do mesmo modo, no segundo caso a chave de nível $l+1$ também pode ser deduzida a partir da chave de nível l. A Figura 3.1(a) ilustra a busca direta para a nuvem de pontos do modelo *Stanford Bunny* com $d = 3$.

3.2.2
Busca de Vizinhos Adjacentes

O procedimento de busca de vizinhos adjacentes se assemelha à busca direta, embora ele deva retornar uma lista de folhas. Foram consideradas duas opções de otimização para a busca dos nós adjacentes. A primeira opção é gerar todas as chaves dos vizinhos adjacentes no nível ótimo $\hat{l}$, e seguir a hierarquia da 2^{d}–*tree* aberta até encontrar as folhas. A segunda opção é gerar apenas as $3^d - 1$ chaves dos vizinhos de mesmo nível de um nó n, e seguir os três casos da busca direta. Nesta opção, deve ser modificado o segundo caso, pois várias chaves $k(n_i)$ podem ser geradas em cada incremento de nível, porém, existe apenas um $(l-1)$–vizinho adjacente se n_i compartilha um ponto (0-face) com n, ou melhor, está na diagonal de n, existem dois se n_i compartilha uma aresta (1-face) com n, de maneira geral, existem 2^f nós se n_i compartilha uma f-face com n. As chaves desses $(l-1)$–nós podem ser geradas (terceiro requisito) e o algoritmo é recursivo em cada um dos vizinhos adjacentes de nível $(l-1)$.

PUC-Rio - Certificação Digital Nº 0410421/CA



a) Busca Direta b) Busca de Vizinhos Adjacentes c) Busca de Vizinhos num raio

Figura 3.1: Ilustração do procedimento de buscas para $d = 3$. A *octree* foi adaptada para os 35.947 vértices da nuvem de pontos do modelo *Stanford Bunny*, onde a *octree* foi subdividida até conter apenas um ponto por folha.

Um erro do nível estimado na primeira opção pode resultar na geração de muitas chaves desnecessárias, por isso é mantida a busca hierárquica da segunda opção começando dos $3^d - 1$ nós adjacentes. Já o terceiro caso do algoritmo pode ser otimizado da mesma forma que a busca por folha: se um vizinho de n possui um nível menos profundo, é feita uma busca direta a partir do nível estimado $\hat{l}$ evitando muitas buscas intermediárias. Isso aproveita do seguinte fato: se o nó n está no nível mais profundo da 2^{d}–*tree* aberta, pelo menos seus irmãos, isto é, pelo menos $2^d - 1$ dos seus nós adjacentes têm o mesmo nível que ele. A Figura 3.1(b) ilustra a busca de vizinhos adjacentes para a nuvem de pontos do modelo *Stanford Bunny* com $d = 3$.

3.2.3 Busca de Vizinhos num raio

A busca por nós num dado raio em torno de um nó n imita a busca por vizinhos adjacentes. A única diferença está no segundo caso: existem $3^d - 1$ vizinhos de nível $l - 1$ para cada l-vizinho, ao invés de $1, 2, 2^2, \ldots, 2^f$ nós de acordo com a região que compartilham. Isso envolve mais gerações de chaves e acessos à estrutura do que a busca por vizinhos adjacentes. Mais ainda, se o raio for maior do que $\frac{1}{2^{\hat{l}}}$, alguns vizinhos num raio podem não ser adjacentes a n, e o conjunto inicial n_i de vizinhos num raio deve conter mais do que $3^d - 1$ nós. A mesma otimização do terceiro caso do algoritmo utilizada na busca por vizinhos adjacentes pode ser aplicada aqui obtendo os mesmo resultados. A Figura 3.1(c) ilustra a busca de vizinhos em torno de um raio para a nuvem de pontos do modelo *Stanford Bunny* com $d = 3$.

PUC-Rio - Certificação Digital Nº 0410421/CA

3.3 Modelagem das Buscas e Otimização Estatística

Os algoritmos de busca anteriores contavam com o nível estimado $\hat{l}$ para procurar uma folha. Se for definido esse nível como sendo zero, os algoritmos sempre começariam da raiz, assim, tanto a busca direta e a busca por vizinhos adjacentes se comportariam como um algoritmo de percorrimento numa 2^{d}–*tree* clássica. Como foi dito anteriormente, as folhas são os nós mais distantes da raiz, logo $\hat{l} = 0$ corresponde, *a priori*, ao pior caso. Será descrito um modelo estatístico simples para $\hat{l}$, de forma a otimizar o custo no procedimento de buscas. O cálculo do nível estimado $\hat{l}$ requer apenas um pequeno tempo extra durante a criação da 2^{d}–*tree* aberta e pode ser atualizado dinamicamente.

3.3.1 Modelo de Custo

O custo total de busca depende de quantas vezes, na média, cada folha n for procurada; considere que $f(n)$ seja esse número. Por exemplo, se for procurada cada posição no domínio da 2^{d}–*tree* aberta, $f(n)$ será proporcional ao tamanho do nó n, que é uma potência do seu nível l: $f(n) \sim 2^{d\cdot(l_{max}-l_n)}$. Se for procurado por algum dado relacionado à estrutura da 2^{d}–*tree* aberta, como uma detecção de colisão interna, pode ser escolhido $f(n) \sim 1$. A principal suposição é que $f(n)$ depende apenas do nível de n: $f(n) = f_l$. Na prática, isso significa que a 2^{d}–*tree* aberta é usada sem nenhum viés geométrico, que é o caso dos dois exemplos mencionados anteriormente. O custo médio de busca é então a soma nas folhas sobre o custo de buscar aquela folha, multiplicado pelo número de vezes da procura: $custo(\hat{l}) = \sum_{n \in folhas} f(n) \cdot custo_{\hat{l}}(l)$. Como o custo de buscar uma folha depende apenas do seu nível, ao se denotar por p_l o número de folhas de nível l, o somatório pode ser reescrito e indexado pelo nível:

$$custo(\hat{l}) = \sum_{l} p_l \cdot f_l \cdot custo_{\hat{l}}(l).$$

3.3.2 Busca Direta

O modelo de custo para os três casos da busca direta é o seguinte: a chave $k_l(n)$ é gerada em tempo constante (primeiro requisito), se o nível está corretamente estimado (caso 1), ou seja, $\hat{l} = l$, o nó é retornado diretamente com tempo constante c. A chave do nó pai ou de um nó filho é gerada em tempo constante (segundo requisito). No caso 2, $l > \hat{l}$ e o algoritmo gera $l - \hat{l}$ chaves de nós filhos. No caso 3, $l < \hat{l}$ e o algoritmo gera $\hat{l} - l$ chaves de nós

PUC-Rio - Certificação Digital Nº 0410421/CA

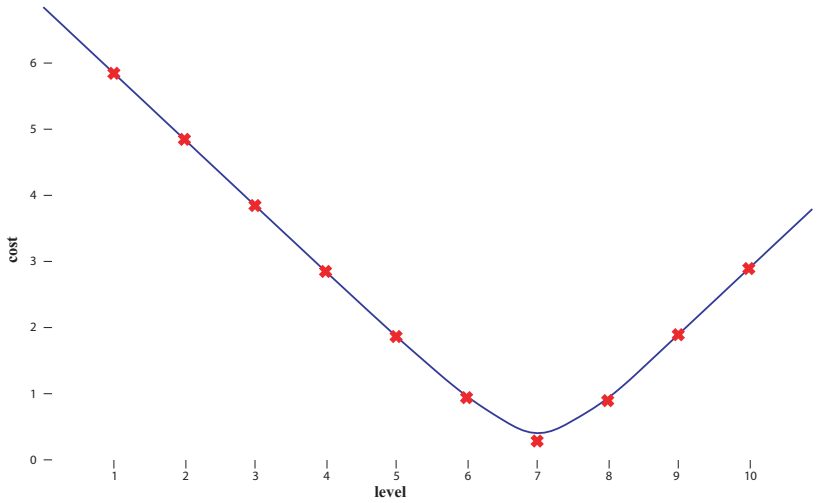


Figura 3.2: Função de custo discreta $custo(\hat{l})$ e a função de custo diferenciável custo$(\hat{l})$ para o modelo *Stanford Bunny*: o custo diferenciável tem um único mínimo, cuja parte inteira é o mínimo do custo discreto.

pais. Então, para todos os casos, são geradas $|l - \hat{l}|$ chaves quando se busca um nó de nível l. O custo da busca é a soma do custo constante c para a geração da chave mais uma sobrecarga proporcional à diferença $|l - \hat{l}|$:

$$custo(\hat{l}) = c + \sum_{l<\hat{l}} p_l \cdot f_l \cdot (\hat{l} - l) + \sum_{l>\hat{l}} p_l \cdot f_l \cdot (l - \hat{l}).$$

