BALL TREES OTIMIZADAS PARA CRIAÇÃO INTERATIVA DE LAYOUTS EM 2D COM FORMAS IRREGULARES

Luis Carlos dos Santos Coutinho Retondaro

Tese de Doutorado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Doutor em Engenharia de Sistemas e Computação.

Orientador: Claudio Esperança

Rio de Janeiro
Abril de 2022

BALL TREES OTIMIZADAS PARA CRIAÇÃO INTERATIVA DE LAYOUTS EM 2D COM FORMAS IRREGULARES

Luis Carlos dos Santos Coutinho Retondaro

TESE SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Orientador: Claudio Esperança

Aprovada por: Prof. Claudio Esperança
Prof. Ricardo Farias
Profa. Doris Clara Kosminsky
Profa. Soraia Raupp Musse
Prof. Luiz Henrique de Figueiredo

RIO DE JANEIRO, RJ – BRASIL
ABRIL DE 2022

Retondaro, Luis Carlos dos Santos Coutinho

Ball trees otimizadas para criação interativa de layouts em 2D com formas irregulares/Luis Carlos dos Santos Coutinho Retondaro. – Rio de Janeiro: UFRJ/COPPE, 2022.

XVI, 77 p.: il.; 29, 7cm.

Orientador: Claudio Esperança

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2022.

Referências Bibliográficas: p. 70 – 77.

1. Geometria computacional. 2. Layout de documento. 3. Projeto de interação. I. Esperança, Claudio. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Tendo dedicado um livro inteiro
à minha amada Raquel.
Cumpre-me deixar este presente
para minhas lindas filhas:
Paula e Isabela.

Agradecimentos

Bem-aventurado é aquele que sabe ser grato.

O registro de alguns nomes aqui são menções honrosas e muito valiosas para meu êxito neste trabalho de pesquisa. Porém, reconheço que muitos foram aqueles que dedicaram pequenos gestos, momentos e atitudes, igualmente caros e poderosos em sua eficácia, mas que não estão listados nominalmente. Importante saber que, notoriamente, sozinho eu não conseguiria.

Gratidão principalmente a Deus, na pessoa de Jesus Cristo, por toda dádiva, misericórdia e fidelidade.

Especialmente, ao meu orientador Claudio Esperança, que além de parceiro, tornou-se amigo querido.

A todos vocês eu bendigo e agradeço, pelas orações, pela paciência comigo, pelas renúncias, pelo cuidado e amor: Raquel, Paula, Isabela e toda minha família querida.

Aos amados irmãos e amigos.

Aos colegas do CEFET.

Aos colegas do LCG.

Deus os abençoe!

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

BALL TREES OTIMIZADAS PARA CRIAÇÃO INTERATIVA DE LAYOUTS EM 2D COM FORMAS IRREGULARES

Luis Carlos dos Santos Coutinho Retondaro

Abril/2022

Orientador: Claudio Esperança

Programa: Engenharia de Sistemas e Computação

A criação interativa de layouts 2D envolvendo formas irregulares, como texto ou imagens, normalmente é realizada usando um dispositivo apontador para selecionar elementos e arrastá-los para novas posições em uma página. Propomos aumentar a interação do layout com duas operações: prevenção de sobreposição e agrupamento por distância. Como estes são computacionalmente caros e podem dificultar a fluidez da interação, examinamos várias técnicas que visam implementar esses recursos de maneira eficiente. Essas técnicas envolvem a aproximação de formas 2D por coleções de bolas (círculos) que são então organizadas em *ball trees*, uma hierarquia de volumes delimitadores. Em particular, discutimos um novo algoritmo para aproximar formas com um conjunto reduzido de bolas, bem como algoritmos aprimorados de construção de *ball trees*. Eles são usados para implementar algoritmos para agrupar formas hierarquicamente por distância e permitir arrastar com prevenção de colisões, no contexto de uma aplicação web de prova de conceito para layout de formas extraídas de imagens segmentadas.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

OPTIMIZED BALL TREES FOR INTERACTIVE CREATION OF 2D LAYOUTS WITH IRREGULAR SHAPES

Luis Carlos dos Santos Coutinho Retondaro

April/2022

Advisor: Claudio Esperança

Department: Systems Engineering and Computer Science

The interactive creation of 2D layouts involving irregular shapes such as text or images is typically carried out using a pointing device to select elements and drag them to new positions on a page. We propose augmenting layout interaction with two operations: overlap avoidance and grouping by distance. Since these are computationally expensive and may hinder the fluidity of the interaction, we examine several techniques that aim at implementing these features in an efficient manner. These techniques involve the approximation of 2D shapes by collections of balls (circles) which are then organized in *ball trees*, a bounding volume hierarchy. In particular, we discuss a new algorithm for approximating shapes with a reduced set of balls, as well as enhanced ball tree construction algorithms. These are used to implement algorithms to group shapes hierarchically by distance and to allow dragging with collision avoidance, in the context of a proof-of-concept web application for laying out shapes extracted from segmented images.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

AABB	Axis-aligned Bounding Box, p. 49
BVH	Bounding Volume Hierarchies, p. 2
EDT	Euclidean Distance Transformation, p. 14
RDMA	Reduced Discrete Medial Axis, p. 13
SDT	Squared Distance Transformation, p. 14

Capítulo 1

Introdução

Documentos multimídia são compostos a partir de uma coleção de vários ativos de mídia, tais como imagens, textos e vídeos. Esses ativos precisam ser dispostos no espaço (por exemplo, numa página) e podem também ser dispostos no tempo, a fim de elaborar composições, tais como animações ou apresentações interativas.

