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.

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.

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 $n_l(p)$.

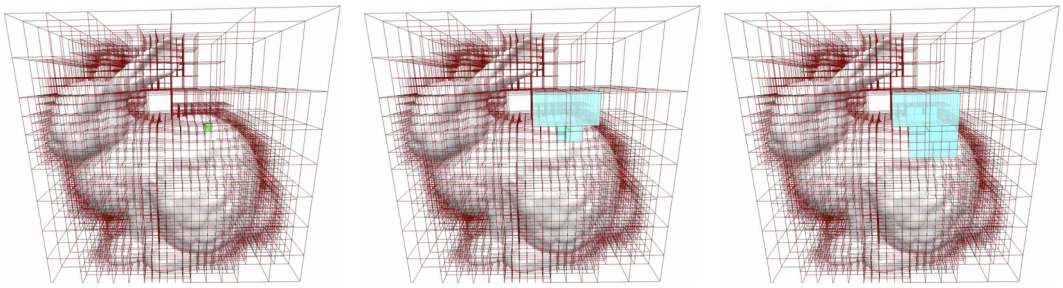
2. O nó $n_l(p)$ existe, mas não é uma folha: isso significa que o nível estimado l é pouco profundo e o nível l é incrementado até que $n_l(p)$ se torne uma folha.

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 $(l-1)$.



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 $d = 3$.

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

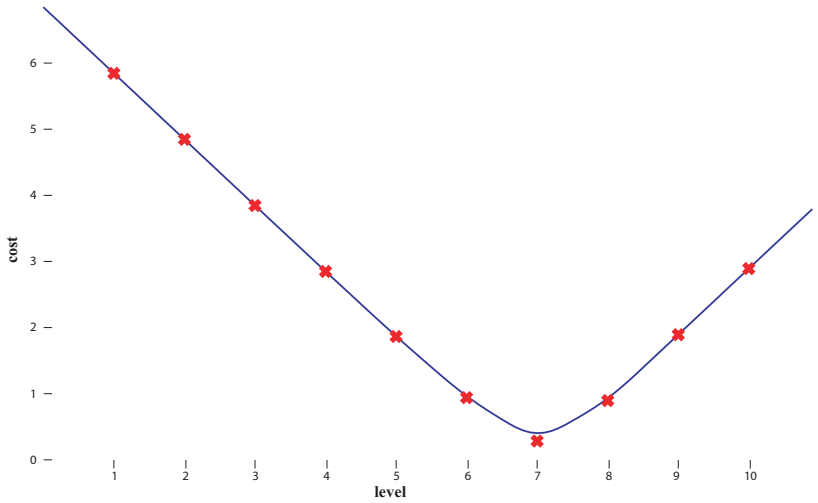


Figura 3.2: Função de custo discreta $custo(\hat{l})$ e a função de custo diferenciável custo$(\hat{l})$ para o modelo *Stanford Bunny*: o custo diferenciável tem um único mínimo, cuja parte inteira é o mínimo do custo discreto.

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:

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

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

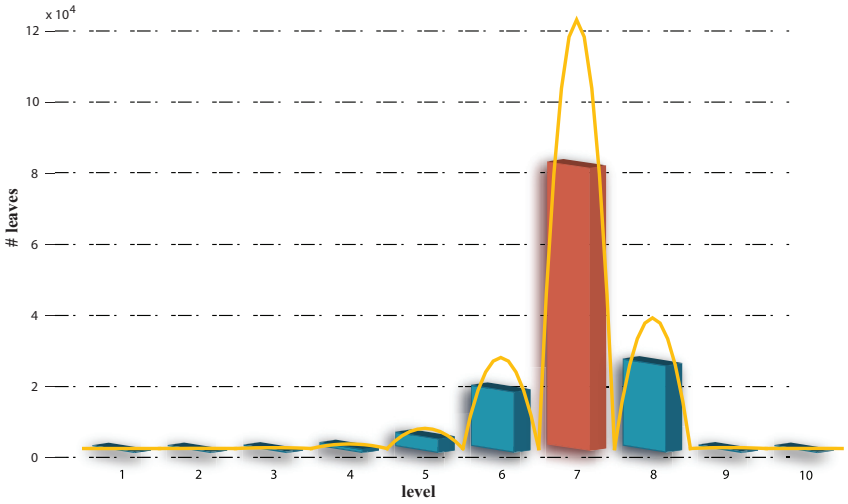


Figura 3.3: A função de probabilidade de densidade $p(l)$ do número de folhas p_l de nível l do modelo *Stanford Bunny*.

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.