3.3.3 Otimização do Custo

Com o intuito de otimizar o valor de $\hat{l}$ de forma a minimizar o custo, será feita uma análise da função custo $custo(\hat{l})$ montando uma interpolação local diferenciável custo$(\hat{l})$, que possua os mesmos valores nos pontos conhecidos preservando sua distribuição parabólica possuindo um único mínimo, como pode ser visto na Figura 3.2. Considera-se então $p(l)$ como sendo a função contínua para p_l, o número de folhas de nível l, e considera-se também $f(l)$ como sendo a função contínua para f_l, o número de vezes que uma folha de nível l é buscada. A construção de $p(l)$ pode ser feita definindo-a por partes em cada intervalo inteiro $[l, l+1]$ onde cada parte é uma parábola de área p_l como pode ser visto na Figura 3.3. Uma construção similar $f(l)$ também pode ser feita. Tais construções permitem que a função custo seja reescrita na sua forma contínua:

$$custo(\hat{l}) = c + \int_0^{\hat{l}} p(l) \cdot f(l) \cdot (\hat{l} - l)\, \mathrm{d}l + \int_{\hat{l}}^{\infty} p(l) \cdot f(l) \cdot (l - \hat{l})\, \mathrm{d}l.$$

PUC-Rio - Certificação Digital Nº 0410421/CA

Como as funções integráveis que estão na definição do custo são contínuas, o custo é uma função contínua e, portanto, diferenciável. Para distribuições genéricas $p(l)$ e $f(l)$, a função custo tem um único mínimo local l_m que pode ser calculado pela seguinte diferenciação:

$$\begin{aligned}
\frac{\mathsf{d}}{\mathsf{d}\hat{l}}\left(\text{custo}(\hat{l})\right) &= \frac{\mathsf{d}}{\mathsf{d}\hat{l}}\left(c + \int_0^{\hat{l}} p(l)f(l)\left(\hat{l}-l\right)\ \mathsf{d}l + \int_{\hat{l}}^{\infty} p(l)f(l)\left(l-\hat{l}\right)\ \mathsf{d}l\right) \\
&= \frac{\mathsf{d}}{\mathsf{d}\hat{l}}\left(\hat{l}\cdot\int_0^{\hat{l}} p(l)f(l)\ \mathsf{d}l\ - \int_0^{\hat{l}} p(l)f(l)\cdot l\ \mathsf{d}l\right. \\
&\qquad \left. + \int_{\hat{l}}^{\infty} p(l)f(l)\cdot l\ \mathsf{d}l\ - \hat{l}\cdot\int_{\hat{l}}^{\infty} p(l)f(l)\ \mathsf{d}l\right) \\
&= \int_0^{\hat{l}} p(l)f(l)\ \mathsf{d}l\ +\ p(\hat{l})f(\hat{l})\cdot\hat{l}\ - p(\hat{l})f(\hat{l})\cdot\hat{l} \\
&\quad - p(\hat{l})f(\hat{l})\cdot\hat{l} - \int_{\hat{l}}^{\infty} p(l)f(l)\ \mathsf{d}l - (-p(\hat{l})f(\hat{l})\cdot\hat{l}) \\
&= \int_0^{\hat{l}} p(l)f(l)\ \mathsf{d}l\quad - \int_{\hat{l}}^{\infty} p(l)f(l)\ \mathsf{d}l
\end{aligned}$$

Enfim, temos a equação:

$$\frac{\mathsf{d}}{\mathsf{d}\hat{l}}\left(\text{custo}(\hat{l})\right) = \int_0^{\hat{l}} p(l)f(l)\ \mathsf{d}l\quad - \int_{\hat{l}}^{\infty} p(l)f(l)\ \mathsf{d}l \tag{3-1}$$

Como o mínimo é único, deve estar perto do mínimo da função custo discreta, como pode ser visto na Figura 3.2.

O valor ótimo de $\hat{l}$ que minimiza o custo é então a mediana dos níveis das folhas da 2^{d}–*tree* aberta ponderada pelo número de vezes que foram procuradas, logo $\hat{l}$ deve satisfazer $\sum_{l<\hat{l}} p_l \cdot f_l = \sum_{l>\hat{l}} p_l \cdot f_l$. Essa média pode ser dinamicamente calculada se for mantido um pequeno histograma dos níveis das folhas, podendo também ser aproximado pela média ponderada reduzindo o desprezível tempo de pré-processamento do cálculo da mediana.

3.3.4 Busca de Vizinhos Adjacentes e num Raio

A estimativa do nível para as outras buscas difere da busca direta em apenas um aspecto: o modelo estatístico de $f(l)$ depende do nó n, cujos vizinhos são procurados. Considerando que $p(l_n)$ depende apenas do nível $l_n = l(n)$, logo a distribuição $p(l)$ depende de n, e com isso o ótimo então depende de cada nó n. Para um uso excessivo de uma 2^{d}–*tree* aberta, esse ótimo pode ser calculado para cada nó, e requer apenas um inteiro a mais de armazenamento por nó, embora esse inteiro seja usado apenas no terceiro caso dos algoritmos, onde o

PUC-Rio - Certificação Digital Nº 0410421/CA

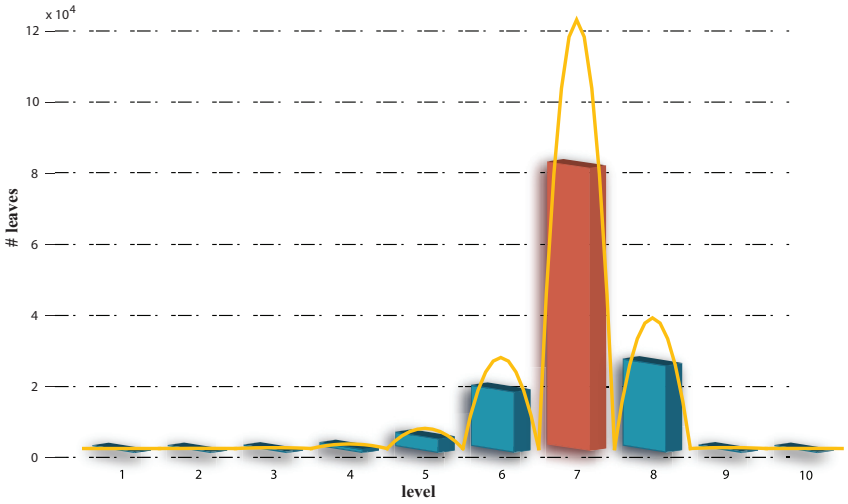


Figura 3.3: A função de probabilidade de densidade $p(l)$ do número de folhas p_l de nível l do modelo *Stanford Bunny*.

vizinho possui um nível de profundidade menor. Para um uso mais simples, pode ser considerado que as distribuições de $p(l_n)$ e $p(l)$ são independentes. Portanto, a aproximação do nível ótimo para todos os nós é feita pelo nível ótimo da busca direta.

PUC-Rio - Certificação Digital Nº 0410421/CA

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

PUC-Rio - Certificação Digital Nº 0410421/CA

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.

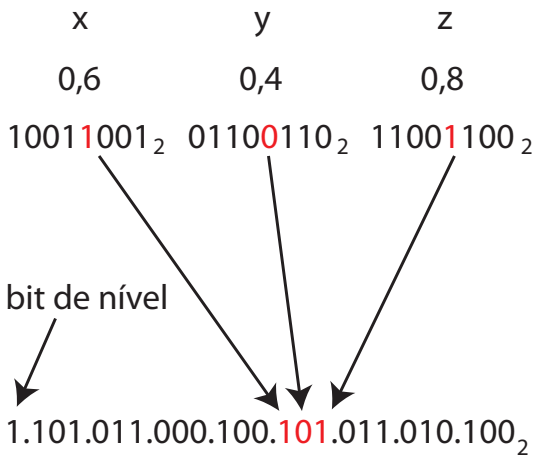


Figura 4.1: Intercalação dos bits para a geração da chave de um ponto 3D com precisão de 8 bits.