Em uma composição, elementos de mídia são dispostos de forma semanticamente coerente, ou seja, sua disposição no espaço e no tempo tende a seguir a linha da narrativa [5].

Angelides [6] sugere que personalizar conteúdo multimídia é um processo trabalhoso, que envolve extrair e modelar informações estruturais sobre o conteúdo, exigindo que os autores especifiquem diferentes tipos de informações em diferentes níveis [7, 8].

Em geral, o paradigma de interface de um sistema de autoria de apresentação é baseado no layout estrutural. Isto significa uma ênfase no arranjo dos elementos e no modo de ativação ou exibição de cada um.

Layout é uma das muitas aplicações em que as formas 2D devem ser processadas e consultadas para extrair relações espaciais, como por exemplo, se elas contêm um ponto ou estão contidas em uma janela retangular. Em softwares genéricos de autoria, elementos como texto, imagens e diagramas são aproximados por representações (*proxies*) de caixas delimitadoras, pois elas podem ser manipuladas com mais eficiência do que formas irregulares. Por exemplo, em um cenário típico, ao mover um conjunto de formas de um lugar para outro em uma página, o usuário deve primeiro selecionar as formas desejadas, clicando nelas individualmente com um dispositivo apontador ou desenhando um retângulo ou uma curva fechada que as contém. As formas selecionadas são então arrastadas para uma nova posição, possivelmente sobrepondo outras formas localizadas ali.

Em contraste, o processo de layout poderia usar outras formas de interação que dependem da estimativa de proximidade entre formas 2D irregulares. Por exemplo, o sistema pode permitir agrupar formas implicitamente dentro de uma certa distância

umas das outras, ou evitar sobreposições quando uma forma é movida para outro ponto, deslocando as formas intermediárias para o lado. A Figura 1.1 ilustra estes conceitos.

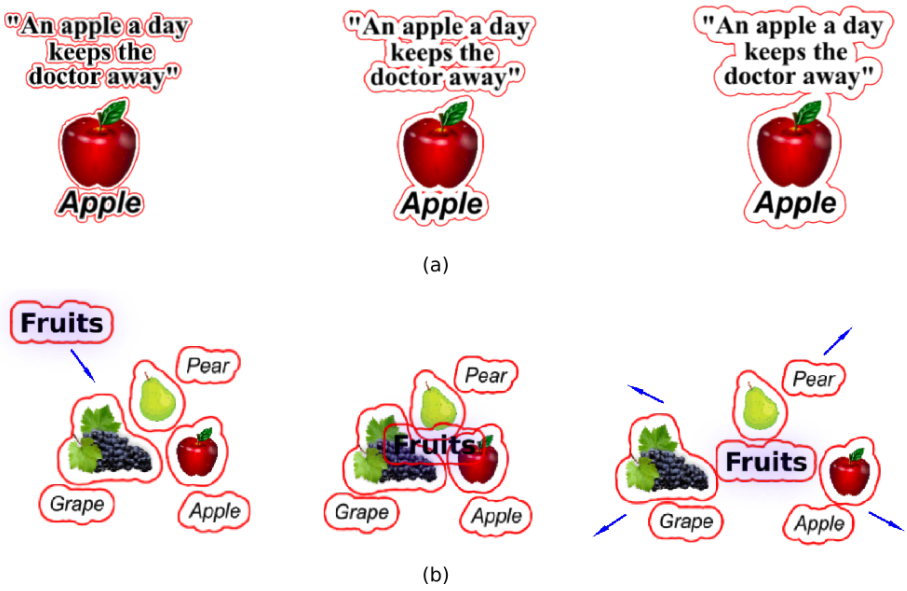


Figura 1.1: Interações de layout de distância. (a) Formas individuais podem ser agrupadas automaticamente de acordo com um determinado limite de distância. (b) Mover uma forma para uma região muito ocupada do layout abre espaço deslocando as formas intermediárias.

Para serem úteis, tais recursos de interação devem considerar a geometria exata das próprias formas, ou usar *proxies* que se aproximem dessas formas dentro de uma pequena tolerância. Além disso, a fluidez da interação exige que predicados geométricos, como distância da forma e sobreposição de forma, sejam computados de forma eficiente.

Propomos otimizar a computação de consultas de proximidade entre formas 2D irregulares usando duas ideias principais: usar coleções de bola (círculo) como *proxies* e organizar essas coleções em *ball trees* [2, 9, 10], uma variação de estruturas de dados conhecidas como hierarquia de volumes delimitadores *(BVH - Bounding Volume Hierarchies)*. A primeira ideia permite que formas sejam aproximadas com um conjunto relativamente pequeno de primitivas simples, enquanto a segunda ideia é um dispositivo frequentemente usado em aplicações de simulação física. Grande parte deste trabalho é dedicado a discutir e propor abordagens aprimoradas para esses dois problemas, que são então aplicadas em um protótipo de layout de formas como prova de conceito que implementa agrupamento hierárquico por distância, bem como interações com prevenção de colisões.

A principal motivação para explorarmos o problema de construção de *ball tree* é nosso interesse em medir distâncias relativas entre elementos gráficos bidimensionais de formato irregular em taxas interativas.

Em princípio, esses elementos podem ser qualquer informação, como letras, palavras, frases, imagens, fragmentos de imagens, etc. Em particular, pretendemos lidar com documentos digitalizados que são segmentados em partes que podem ser agrupadas independentemente de sua posição original na página.

Empregamos o termo *patch* para nos referirmos a cada elemento gráfico, ou seja, cada componente conexa advinda da segmentação da imagem original de entrada. Na Figura 1.2, por exemplo, há uma série de *patches* correspondentes a marcas gráficas de texto e um outro *patch* correspondente a uma figura de coelho. Claramente, os *patches* referentes ao texto têm uma hierarquia de agrupamento natural, que é definido pela distância entre as marcas gráficas.

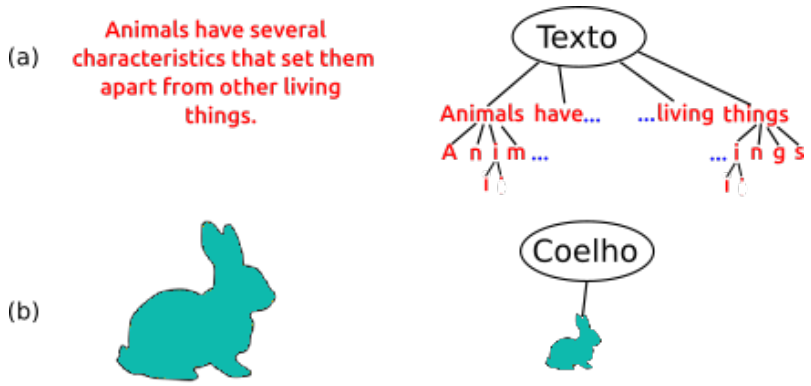


Figura 1.2: Exemplo de *patches*. Esquerda: *patches* formados por cada marca gráfica, a partir da segmentação de imagem - (a) texto e (b) coelho. — Direita: Hierarquia de agrupamento natural correspondente a (a) e (b).

A ideia básica é organizar os *patches* de um documento escaneado considerando que os elementos gráficos próximos estão semanticamente relacionados (veja a Figura 1.3). Além disso, essa associação funciona hierarquicamente, como por exemplo, em documentos impressos onde letras próximas formam palavras e grupos de palavras formam linhas de texto.

Neste trabalho propomos algumas contribuições que se baseiam nos algoritmos e experimentos em *ball trees* como descrito no artigo seminal de Omohundro [9], entre outras adições que estão abaixo relacionadas de forma resumida:

- Propomos adicionar agrupamento hierárquico e prevenção de colisões como dispositivos de interação em aplicações de layout usando conjuntos de círculos organizados em *ball trees* como *proxies* geométricos.

- Os três algoritmos de construção de *ball trees* de melhor desempenho descritos por Omohundro [9] foram estendidos com o que chamamos de otimização *enclosing leaves - (EL)*, ou seja, nós internos construídos como os menores círculos delimitadores de todos os nós folha descendentes. Isso contrasta com a prática usual de limitar apenas os dois descendentes imediatos (Seção 2.1.1).

(a) (b)

Figura 1.3: Parte de um documento digitalizado - A imagem da estátua e o primeiro parágrafo do texto têm suas *ball trees* desenhadas em azul. (a) A forma 2D da estátua é selecionada, arrastada e posicionada para a direita. (b) A forma 2D do texto é empurrada para se ajustar ao novo layout.

- Três novos algoritmos de construção de *ball trees* são propostos, também estendidos com a otimização *EL* (Seção 3.1).
- A estratégia *branch and bound* usada em consultas de distância entre árvores e interseção de árvores emprega uma nova métrica de limite de distância que é mais justa do que a métrica de distância máxima usual (Seção 3.2).
- Além de experimentos com diferentes distribuições aleatórias de pontos, também usamos conjuntos de dados contendo coleções de círculos aproximando formas de teste, uma vez que surgem em muitas aplicações importantes (Seção 3.3).
- Experimentos foram realizados para avaliar a qualidade das *ball trees*, pois são usadas em seis tipos diferentes de consultas. Esta experimentação complementa a prática usual de igualar a qualidade à área total da *ball tree* (Seção 3.3).
- Um novo algoritmo para aproximar uma imagem binária com conjuntos de bolas de cardinalidade ajustável (Capítulo 4).
- Duas operações de layout de forma e algoritmos para sua implementação usando *ball trees* são propostos: agrupamento baseado em distância e arraste de forma usando prevenção de colisões (Capítulo 5).
- Implementamos uma aplicação web de layout de forma como prova de conceito, onde muitos algoritmos de *ball tree* podem ser testados em um ambiente interativo (Capítulo 5).

Capítulo 2

Fundamentos

Este capítulo apresenta uma revisão sobre os principais trabalhos relacionados ao problema de organização de layout e agrupamento, considerando ainda os arranjos espaciais e os modelos de interação que podem permitir um processo de organização de elementos gráficos de maneira mais intuitiva. São destacados os desafios técnicos para implementar um sistema relevante e eficiente no apoio à solução proposta.

2.1 Consulta de relações espaciais 2D

A consulta de relações espaciais em modelos geométricos bidimensionais é uma atividade comum em diversas aplicações. Para resolver o problema, muitas abordagens têm usado decomposição espacial ou *Bounding Volume Hierarchy (BVH)*, que é uma estrutura de dados hierárquica sobre dado conjunto de objetos geométricos.