4.1.2 Codificando uma 2^d–*tree*

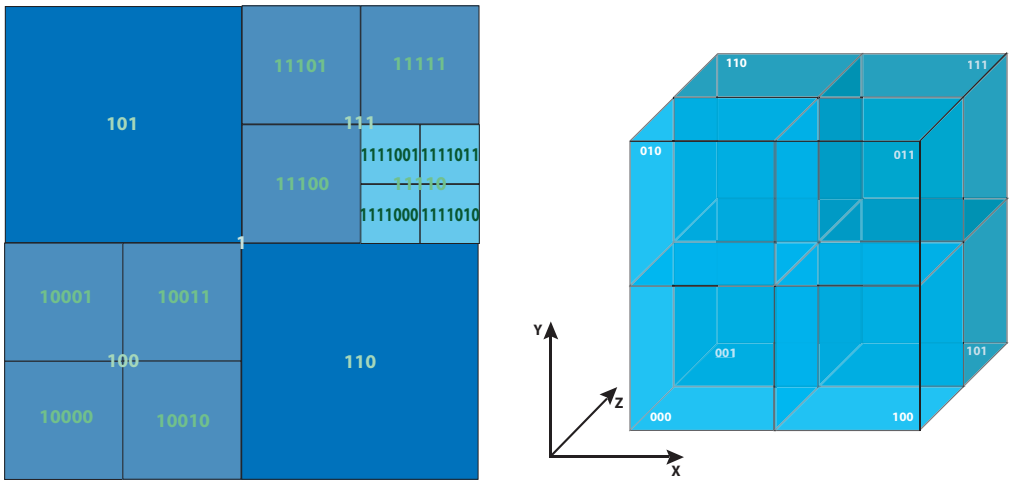
A chave $k(n)$ de um nó n pode ser gerada recursivamente a partir da hierarquia da 2^d–*tree*: a chave do nó raiz é 1, a chave dos filhos de um nó n qualquer é a concatenação da sua chave $k(n)$ com os d bits do código direcional dos filhos: $(k(n) \ll d) \mid (\text{código da orientação do filho})_{\text{d bits}}$. 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

PUC-Rio - Certificação Digital Nº 0410421/CA

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*.

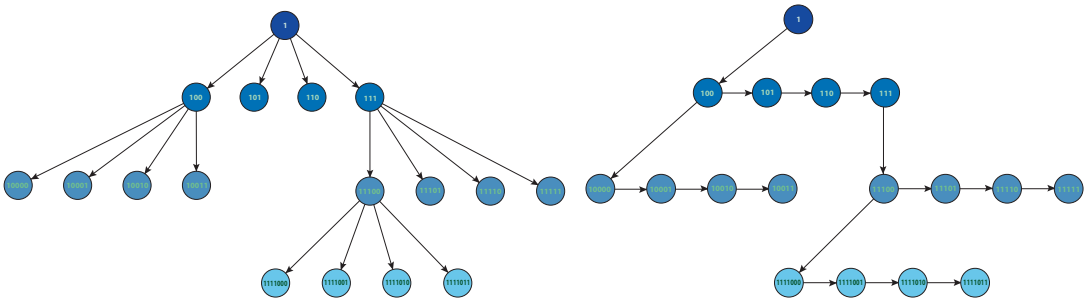
4.1.3 Função de *Hash*

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:

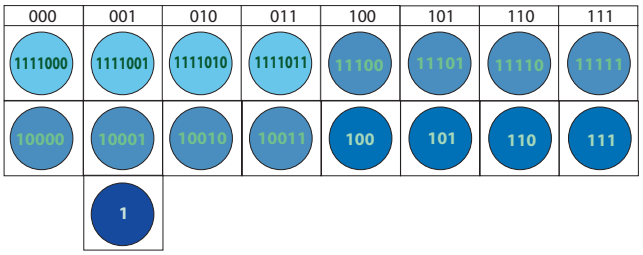
$$h(n) = k(n) \mod 2^h.$$

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).

PUC-Rio - Certificação Digital Nº 0410421/CA



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: $k(n_{adj}) = k(n) \oplus_{Morton} \Delta(k)$.

PUC-Rio - Certificação Digital Nº 0410421/CA

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 $(b_m^i, \ldots, b_1^i)_2 = \lfloor x_i \times 2^m \rfloor$.

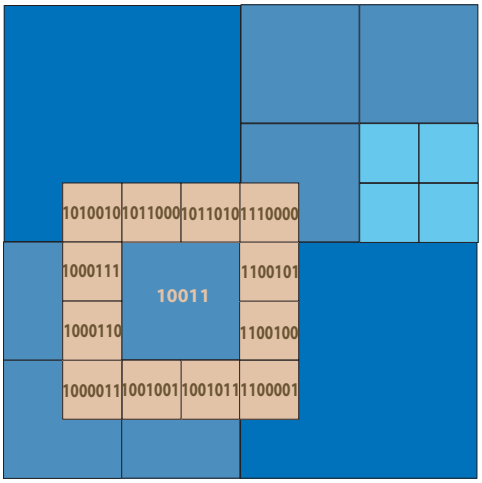


Figura 4.4: Chaves dos nós adjacentes de mesmo nível.

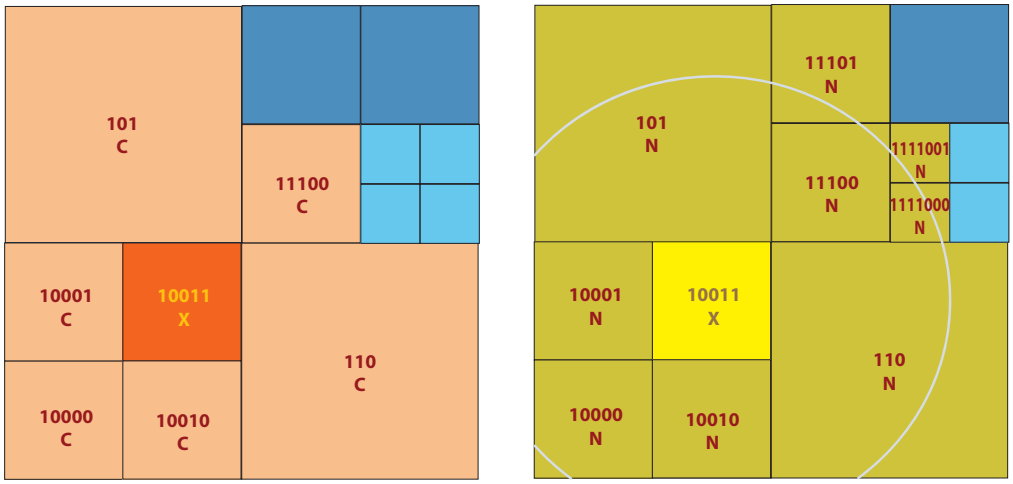
4.2 Implementando as Buscas

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.

PUC-Rio - Certificação Digital Nº 0410421/CA



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.

PUC-Rio - Certificação Digital Nº 0410421/CA

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

PUC-Rio - Certificação Digital Nº 0410421/CA

5
Operações Eficientes em Inteiros

Como as chaves são números inteiros, algumas operações em inteiros podem ser usadas para acelerar a geração, a manipulação e a conversão de chaves. Dilatação, contração e código adicional local são algumas das operações em inteiros usadas na geração, na manipulação e na conversão de chaves. Schrack em (20) dá uma boa definição de inteiros dilatados e define, além de outras coisas, operações de adição e subtração em inteiros dilatados em *quadtrees lineares*. Com tais operações ele define a operação de adição em códigos locais (seção 2.4.2) em *quadtrees lineares* para encontrar o código local dos nós adjacentes de mesmo nível. Este capítulo apresenta de forma geral inteiros dilatados e contraídos, operações eficientes em inteiros e operações em códigos de Morton, tudo para uma 2^d–*tree* num espaço d-dimensional, utilizando a mesma notação de operações de máquina do capítulo anterior.

5.1
Dilatação e Contração

Dilatação e contração de inteiros são operações usadas em conjunção com sistemas de indexação do espaço, auxiliando na conversão entre a chave e a posição espacial de um elemento multidimensional. Dilatação de inteiros é o processo de inserção de zeros no conjunto de bits que compõe o inteiro na base binária, dependendo do tipo da ordenação espacial. Contração de inteiros é o processo de remoção desses zeros. Esses zeros são inseridos ou removidos através de operações binárias nos inteiros, para o processo de intercalar ou concatenar o inteiro, com o objetivo de formar uma chave. Essas operações dependem da dimensão do espaço e também da precisão da conversão do número para a base binária. Existem muitas maneiras de efetuar tais operações nos inteiros: uma maneira simples, mas não eficiente, é operar cada bit do número inteiro por vez, a qual precisaria de muitas operações. Stocco e Schrack em (19) apresentaram uma maneira muito eficiente de dilatar e contrair inteiros para uma *quadtree linear*. Essa mesma idéia será descrita aqui, mas de forma mais geral, e para uma 2^d–*tree* num espaço multidimensional.

PUC-Rio - Certificação Digital Nº 0410421/CA

5.1.1 Definições

Seja um inteiro $q = (q_r q_{r-1} \ldots q_1 q_0)_2$ que representa a conversão na base binária com precisão r de um número pertencente ao intervalo [0, 1] onde $q_i = 0$ ou $1 \ \forall i = 0, \ldots, r$. O inteiro $q_{dil} = (q_r 0^{d-1} q_{r-1} 0^{d-1} \ldots 0^{d-1} q_1 0^{d-1} q_0)_2$ é chamado de *dilatação* de q. O inverso dessa operação, que é a remoção dos elementos 0^{d-1} de um inteiro dilatado resultando no inteiro original, é chamada de *contração*. Obviamente, as operações $q_{dil} = dil(q)$ e $q = con(q_{dil})$ são operações inversas uma da outra, isto é, $q_{dil} = dil(con(q_{dil}))$ e $q = con(dil(q))$.