A ideia por trás dessas abordagens é, respectivamente, decompor o espaço que os objetos ocupam (usando decomposições) e aproximar os objetos (com volumes delimitadores) para reduzir o número de pares de objetos ou primitivas que precisam ser verificados para responder à consulta requerida.

Octrees [11, 12](Fig. 2.1), *k-d-trees* [13, 14] e *BSP-trees* [15, 16](Fig. 2.2) são exemplos de técnicas de decomposição espacial.

Tipicamente, BVHs são estruturas de dados em árvore, que cobre um número crescente de volumes simples, como bolas (círculos) [2, 10], caixas delimitadoras alinhadas ao eixo [17, 18] ou caixas delimitadoras orientadas [19, 20].

Muitas variedades de BVHs foram propostas para o processamento eficiente de consultas espaciais, como M-trees [21], VP-trees [22], R-trees [23, 24] e Ball trees [9, 25].

As BVHs têm muitas aplicações, como correspondência de imagens [26], anonimização de dados [27], métodos não paramétricos [28]. Em particular, eles são úteis no contexto das extensões de interações de layout propostas neste trabalho, uma vez

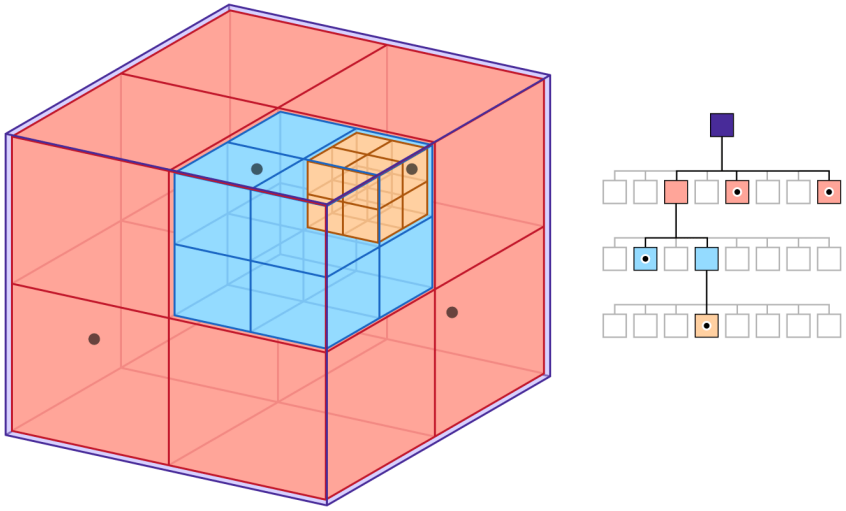


Figura 2.1: Exemplo de um arranjo espacial e lógico de uma *octree*

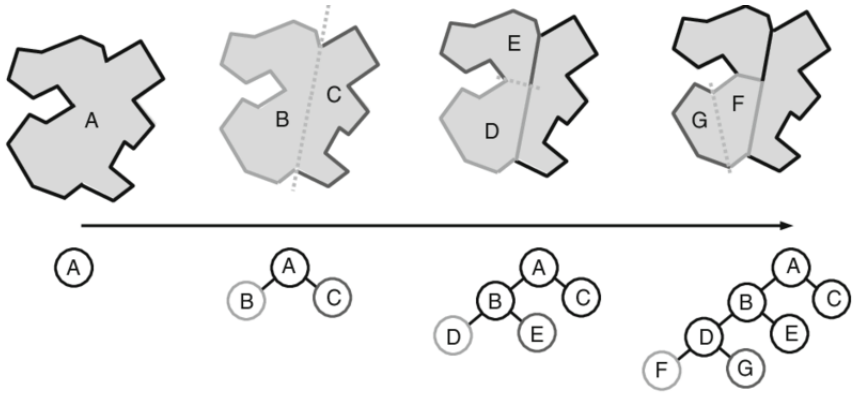


Figura 2.2: Exemplo das etapas de construção de uma *BSP tree*

que foram usados para detecção de colisões [29, 30] e indexação de dados espaciais [24, 25].

Para superar o desafio de encontrar o ponto mais próximo entre dois *patches*, bem como outras consultas espaciais recorremos às *ball trees* [9].

2.1.1 Ball trees

Ball trees permitem representar uma forma complexa em vários níveis de detalhe usando círculos delimitadores. Usando níveis rasos da árvore, é possível ter uma aproximação da forma do objeto. Além disso, encontrar o par mais próximo de pontos entre dois objetos representados por ball trees leva tempo $O(\log^2 n)$ em vez de $O(n^2)$. Tipicamente, Ball trees otimizadas e com performance eficiente para as consultas propostas contêm um número reduzido de bolas de entrada, bem como uma área total minimizada.