Os processos de dilatação e contração envolvem inteiros auxiliares, chamados de *máscaras*, que são usados para filtrar elementos inconvenientes resultantes das operações usadas para dilatar ou contrair. Necessita também de posições auxiliares, chamadas de *deslocamentos*, que são posições estratégicas para posicionar partes do inteiro original a ser filtrado.

5.1.2 Máscaras

As máscaras usadas para a dilatação e contração de um inteiro que representa a conversão na base binária de um número do hipercubo d-dimensional são da seguinte forma:

1. A primeira máscara possui 1 nas posições 0, d, $2d$, ..., jd, ... até completar o armazenamento e zero nas outras.

2. A segunda máscara possui 1 nas posições 0 até d, d^2 até $d^2 + d$, $2d^2$ até $2d^2 + d$, ..., jd^2 até $jd^2 + d$, ... , até completar o armazenamento e zero nas outras.

3. A i-ésima máscara possui 1 nas posições 0 até d^{i-1}, d^i até $d^i + d^{i-1}$, $2d^i$ até $2d^i + d^{i-1}$, ..., jd^i até $jd^i + d^{i-1}$, ... , até completar o armazenamento e zero nas outras.

A seqüência de máscaras segue até a m-ésima máscara, interrompendo-se quando d^m for maior que a quantidade de armazenamento. Por exemplo, para um armazenamento de 64 bits, no caso 2D temos seis máscaras, em 3D temos quatro máscaras e em 4D são apenas três máscaras. Essas máscaras estão escritas no formato hexadecimal na tabela abaixo:

PUC-Rio - Certificação Digital Nº 0410421/CA

2D	3D	4D
0x5555555555555555	0x9249249249249249	0x1111111111111111
0x3333333333333333	0x01C0E070381C0E07	0x000F000F000F000F
0x0F0F0F0F0F0F0F0F	0x7FC0000FF80001FF	0x0000000000000FFFF
0x00FF00FF00FF00FF	0x0000000007FFFFFF	
0x0000FFFF0000FFFF		
0x00000000FFFFFFFF		

5.1.3 Deslocamentos

Para as operações de dilatação e contração, antes de aplicar as máscaras é preciso replicar partes do inteiro, estrategicamente posicionadas. Assim, quando for filtrado pelas máscaras, devem aparecer as partes do inteiro nas posições desejadas. Tais posições auxiliares dependem da dimensão d do espaço e da precisão r do inteiro convertido na base binária. Conhecidas essas grandezas, existem $m = \lceil \log_d r \rceil$ grupos de deslocamento contendo $s = \lceil \frac{r}{d^{m-1}} \rceil - 1$ posições. São eles:

1. Primeiro grupo são as posições:
   $(d-1)$, $2(d-1)$, ..., $s(d-1)$.

2. Segundo grupo são as posições:
   $(d-1)d$, $2(d-1)d$, ..., $s(d-1)d$.

3. i-ésimo grupo são as posições:
   $(d-1)d^{i-1}$, $2(d-1)d^{i-1}$,..., $s(d-1)d^{i-1}$.

4. m-ésimo grupo são as posições:
   $(d-1)d^{m-1}$, $2(d-1)d^{m-1}$, ..., $s(d-1)d^{m-1}$.

Por exemplo, no caso 2D para um inteiro com precisão 32 bits temos cinco grupos com apenas uma posição; já no caso 3D para um inteiro com precisão 21 bits temos três grupos com duas posições cada; e no caso 4D para um inteiro com precisão 16 bits são dois grupos com três posições cada. Esses grupos e suas posições encontram-se na tabela abaixo:

2D	3D		4D		
1					
2	2	4			
4	6	12	3	6	9
16	18	36	12	24	36
32					

PUC-Rio - Certificação Digital Nº 0410421/CA

5.1.4 Algoritmos

Algorithm 3 dil(q): faz a dilatação do inteiro q.

```
1: static d                                   // dimensão do espaço
2: static r                                   // precisão da conversão no inteiro
3: static m = ⌈log_d r⌉                       // número de máscaras menos um
4: static s = ⌈r/d^(m-1)⌉ − 1                 // número de deslocamentos
5: static shift[m+1][s]                       // vetor do grupo de deslocamentos
6: static mask[m+1]                           // vetor de máscaras
7: for j = m a 1 do
8:    for i = 1 a s do
9:       q |= (q ≪ shift[j][i])
10:   end for
11:   q &= mask[j]
12: end for
13: return q
```

Dado um inteiro que representa a conversão de precisão r de um número pertencente ao hipercubo d-dimensional, são necessários pelo menos $r * d$ bits para armazenar a sua dilatação. Então o número de bits n_{bits} usado para o armazenamento deve ser maior ou igual a $r * d$: $n_{bits} \geq r * d$. Para tal armazenamento, são usadas nas operações de dilatação e contração $m + 1$ máscaras e m grupos de deslocamento contendo s posições, como definido acima. Para dilatar são usadas as primeiras m máscaras da seguinte forma: primeiramente é aplicado o m-ésimo grupo de deslocamentos no inteiro e depois este é filtrado com a m-ésima máscara. Depois é aplicado o $(m-1)$-ésimo grupo de deslocamentos no inteiro e depois este é filtrado com a $(m-1)$-ésima máscara, e assim por diante, até aplicar o primeiro grupo de deslocamentos, e filtrando-o com a primeira máscara. O algoritmo 3 descreve em detalhes esse procedimento. Observe que s foi definido de forma que satisfaça as desigualdades: $(s+1)d^{m-1} < r \leq (s+2)d^{m-1}$. Veja o que acontece com o inteiro no processo de dilatação:

1. Inteiro Original:
   $1\ 0 \ldots 0\ b_r \ldots b_1$.

2. Passo 1: $[g = (d-1)d^{m-1}]$
   $1\ 0 \ldots 0\ b_{d^r} \ldots b_{(s+1)d^{m-1}+1} \ldots 0^g b_{2d^{m-1}} \ldots b_{d^{m-1}+1}\ 0^g b_{d^{m-1}} \ldots b_1$.

3. Passo 2: $[g = (d-1)d^{m-2}]$
   $1\ 0 \ldots 0\ b_{d^r} \ldots b_{(s+1)d^{m-1}+(d-1)d^{m-2}+1} \ldots 0^g b_{d^{m-1}+d^{m-2}} \ldots b_{d^{m-1}+1}$
   $0^g b_{d^{m-1}} \ldots b_{(d-1)d^{m-2}+1} \ldots 0^g b_{2d^{m-2}} \ldots b_{d^{m-2}+1}\ 0^g b_{d^{m-2}} \ldots b_1$.

PUC-Rio - Certificação Digital Nº 0410421/CA

4. Passo i: $[g = (d-1)d^{m-i}]$
   $1\ 0 \ldots 0\ b_{d^r} \ldots b_{(s+1)d^{m-i}+(d-1)d^{m-i-1}+1} \ldots 0^g b_{d^{m-i}+d^{m-i-1}} \ldots b_{d^{m-i}+1}$
   $0^g b_{d^{m-i}} \ldots b_{(d-1)d^{m-i-1}+1} \ldots 0^g b_{2d^{m-i-1}} \ldots b_{d^{m-i-1}+1}\ 0^g b_{d^{m-i-1}} \ldots b_1.$

5. Passo m: $[g = (d-1)]$
   $1\ 0 \ldots 0\ b_{d^r} \ldots 0^g b_{(s+1)d^{m-1}} \ldots 0^g b_{d^2}\ 0^g b_1.$

Para contrair são usadas as últimas m máscaras da seguinte forma: primeiramente é aplicado o primeiro grupo de deslocamentos no inteiro e depois este é filtrado com a segunda máscara. Depois é aplicado o segundo grupo de deslocamentos no inteiro e depois este é filtrado com a terceira máscara, e assim por diante, até aplicar o m-ésimo grupo de deslocamentos, filtrando-o com a $m+1$-ésima máscara. O algoritmo 4 descreve em detalhes esse procedimento.

Algorithm 4 con(q_{dil}): faz a contração do inteiro dilatado q_{dil}.

```
1: static d                                          // dimensão do espaço
2: static r                                          // precisão da conversão no inteiro
3: static m = ⌈log_d r⌉                              // número de máscaras menos um
4: static s = ⌈r/d^{m-1}⌉ − 1                        // número de deslocamentos
5: static shift[m+1][s]                              // vetor do grupo de deslocamentos
6: static mask[m+1]                                  // vetor de máscaras
7: for j = 2 a m + 1 do
8:    for i = 1 a s do
9:       q_dil |= (q_dil ≫ shift[j][i])
10:   end for
11:   q_dil &= mask[j]
12: end for
13: return q_dil
```

A Figura 5.1 exemplifica as etapas do processo de dilatação, e na ordem contrária o processo de contração. A Figura 5.1(a) mostra em 2D, para uma precisão 8; já a Figura 5.1(b) mostra em 3D, para uma precisão 9; e a Figura 5.1(c) mostra em 4D, para uma precisão 8, onde os espaços vazios são zeros e as letras gregas simbolizam os bits da conversão.

5.1.5 Acelerando o Processo de Intercalação

O processo de intercalação dos bits para a geração da chave pelo método de códigos de Morton pode ser feito usando as operações de dilatação e contração em inteiros da seguinte forma: 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 $k(p)$ é gerada a partir das conversões de precisão m das coordenadas

PUC-Rio - Certificação Digital Nº 0410421/CA

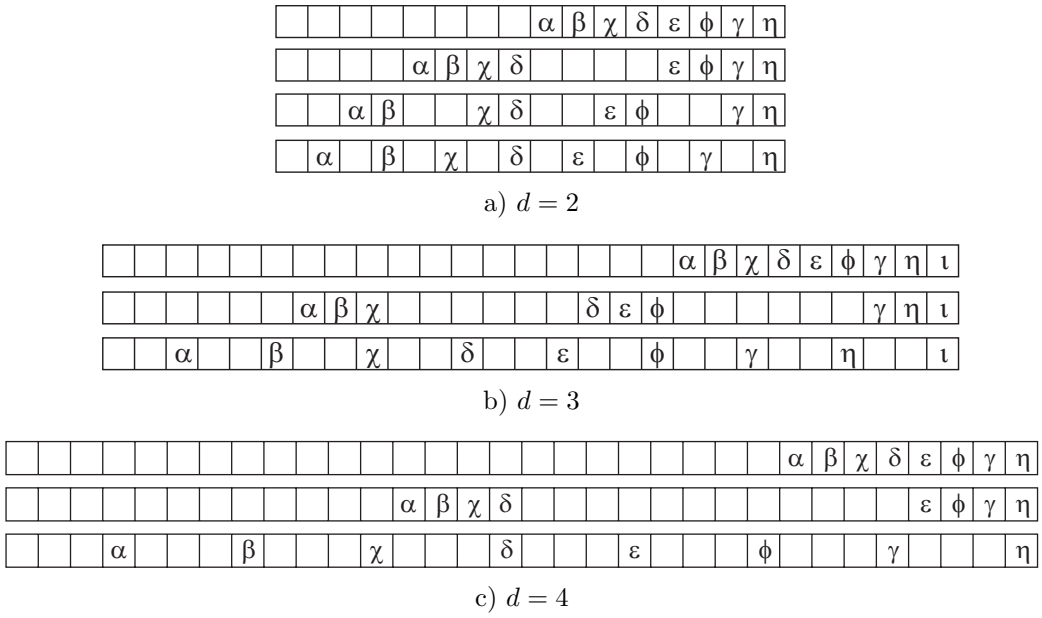


Figura 5.1: Etapas de dilatação e contração de inteiros.

de p na base binária $b^i = (b^i_m, \ldots, b^i_1)_2 = \lfloor x_i \times 2^m \rfloor$ para $i = 1, \ldots, d$ e usando operações de dilatação:

$$k(p) = (1b^d_m \ldots b^1_m \ldots b^d_2 \ldots b^1_2 \ldots b^d_1 \ldots b^1_1)_2 = \bigmid_{i=1}^{d} \big(dil(b^i) \ll (i-1)\big)$$

e o processo inverso de obter as coordenadas do ponto p com uma precisão m a partir de sua chave $k(p)$ utiliza operações de contração:

$$x_i = \frac{1}{2^m} \times con(k(p) \gg (i-1)).$$

5.2 Adição em Inteiros Dilatados

A operação de adição em inteiros dilatados evita conversões desnecessárias quando se está operando um inteiro. Operações em inteiros dilatados devem resultar também em inteiros dilatados, e para isso é preciso separar e filtrar os bits importantes do inteiro original após o uso dessas operações. Para isso, é preciso definir máscaras binárias para auxiliar essas operações. Dada uma precisão r, tais máscaras binárias são dadas por r blocos de tamanho d cada um, que é o tamanho de um inteiro dilatado de precisão r: $t_1 = (0\ldots01)^r, t_2 = (0\ldots10)^r, \ldots, t_{d-1} = (010\ldots0)^r$ e $t_d = (10\ldots0)^r$. As máscaras binárias opostas (negação) às anteriores que também auxiliam a operação de adição, basta trocar 0 por 1 e 1 por 0 nas máscaras anteriores: $\tilde{t}_1 = (1\ldots10)^r, \tilde{t}_2 = (1\ldots01)^r, \ldots, \tilde{t}_{d-1} = (101\ldots1)^r$ e $\tilde{t}_d = (01\ldots1)^r$, é fácil

PUC-Rio - Certificação Digital Nº 0410421/CA

ver que $t_i = t_{i-1} \ll 1$ e $\tilde{t}_i = \tilde{t}_{i-1} \ll 1$. Um caso particular é o espaço bidimensional, pois existem apenas duas máscaras, sendo uma oposta à outra. É mais intuitivo denotar a primeira máscara por t_x ao invés de t_1, a segunda máscara por t_y ao invés de t_2, a terceira máscara por t_z ao invés de t_3 e a quarta máscara por t_w ao invés de t_4, para dimensões menores. Veja as chaves no formato hexadecimal na tabela abaixo:

2D
$t_x = \tilde{t}_y$=0x5555555555555555
$t_y = \tilde{t}_x$=0xAAAAAAAAAAAAAAAA

3D
t_x=0x4924924924924924
t_y=0x2492492492492492
t_z=0x9249249249249249
$\tilde{t}_x$=0xB6DB6DB6DB6DB6DB
$\tilde{t}_y$=0xDB6DB6DB6DB6DB6D
$\tilde{t}_z$=0x6DB6DB6DB6DB6DB6

4D
t_x=0x1111111111111111
t_y=0x3333333333333333
t_z=0x4444444444444444
t_w=0x8888888888888888
$\tilde{t}_x$=0xEEEEEEEEEEEEEEEE
$\tilde{t}_y$=0xDDDDDDDDDDDDDDDD
$\tilde{t}_z$=0xBBBBBBBBBBBBBBBB
$\tilde{t}_w$=0x7777777777777777

A operação de adição em inteiros dilatados q_{dil} e p_{dil} de mesma precisão r é definida como

$$q_{dil} \oplus p_{dil} = \big((q_{dil} \mid \tilde{t}_1) + p_{dil}\big) \ \& \ t_1$$

onde o operador $\oplus$ representa a adição de inteiros dilatados. A adição é definida de forma que a soma $s_{dil} = q_{dil} \oplus p_{dil}$ respeite o princípio de unicidade da dilatação, preservando o inteiro original após operá-lo de forma dilatada. Então a soma deve satisfazer $con(s_{dil}) = q + p$ ou então $s_{dil} = dil(q + p)$. Um exemplo no espaço bidimensional: sejam os inteiros 7 e 6, suas conversões na base binária de precisão 4 são $7_2 = 0111$ e $6_2 = 0110$. Seja 10 o número de bits usado para a dilatação, então $dil(7) = 0000010101$ e $dil(6) = 0000010100$. Com as máscaras $t_x = \tilde{t}_y = 0101010101$ e $t_y = \tilde{t}_x = 1010101010$, temos $dil(7) \mid t_y = 1010111111$, logo $(dil(7) \mid \tilde{t}_1) + dil(6) = 1011010011$ e então $dil(7) \oplus dil(6) = ((dil(7) \mid t_y) + dil(6)) \ \& \ t_x = 0001010001 = dil(13)$.

5.3 Adição em Códigos de Morton

A adição em chaves geradas pelo método códigos de Morton pode ser feita separando as adições de cada coordenada das chaves e recombinando os resultados em uma nova chave resultante da soma das duas primeiras. Inspirado na operação de adição em inteiros dilatados, dadas as chaves k e h geradas pelo

PUC-Rio - Certificação Digital Nº 0410421/CA

método códigos de Morton num espaço d-dimensional de mesmo nível, a soma delas é:

$$k \underset{Morton}{\oplus} h = \bigmid_{i=1}^{d} (\{[(k \,\&\, t_i) \gg (i-1)] \oplus [(h \,\&\, t_i) \gg (i-1)]\} \ll (i-1))$$

onde o operador $\oplus_{Morton}$ representa a adição de chaves geradas pelo método códigos de Morton e o operador $\oplus$ representa a adição de inteiros dilatados. A parcela $(k \,\&\, t_i) \gg (i-1)$ remove da chave os bits relativos à coordenada i, montando um inteiro dilatado, para que possa ser adicionado com o outro inteiro dilatado da outra chave. A parcela final $\ll (i-1)$ recoloca o resultado da soma dos inteiros dilatados na posição certa relativa à coordenada i para a geração da chave resultante da soma que ainda é uma chave gerada pelo método de códigos de Morton. Substituindo a adição de inteiros dilatados temos a seguinte simplificação:

$$k \underset{Morton}{\oplus} h = \bigmid_{i=1}^{d} \left(\left[(k \mid \tilde{t}_i) + (h \,\&\, t_i) \right] \,\&\, t_i \right).$$

Para verificar tal simplificação note que na parcela $[(k \,\&\, t_i) \gg (i-1)] \mid \tilde{t}_1$ os bits relativos à i-ésima coordenada estão na primeira posição dos blocos de tamanho d da chave, e o resto preenchido por uns, e para a parcela $(k \mid \tilde{t}_i)$ os bits relativos à i-ésima coordenada estão na i-ésima posição dos blocos de tamanho d da chave, e o resto preenchido por uns, dispensando assim o uso de operações de deslocamento.