Claramente, alguns tipos de BVH podem ser usados para os propósitos deste

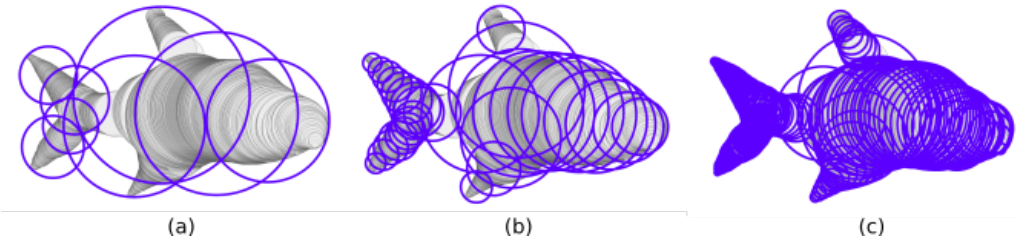


Figura 2.3: Exemplo de uma forma representada por uma *ball tree* nos níveis 2, 3 e 4 (a-c)

trabalho, mas optamos por *ball trees* porque:

1. Elas combinam com nossa escolha de coleções de bolas como *proxies* geométricos;

2. elas fornecem uma maneira conveniente de representar clusters de formas (consulte a Seção 5.2) e;

3. elas simplificam a computação da profundidade de penetração e direção (consulte a Seção 5.3).

Além disso, testes comparando *ball trees* com um BVH [31] baseado em retângulo indicam que o primeiro tende a ter um desempenho melhor do que o segundo em consultas de distância a um ponto em formas tipográficas.

Uma *ball tree* é uma árvore binária onde cada nó está associado a uma porção do espaço delimitada por uma bola, ou seja, um círculo em 2D, uma esfera em 3D e assim por diante. Os nós folha apontam para os objetos que estão sendo indexados, normalmente pontos, bolas ou outros objetos para os quais uma bola delimitadora mínima pode ser produzida, ou seja, a menor bola que envolve o objeto.

As bolas associadas aos nós internos são geralmente as menores bolas contendo os dois nós filhos. A Figura 2.4 mostra um exemplo.

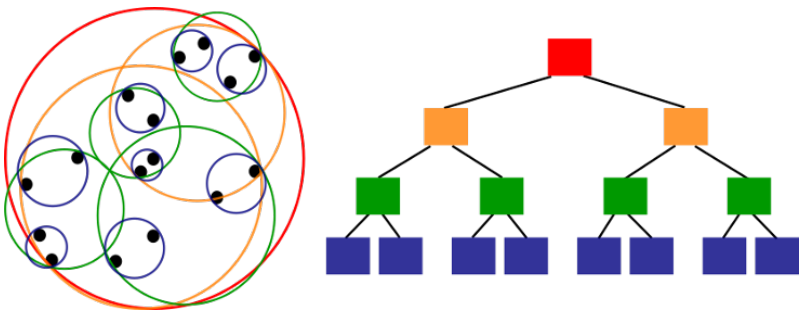


Figura 2.4: Representação típica de *ball tree* binária. (Esquerda) mostra uma *ball tree* onde os nós internos envolvem os dois filhos imediatos. (Direita) topologia da árvore.

Algoritmos de construção para BVHs, como *ball trees*, podem ser divididos em três classes: construção de cima para baixo (*top-down*), de baixo para cima (*bottom-up*) e baseada em inserção no ponto ideal (*insertion based*) [30]. A abordagem *top-down* é a mais popular: ela subdivide recursivamente a coleção em coleções parciais cada vez menores até que apenas um elemento seja deixado e colocado em um nó folha. Algoritmos *bottom-up* localizam pares de nós que estão próximos, criando nós pais apontando para eles; isso é feito recursivamente a partir de nós folha individuais até atingir a raiz da árvore. A abordagem de inserção (também chamada de incremental) começa com uma árvore vazia, insere uma bola de cada vez no melhor local da árvore.

Omohundro [9] afirma que as estratégias *bottom-up* geralmente são melhores, apesar de terem um alto custo de construção, pois tendem a ser mais eficientes em encontrar agrupamentos ótimos. Omohundro usa o volume total de todas as bolas em uma *ball tree* como medida de sua qualidade e conclui que minimizar a área total pode ser o critério mais eficiente para a maioria das aplicações. Por esta razão, tal critério foi empregado como métrica fundamental em nossos experimentos.

Consideremos o problema de construir uma *ball tree* para uma coleção de n bolas. Para simplificar a análise, assumimos uma abordagem *bottom-up*, onde todos os nós folha são emparelhados para construir o próximo nível superior. Assumindo que n é par, o nível mais baixo da árvore consistirá de $m = \frac{n}{2}$ pares de bolas.

Vamos chamar P o número total de pares possíveis. Então,

$$P(n) > \binom{2m}{m} \sim 4^m/\sqrt{\pi m}.$$

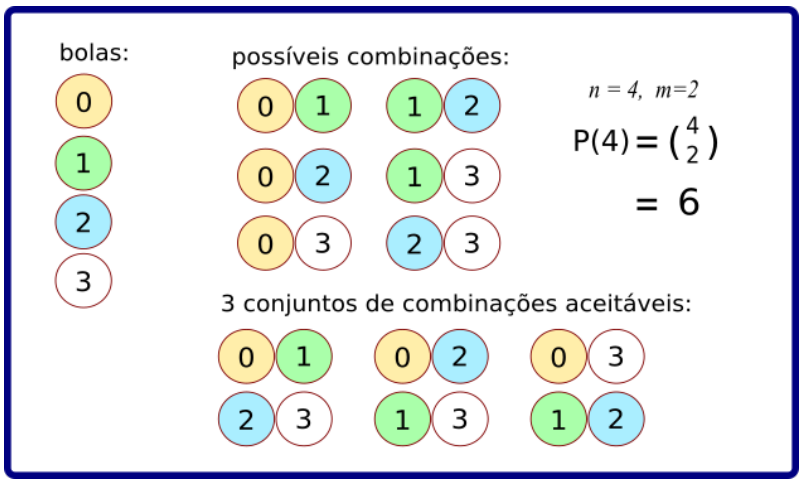


Figura 2.5: Exemplo de combinações de pares para um conjunto pequeno de bolas ($n = 4$).

A Figura ilustra as possíveis combinações de pares para $n = 4$, sem repetições de bolas, ou seja, com 3 possíveis combinações aceitáveis.

Pode-se mostrar que $P(n)$ cresce exponencialmente rápido; de fato, por exemplo, $P(50) \sim 10^{57}$ (tab. 2.1).

Tabela 2.1: Número de pares de n elementos para alguns valores de n.

n	4	6	8	10	30	50
$P(n)$	6	90	2,520	113,400	8×10^{27}	9×10^{56}

Como o espaço de busca para este problema é enorme, uma abordagem determinística para encontrar uma árvore que possa ser considerada "ótima" em algum sentido está fora de questão e devemos confiar em heurísticas.

2.2 Aproximação de formas

Representações simplificadas para objetos complexos são úteis em cenários em que sacrificar a fidelidade da forma é aceitável para aumentar a eficiência da computação.

Em muitas aplicações, como layout de forma, as aproximações não precisam ser conservadoras, ou seja, incluir a forma exata, mas devem produzir resultados bons o suficiente para consultas de proximidade.

Por exemplo, uma forma bidimensional S pode ser representada por um polígono com um número arbitrário de lados que, por sua vez, pode ser simplificado com ferramentas como o algoritmo Douglas–Peucker–Ramer [32].

Subdivisão também pode ser empregada para dividir uma forma recursivamente em pedaços menores de tamanho decrescente, geralmente quadrados ou retângulos, seguidos de descarte de pequenos pedaços [33].

Outra alternativa é computar um conjunto mínimo de bolas B tal que a distância de um ponto a S é limitada pela distância desse ponto a B, dentro de uma determinada margem de erro ϵ.

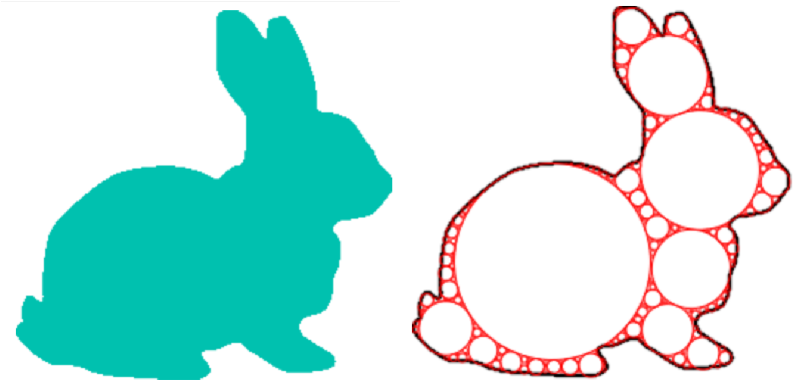


Figura 2.6: Aproximação de uma forma com um conjunto de bolas baseada em *circle packing* [1]. (a) Forma original, (b) Conjunto de bolas que aproximam a forma do objeto.

O problema de aproximar uma forma com um conjunto de bolas tem sido estudado por muitos autores. Por exemplo, Weller, et al. [1] propôs a implementação de uma abordagem baseada em *circle packing* para minimizar eficientemente a quantidade de bolas de entrada na construção de uma estrutura chamada *Inner Sphere Trees*. A Figura 2.6 mostra o resultado de uma implementação que adaptamos a partir desta abordagem. Uma espécie de "pixelização" é suportada por uma grade retangular com a mesma proporção da imagem, onde cada unidade é usada para construir os círculos internos.

Outras abordagens baseiam-se no conceito de *eixo medial*, ou seja, o conjunto de pontos que possuem mais de um ponto mais próximo na fronteira do objeto, proposto inicialmente por Blum [34].

A abordagem medial para representar um objeto descreve um locus a meio caminho (no centro de um círculo bitangente) entre duas seções da borda do objeto e fornece a distância até o limite (chamado raio medial), produzindo o objeto como uma união de círculos bitangentes sobrepostos. Uma maneira de pensar sobre essa representação é considerar um locus dado por (p, r), onde p é o centro do círculo e r o raio. Em algumas representações, os vetores do ponto medial aos dois ou mais pontos de fronteira correspondentes são incluídos; em outras, esses vetores são obtidos através da transformada de distância na vizinhança de p, já que em p há uma descontinuidade do gradiente dessa função. Quando esses vetores são incluídos, a primitiva chamada de átomo medial, é um ponto central p com dois vetores de comprimento igual a r (ver Fig. 2.7). Blum descreveu o locus medial interior restringindo os círculos bitangentes aos contidos inteiramente no interior do objeto. Os raios desses átomos mediais não se sobrepõem e varrem o interior do objeto.