5.4 Chaves Adjacentes

Um nó pertencente a uma 2^{d}–*tree* num espaço d-dimensional, possui 3^d-1 direções de adjacência, são elas: $\Delta v = (v_1, \ldots, v_d)$ com $v_i \in \{-1, 0, 1\}$ $\forall i = 1, \ldots, d$ e com $v_i \neq 0$ para algum $i \in \{0, \ldots, d\}$. A tabela abaixo mostra essas direções no espaço bidimensional:

NO	N	NL
O	X	L
SO	S	SL

(-1, 1)	(0, 1)	(1, 1)
(-1, 0)	X	(1, 0)
(-1,-1)	(0,-1)	(1,-1)

Torna-se possível o cálculo em tempo constante das chaves dos nós adjacentes a um nó a partir de sua chave, utilizando-se a operação de adição de chaves geradas pelo método de códigos de Morton, satisfazendo assim o terceiro requerimento da seção 4.1.4. Para isto, basta adicionar à chave do nó uma das $3^d - 1$ chaves de mesmo nível que representam as direções

PUC-Rio - Certificação Digital Nº 0410421/CA

adjacentes. Dada a chave $k(n)$ de um nó n de uma 2^{d}–*tree*, a chave $k(n_{adj})$ de algum dos nós adjacentes n_{adj} de mesmo nível é gerada da seguinte forma: $k(n_{adj}) = k(n) \oplus_{Morton} \Delta k$, onde Δk simboliza a chave gerada no mesmo nível que n pelo método de códigos de Morton de alguma das $3^d - 1$ direções de adjacência $\Delta k = k(\Delta v)$, fazendo uma intercalação das coordenadas da direção de adjacência, as quais assumem apenas os valores $-1 = 11\ldots11_2$, $0 = 00\ldots00_2$ e $1 = 00\ldots01_2$.

Em 2D temos 8 direções adjacentes. As chaves de tamanho 64 bits dessas direções no formato hexadecimal na tabelas abaixo:

2D	
(-1,-1)=0xFFFFFFFFFFFFFFFF	(0, 1)=0x0000000000000001
(-1, 0)=0xAAAAAAAAAAAAAAAA	(1,-1)=0x5555555555555557
(-1, 1)=0xAAAAAAAAAAAAAAAB	(1, 0)=0x0000000000000002
(0,-1)=0x5555555555555555	(1, 1)=0x0000000000000003

Em 3D temos 26 direções adjacentes. As chaves de tamanho 64 bits dessas direções no formato hexadecimal na tabela abaixo:

3D	
(-1,-1,-1)=0xFFFFFFFFFFFFFFFF	(0, 0, 1)=0x0000000000000001
(-1,-1, 0)=0x6DB6DB6DB6DB6DB6	(0, 1,-1)=0x924924924924924B
(-1,-1, 1)=0x6DB6DB6DB6DB6DB7	(0, 1, 0)=0x0000000000000002
(-1, 0,-1)=0xDB6DB6DB6DB6DB6D	(0, 1, 1)=0x0000000000000003
(-1, 0, 0)=0x4924924924924924	(1,-1,-1)=0xB6DB6DB6DB6DB6DF
(-1, 0, 1)=0x4924924924924925	(1,-1, 0)=0x2492492492492496
(-1, 1,-1)=0xDB6DB6DB6DB6DB6F	(1,-1, 1)=0x2492492492492497
(-1, 1, 0)=0x4924924924924926	(1, 0,-1)=0x924924924924924D
(-1, 1, 1)=0x4924924924924927	(1, 0, 0)=0x0000000000000004
(0,-1,-1)=0xB6DB6DB6DB6DB6DB	(1, 0, 1)=0x0000000000000005
(0,-1, 0)=0x2492492492492492	(1, 1,-1)=0x924924924924924F
(0,-1, 1)=0x2492492492492493	(1, 1, 0)=0x0000000000000006
(0, 0,-1)=0x9249249249249249	(1, 1, 1)=0x0000000000000007

PUC-Rio - Certificação Digital Nº 0410421/CA

6
Resultados

Este capítulo descreve os resultados das otimizações descritas na seção 3.3, apresentando apenas testes feitos com $d = 3$, isto é, no espaço tridimensional. Comparações com uma implementação clássica de uma *octree* foram feitas para diferentes formas de busca. Modelos de teste, modelos de busca e modelos de comparação são descritos abaixo. A 2^{d}–*tree* aberta usada nos testes para validar as otimizações foi descrita de forma geral nas seções 4.1 e 2.5.3. Os testes foram rodados numa máquina com uma configuração contendo um Pentium IV de $3GHz$ e 2 Gb de RAM.

PUC-Rio - Certificação Digital Nº 0410421/CA

6.1
Modelo de Teste

Foram testadas as otimizações propostas na seção 3.3 em uma *hashed octree* adaptada num conjunto de pontos da seguinte forma: os nós são subdivididos até conterem apenas um ponto do conjunto ou se a *hashed octree* atingir um nível máximo, que foi estipulado como 21, devido ao uso de armazenamento de chaves de tamanho 64-bits no espaço tridimensional. A função de *hash* usada foi a mesma para todos os modelos e para todos os tipos de busca retornando os 21 bits menos significantes da chave. Foi constatado que tal função de *hash* tende a distribuir de maneira eficiente as chaves na *hash table*, mesmo para dados propícios a desbalancear uma *octree* clássica.

6.2
Modelo de Comparação

Os resultados foram comparados com os procedimentos de busca clássicos (seção 2.3) em um armazenamento clássico na forma de árvore, mas em duas implementações diferentes: uma implementação clássica (oito ponteiros) e outra implementação filho/irmão (seção 2.5.2). Os resultados foram também comparados com um armazenamento na forma de uma *hash table* (seção 2.5.3), mas sem as otimizações propostas na seção 3.3. Foi verificado que buscas num armazenamento na forma de uma *hash table* sem otimização são mais lentas,

	f_{pontos}	f_{folhas}	f_{grade}
Busca por Pontos	**1.30**	1.33	2.45
Busca por Folhas	0.95	**0.93**	1.52
Busca numa Grade	1.70	1.66	**0.89**

Tabela 6.1: Média em todos os conjuntos de pontos do tempo de execução (em microssegundos) da busca direta, simulando diferentes modelos de freqüência (linhas). Otimizando de acordo com a freqüência correta do modelo (termos diagonais) melhora-se o método de busca 2% (pontos) e 92% (grade). O número de buscas em cada caso está descrito na tabela 6.3.

já que o acesso ao filho de um nó exige um acesso à *hash table*, que é mais lento do que o acesso direto ao filho de um nó via ponteiros.

6.3 Modelo de Buscas Diretas

Três tipos de busca diretas foram usados para variar a freqüência de acesso *a priori* f_n com que uma folha de nível n é buscada, como descrito na seção 3.3, a fim de testar o modelo de custo em diferentes cenários. Foram usados três modelos de freqüência: buscando por cada ponto do conjunto, onde a freqüência de cada folha é definida como sendo a quantidade de pontos dentro dela: ($f_{pontos}(n) = \#\{pontos \in n\}$); buscando por cada folha da *octree*, onde a freqüência é definida como sendo 1 para cada folha: ($f_{folhas}(n) = 1$); e por cada célula da grade regular 127^3, criada na região definida pela *octree* e buscada cada célula da grade uma vez: ($f_{grade}(n) = 2^{3(7-l(n))}$). Essas otimizações estão experimentalmente validadas na tabela 6.1: as buscas foram significantemente rápidas onde foi escolhida a freqüência correta do modelo para a otimização.