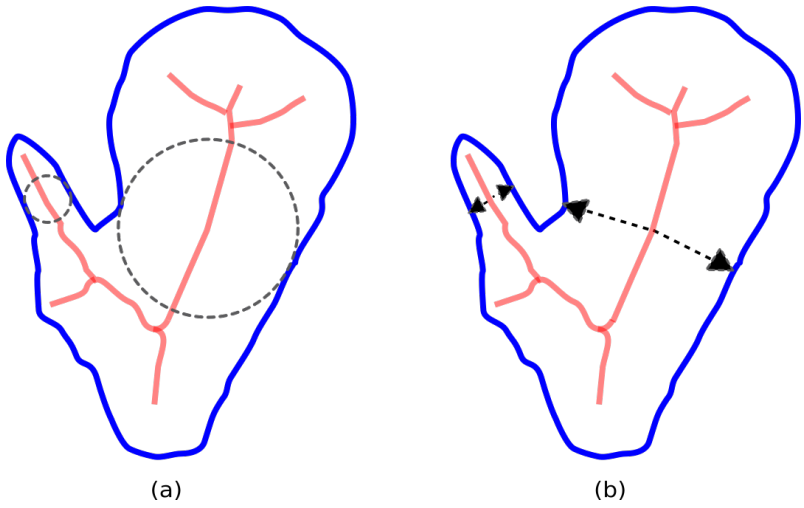


Figura 2.7: (a) Uma representação de círculos mediais: locus contínuo ou coleção de loci contínuos de círculos mediais. (b) Uma representação de átomos mediais: locus contínuo ou coleção de loci contínuos de átomos mediais.

Frequentemente, o eixo medial é computado em imagens digitais e é então chamado de *Discrete Medial Axis* (DMA), que tem a boa propriedade de ser reversível, ou seja, a partir de seus pontos, podemos reconstruir a forma original [2].

Na verdade, o DMA é definido como o conjunto de pixels centrais dos círculos máximos contidos na forma do objeto [2, 35–37], onde um círculo máximo é definido como um disco contido em uma determinada forma que não é totalmente coberto por nenhum outro disco contido nessa forma.

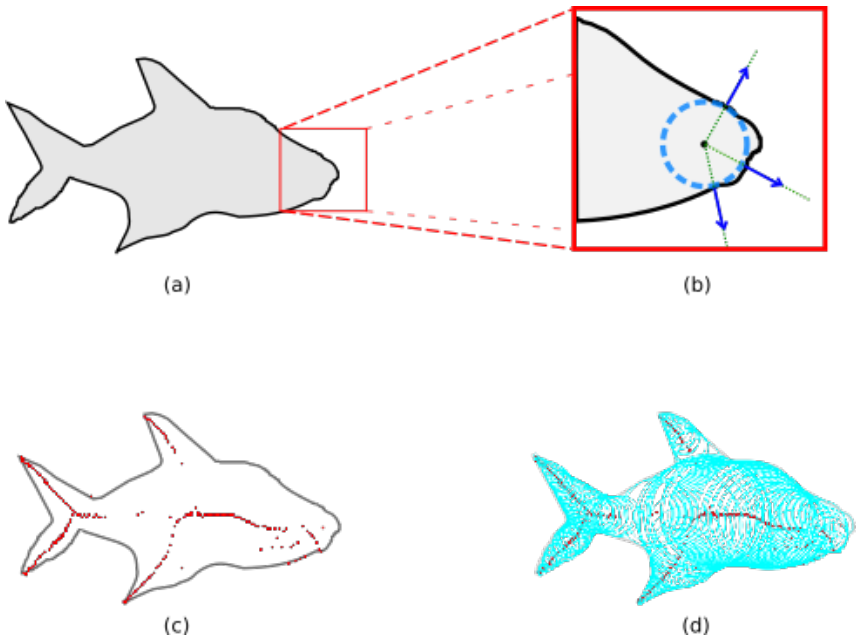


Figura 2.8: Aproximação da forma baseada no eixo medial. (a) Forma original, (b) Detalhe de um círculo máximo contido na forma com centro no DMA, (c) Pontos do DMA. (d) Aproximação da forma do objeto pela união dos círculos.

Outra inspiração para nossa abordagem foi o trabalho de Broutta, et al. [2] que implementou um método para construir BVHs de esferas - também conhecido como *sphere-trees* - em tempo linear onde os conjuntos de esferas em cada nível são obtidos por uma extração discreta do eixo medial em diferentes níveis de resolução de objeto, usando uma abordagem piramidal regular e controlando de forma eficiente a propagação do erro e sua distribuição em cada nível. A Figura 2.9 ilustra os principais estágios desse processo.

Um método clássico para computar distâncias a um objeto em uma imagem digital foi proposto por Borgefors [38] e é conhecido como a transformada de distância (DT). Em imagens binárias, DT e DMA são ferramentas clássicas para análise de formas [39–41]. As transformadas de distância usam uma grade ou discretização de objetos para resolver o problema de encontrar a distância de um determinado ponto a uma forma e podem ser computadas eficientemente com shaders de GPU usando um algoritmo chamado *jump flooding* (JFA) [42].