6.4 Modelo de Objetos Geométricos

Os objetos geométricos usados para os testes são classificados em quatro categorias: densidade altamente variável, a qual tende a linearizar o tempo das buscas numa *octree* clássica; dados volumétricos conhecidos que preenchem todo o domínio; dados randômicos, seguindo uma distribuição uniforme dos dados; superfícies de grande extensão. Para dados de densidade altamente variável, as *octrees* de ponteiros têm uma tendência a ficar desbalanceadas, e por isso os tempos de busca são bem piores do que para um modelo do mesmo tamanho melhor distribuído. Devido à função de *hash* escolhida, isso não acontece com a *hashed octree*. Ela não fica desbalanceada, e o tempo de busca para alguns modelos se comportou mais de dez vezes mais rápido do que a *octree* clássica (modelo *Maracanã* na 6.2). Para dados volumétricos, que são

PUC-Rio - Certificação Digital Nº 0410421/CA

distribuídos no domínio todo, a *hashed octree* se comportou na média quatro vezes mais rápida do que a *octree* clássica. E em dados grandes de superfícies, como, por exemplo, o dado *Happy Budda*, a *hashed octree* foi mais de três vezes mais rápida.

Para modelos muito grandes, como, por exemplo, o modelo *Thai Statue* que possui 5.000.000 pontos, onde foram gerados 22.296.000 nós na *octree*, a *hashed octree* encontrou um pouco de dificuldade, ganhando por pouco em tempo. Isso acontece por causa do grande número de nós gerados, exigindo uma lista de colisão na *hash table* com vários níveis, com o fator de balanço aproximadamente $\frac{1}{10}$, pois o número de entradas da *hash table* é de 2.097.152 para uma função de *hash* retornando os 21 bits menos significantes, fazendo com que a busca deixe de ser tão eficiente devido a uma grande parte dela conter acessos seqüenciais da lista de colisão. Seria necessário aumentar o número de bits de retorno da função de *hash* para aumentar o número de entradas e assim diminuir alguns níveis da lista de colisão, a fim de evitar a influência do acesso seqüencial nas buscas. Mas note que a memória consumida pela *hashed octree* para modelos grandes foi metade da consumida pela *octree* clássica e a mesma quantidade consumida pela *octree* filho-irmão, como por exemplo, o modelo *Thai Statue*: a *octree* clássica gastou 1.070 MB de memória para armazenar os nós da *octree*, enquanto a *octree* filho-irmão e a *hashed octree* gastaram 535 MB. Para esse objeto, o método chegou num extremo da relação de eficiência entre tempo de busca e memória consumida, gastando metade da memória e ainda assim ganhando de 0.3 vezes no tempo de busca, comparado com a *octree* clássica. Agora, se comparado com a *octree* filho-irmão, as duas consomem a mesma quantidade de memória, mas a *hashed octree* é quase seis vezes mais rápida na busca direta por pontos. Com as otimizações feitas na *hashed octree*, o usuário controla a relação de eficiência entre tempo de busca e memória consumida pela função de *hash*. Todos os dados podem ser conferidos na tabela 6.2. E para os níveis e tempos de pré-processamento, conferir a tabela 6.3.

PUC-Rio - Certificação Digital Nº 0410421/CA

PUC-Rio - Certificação Digital Nº 0410421/CA

Conjunto de Pontos	**Busca Direta**				**Busca por Vizinhos Adjacentes**				**Busca por Vizinhos num Raio**			
	8_{ptr}	**F-I**$_{ptr}$	**Desopt**	**Nosso**	8_{ptr}	**F-I**$_{ptr}$	**Desopt**	**Nosso**	8_{ptr}	**F-I**$_{ptr}$	**Desopt**	**Nosso**
Sphere	2.17	7.24	11.47	0.58	23.10	23.85	41.55	7.63	28.38	25.39	41.49	0.71
Bunny	2.86	11.28	18.67	0.75	38.34	35.93	59.70	8.70	36.70	35.44	59.26	0.73
Maracanã	6.53	31.21	53.47	0.60	58.98	55.79	95.70	9.12	77.75	76.31	136.91	33.72
David head	3.29	12.98	21.27	0.64	43.57	40.88	68.69	9.43	42.61	40.93	70.30	1.39
Pig	3.71	15.40	25.65	0.71	46.46	43.90	74.64	11.73	53.53	51.59	91.88	10.67
CSG	3.30	13.34	21.98	0.79	40.60	37.75	63.22	9.76	38.79	37.18	62.81	4.57
Volcano	3.28	13.42	22.73	0.71	43.04	40.37	68.74	8.83	40.18	39.43	69.16	2.78
Fighter	3.89	14.82	24.53	1.06	54.70	50.38	86.60	12.88	51.81	49.92	87.79	2.18
Deltao	4.47	18.63	31.64	1.65	49.07	45.56	77.67	11.30	180.30	169.13	332.11	129.28
David	3.58	15.04	26.54	0.87	46.13	42.64	73.64	10.39	43.71	42.17	74.79	11.88
Rnd Sphere	4.50	16.42	27.92	1.59	46.51	42.55	73.15	14.45	42.43	41.42	72.71	9.02
Dragon	3.81	15.88	27.43	1.17	45.53	42.80	74.31	12.79	45.88	44.44	79.03	11.67
Happy	3.80	16.17	28.69	1.21	46.30	43.05	75.37	13.18	48.60	46.39	83.23	15.46
Blade	3.65	15.25	27.98	1.27	44.05	40.89	73.09	13.11	43.84	41.54	75.27	11.45
Chair	3.83	16.38	31.41	2.15	46.56	43.43	80.89	19.34	52.92	50.80	96.60	24.48
Rnd Cube	4.18	14.76	33.54	3.79	51.36	46.69	95.21	27.44	45.34	44.04	84.09	0.96
Thai Statue	3.93	17.38	37.51	3.13	49.99	47.58	94.86	22.43	79.50	77.18	163.46	53.56
média	**3.81**	**15.62**	**27.79**	**1.33**	**45.55**	**42.59**	**75.12**	**13.09**	**56.02**	**53.72**	**98.88**	**19.09**
ganho	× **3.8**	× **15.7**	× **26.9**	× 1	× **3.8**	× **3.6**	× **6.2**	× 1	× **14.5**	× **13.8**	× **24.2**	× 1

Tabela 6.2: Tempo de execução (em microssegundos) de nossas buscas otimizadas, comparada com buscas clássicas de cima para baixo numa *octree* clássica 8-ponteiros (8_{ptr}), numa *octree* filho-irmão (F-I$_{ptr}$) e numa *hashed octree* sem otimização (*Desopt*). Nossa otimização na busca direta retornou 15 vezes mais rápida do que a filho-irmão, e entre 10% (volume randômico) e 1090% mais rápida (*Maracanã*) do que a *octree* clássica. Cada tempo representa a média do tempo das buscas por cada ponto do conjunto. O ganho corresponde à razão do tempo de busca por ponteiro sobre o nosso tempo de busca. O tamanho do raio usado foi de 0.1 da normalização dos dados.

PUC-Rio - Certificação Digital Nº 0410421/CA

Conjunto de Pontos	**# Nós** (x1000)	**Profundidade Nível**	**Memória** $\mathbf{8}_{ptr}$ (MB)	**F-I**$_{ptr}$ (MB)	**Hash** (MB)	**Construção** $\mathbf{8}_{ptr}$ (msec)	**F-I**$_{ptr}$ (msec)	**Hash** (msec)	**Otimização Tempo** (msec)	**Nível**	**Número de Buscas Ponto** (x1000)	**Folha** (x1000)
Sphere	4.4	6	0.2	0.1	45	5.6	5.4	11.1	7.8	4	0.9	3.8
Bunny	144	9	6.9	3.4	50	224	215	187	14.2	7	35	126
Maracanã	190	21	9.1	4.6	51	268	257	230	27.1	21	10	167
David head	395	13	19	9.5	51	726	682	548	32.4	9	100	346
Pig	401	11	19	10	51	772	725	583	38.2	10	100	351
CSG	529	19	25	13	52	819	768	642	50.7	8	82	463
Volcano	551	21	26	13	52	1 043	967	795	50.2	9	133	482
Fighter	723	14	35	17	54	1 584	1 509	1 211	58.6	9	257	632
Deltao	1 210	20	58	29	60	2 466	2 119	1 695	116	12	208	1 058
David	1 534	21	74	37	62	3 124	2 819	2 379	135	10	350	1 342
Rnd Sphere	2 883	18	138	69	82	5 756	5 100	4 053	268	9	500	2 523
Dragon	3 060	21	147	73	85	5 634	5 280	4 286	314	10	438	2 678
Happy	3 617	21	174	87	96	7 227	6 627	5 361	377	10	544	3 165
Blade	5 412	16	260	130	134	11 641	10 079	8 384	518	10	883	4 735
Chair	9 255	18	444	222	223	34 471	24 833	20 076	930	11	1 669	8 098
Rnd Cube	15 675	12	752	376	376	45 867	39 380	31 012	1 202	8	4 000	13 715
Thai Statue	22 296	20	1070	535	535	155 803	106 509	81 446	2 229	12	5 000	19 509
	média ponderada		**632**	**316**	**321**	**68 436**	**49 213**	**38 084**	**354**			

Tabela 6.3: Tempos de construção e pré-processamento (em microssegundos) em diferentes conjuntos de pontos: densidade altamente variável (Maracanã, David head, Pig, Fighter, Deltao), volumétrico (Volcano, Fighter, Deltao, Cube), Randômico (Rnd Sphere, Rnd Cube). A construção da *hashed octree* é ligeiramente mais rápida do que a da *octree* com ponteiros. O pré-processamento consiste do cálculo do nível ótimo e representa menos de um por cento do tempo de construção. As médias são ponderadas pelo tamanho da *octree*. O número de buscas está relacionado com cada ponto do conjunto de acordo com o modelo de freqüência simulado (busca numa grade tem um número constante de buscas 2097).

7
Conclusão e Trabalhos Futuros

Esta tese apresentou um método de otimização estatística de buscas para estruturas hierárquicas do tipo 2^d–*trees*. Essa otimização requer apenas que os nós da estrutura possam ser acessados em tempo constante e indexados por um critério que satisfaça quatro requisitos. Um exemplo completo dessa otimização foi descrito para *hashed* 2^d–*trees*, tendo seus nós armazenados numa *hash table* e indexados pelo método de códigos de Morton. Os quatro requisitos foram provados neste caso. Além disso, operações de inteiros eficientes para melhorar o desempenho dessa estrutura foram apresentadas em dimensão qualquer. O método foi experimentado para $d = 3$, comparado com a representação clássica de uma 2^d–*tree*, o novo método reduz, tanto o tempo de execução por um fator médio de 4 vezes, como o consumo de memória utilizada, em média, metade da memória exigida pela representação clássica. No caso de uma 2^d–*tree* aberta, o método mantém o mesmo consumo de memória, mas melhora, em média, 15 vezes o tempo de execução.

Este trabalho pode ser aprimorado integrando mais ainda a estrutura hierárquica com a *hash table*. Por exemplo, pode-se definir uma maneira de ajustar seu tamanho automaticamente, de acordo com o dado e com a escolha do usuário em dar preferência à memória ou ao tempo de execução. Novas funções de *hash* que distribuam uniformemente as chaves na *hash table* podem ser criadas. Outras funções de *hash* tais como *linear hashing* ou *spiral hashing* deverão ser usadas quando é exigido um comportamento dinâmico da estrutura com exaustivas inserções e remoções. Para um armazenamento em disco, devem ser usadas funções de *hash* que preservem a proximidade das chaves para garantir boa coerência nos *buckets*. No caso do uso de listas encadeadas, critérios podem ser escolhidos para o gerenciamento das listas, definindo as posições por acesso e melhorando o *cache*.

Este trabalho pode ser estendido estudando modelos de custo mais precisos e adaptando-os a outras estruturas de busca, tais como *kd-tree* e o *particionamento binário do espaço*. Neste caso, devem ser definidas indexações compatíveis para cada uma.

PUC-Rio - Certificação Digital Nº 0410421/CA

Referências Bibliográficas

[1] ADELSON-VELSKY, G. M.; LANDIS, E. M.. **An algorithm for the organization of information**. Soviet Math. Dokl, 3:1259–1262, 1962. 2.5.2

[2] AHUJA, N.. **On approaches to polygonal decomposition for hierarchical image representation**. Computer Vision, Graphics, and Image Processing, 24(2):200–214, 1983. 2.1

[3] BELL, S.; DIAZ, B.; HOLROYD, F. ; JACKSON, M.. **Spatially referenced methods of processing raster and vector data**. Image and Vision Computing, 1(4):211–220, 1983. 2.1

[4] BENTLEY, J. L.. **Multidimensional binary search trees used for associative searching**. Commun. ACM, 18(9):509–517, 1975. 1

[5] BORDIGNON, A.; LEWINER, T.; LOPES, H.; TAVARES, G. ; PEREIRA, R.. **Point set compression through BSP quantization**. In: SIBGRAPI, p. 229–236. IEEE, 2006. 2.2.3

[6] CASTRO, R.; LEWINER, T.; LOPES, H.; TAVARES, G. ; BORDIGNON, A.. **Statistical optimization of octree searches**. Computer Graphics Forum, 27(1), 2008. to appear. 1, 3

[7] GARGANTINI, I.. **An effective way to represent quadtrees**. Commun. ACM, 25(12):905–910, 1982. 1, 2.4.2, 3.1

[8] GOODCHILD, M. F.; GRANDFIELD, A. W.. **Optimizing raster storage: an examination of four alternatives**. In: IN PROCEEDINGS OF AUTO-CARTO 6, volumen 1, p. 400–407, Ottawa, October 1983. 2.4.1

[9] HENRY FUCHS, Z. M. K.; NAYLOR, B. F.. **On visible surface generation by a priori tree structure**. In: SIGGRAPH'80 CONFERENCE PROCEEDINGS, SEATTLE, p. 124–133. ACM SIGGRAPH, July 1980. 1

[10] HILBERT, D.. **Ueber stetige abbildung einer linie auf ein flashenstuck**. Mathematishe Annalen, p. 459–460, 1891. 2.4.1

PUC-Rio - Certificação Digital Nº 0410421/CA

[11] HOARE, C. A. R.. **Notes on data structuring**. Structured programming, p. 83–174, 1972. 1

[12] WARREN, M. S.; SALMON, J. K.. **A parallel hashed octtree N-body algorithm**. In: SUPERCOMPUTING, p. 12–21, 1993. 3.1

[13] FRIEDMAN, J. H.; BENTLEY, J. L. ; FINKEL, R. A.. **An algorithm for finding best matches in logarithmic expected time**. Transactions on Mathematical Software, 3(3):209–226, 1977. 1

[14] GARGANTINI, I.. **Detection of connectivity for regions represented by linear quadtrees**. Computers and Math with applications 8, 4(20):319–327, 1982. 1, 2.4.2

[15] GARGANTINI, I.. **Linear octrees for fast processing of three-dimensional objects**. Computers Graphics and Image Processing, 20(4):365–374, 1982. 2.4.2

[16] FINKEL, R. A.; BENTLEY, J. L.. **Quad trees: A data structure for retrieval on composite keys**. Acta Informatica, 4:1–9, 1974. 1

[17] SAMET, H.. **Applications of spatial data structures: Computer graphics, image processing, and GIS**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. 2

[18] SAMET, H.. **The design and analysis of spatial data structures**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. 2

[19] STOCCO, L.; SCHRACK, G.. **Integer dilation and contraction for quadtrees and octrees**. In: COMMUNICATIONS, COMPUTERS AND SIGNAL PROCESSING, p. 426–428, 1995. 5.1

[20] SCHRACK, G.. **Finding neighbors of equal size in linear quadtrees and octrees in constant time**. Computer Vision, Graphics and Image Processing, 55(3):221–230, 1992. 5

[21] GLASSNER, A.. **Space subdivision for fast ray tracing**. IEEE Computer Graphics and Applications, 4(10):15–22, 1984. 3.1

[22] HUNTER, G. M.. **Efficient computation and data structures for graphics.** PhD thesis, Princeton University, Princeton, NJ, USA, 1978. 1

[23] KAWAGUCHI, E.; ENDO, T.. **On a method of binary picture representation and its application to data compression**. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2(1):27–35, January 1980. 1

PUC-Rio - Certificação Digital Nº 0410421/CA

[24] KAWAGUCHI, E.; ENDO, T. ; YOKOTA, M.. **Df-expression of binary-valued picture an its relation to other pyramidal representation**. In: PROCEEDINGS OF THE FIFTH INTERNATIONAL CONFERENCE ON PATTERN RECOGNITION, p. 822–827, December 1980. 1

[25] KLINGER, A.. **Patterns and search statistics**. Optimizing Methods in Statistics. Academic Press, 1971. 1

[26] KLINGER, A.; DYER, C. R.. **Experiments in picture representation using regular decomposition**. Computer Graphics and Image Processing 5, 1:68–105, 1976. 1

[27] KNUTH, D. E.. **The art of computer programming, volume 3: Sorting and Searching**. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1973. 1, 2.5.3

[28] MORTON, G. M.. **A computer oriented geodetic data base and a new technique in file sequencing**. Technical report, IBM, Ottawa, 1966. 1, 2.4.1

[29] NILSSON, N.. **A mobile automation: An application of artificial intelligence techniques**. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, p. 509–520, 1969. 1

[30] PEANO, G.. **Sur une courbe qui remplit toute une aire plaine**. Mathematishe Annalen, 36:157–160, 1890. 2.4.1

[31] REDDY, D. R.; RUBIN, S.. **Representation of three-dimensional objects**. Computer Science Department CMU-CS-78-113, Carnegie-Mellon University, April 1978. 1

[32] WARNOCK, J. E.. **A hidden surface algorithm for computer generated halftone pictures**. PhD thesis, The University of Utah, 1969. 1

[33] YAU, M.-M.; SRIHARI, S. N.. **A hierarchical data structure for multidimensional digital images**. Commun. ACM, 26(7):504–515, 1983. 1

PUC-Rio - Certificação Digital Nº 0410421/CA