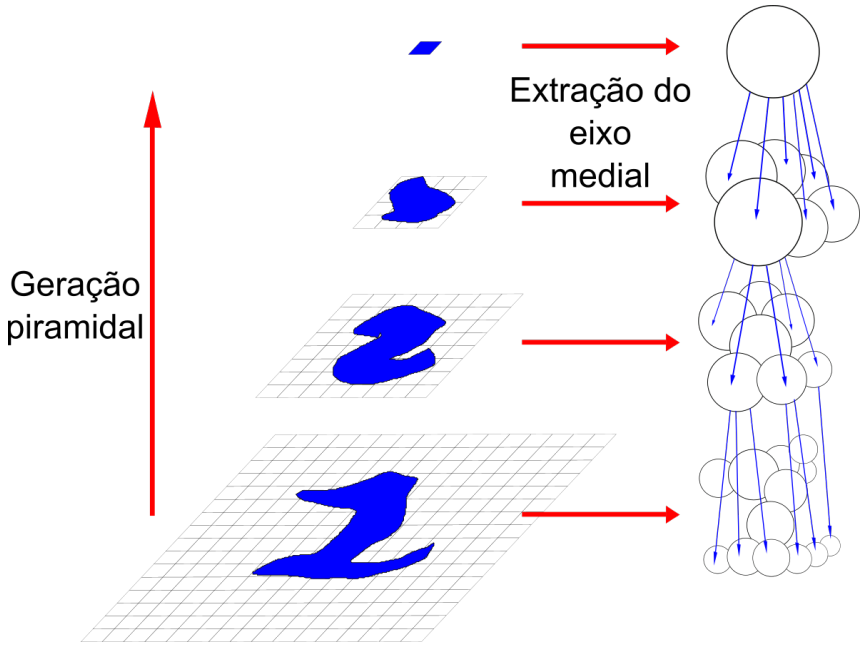


Figura 2.9: Principais estágios da construção da *sphere tree* [2], onde os conjuntos de esferas em cada nível são obtidos por uma extração de DMA em diferentes níveis de resolução do objeto, em uma abordagem piramidal regular.

O algoritmo (JFA) calcula uma transformada de distância em um espaço discreto 2D representado por uma textura. Quando mais de um objeto é considerado, pode-se adaptar o JFA para restringir a propagação ao objeto mais próximo de cada pixel, resultando assim no diagrama de Voronoi, onde as sementes são os objetos [43].

Em resumo, o JFA propaga as informações de todos os sítios para todos os pixels da textura. No primeiro estágio, chamado estágio de mapeamento, cada semente (ou sítio) é mapeada para seu pixel correspondente na textura. Isso pode ser feito em paralelo pelo rasterizador de GPU. O segundo estágio, o estágio de propagação, consiste em $\log_2(n)$ passos de propagação, onde n é a dimensão da textura. Todos os pixels são processados em paralelo pela GPU. Em cada passo, a informação da semente mais próxima armazenada em cada pixel $p(x, y)$ é propagada para no máximo oito outros pixels em $(x+i, y+j)$ onde $i, j \in \{-k, 0, k\}$, e k é o comprimento do passo atual. No primeiro passo, usa-se $k = n/2$ como comprimento inicial da etapa para garantir que cada pixel seja atingido por pelo menos um sítio. Aqui, assumimos que n é uma potência de 2. Caso contrário, o comprimento inicial da etapa é definido como $2^{\lceil log n-1 \rceil}$. O comprimento do passo k é dividido na metade em cada um dos passos seguintes. Quando mais de um sítio é propagado para o mesmo pixel, o sítio mais próximo do pixel será escolhido para atualizá-lo. Após realizar o passo com comprimento 1, o resultado é o diagrama de Voronoi computado. Um exemplo de diagrama de Voronoi computado com esta técnica pode ser visualizado na Fig. 2.10.

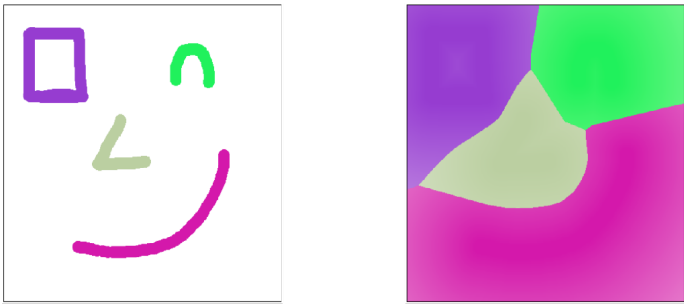


Figura 2.10: Ilustração do algoritmo *JFA*. (a) Formas como sementes de Voronoi, (b) Diagrama de Voronoi computado.

Além disso, um diagrama de Voronoi pode também ser usado para construir um histograma de distâncias entre os vários segmentos de uma imagem. Dessa forma, os intervalos mais frequentes desse histograma podem ser usados para estimar limites de distância que darão origem à hierarquia de agrupamentos.

Outras abordagens mais rápidas que JFA foram implementadas [44], mas com menor precisão. Neste trabalho, descartamos JFA na GPU, pois os experimentos que realizamos resultaram em um baixo desempenho para reconstrução e manipulação da estrutura a taxas interativas.

Outra técnica comum para computar o eixo medial é o afinamento morfológico (*thinning*) que visa iterar sucessivamente camadas da transformada da distância [45]. Morfologia matemática binária é baseada nas operações de erosão e dilatação de um objeto através de um elemento estruturante. Quando o elemento estruturante é um círculo (2D), temos a relação entre o raio e a curva de nível correspondente do campo da transformada de distância. De forma semelhante, pode-se obter uma aproximação da representação medial através de um processo de erosão repetida usando uma bola de raio unitário, afinando o objeto até que ele tenha um pixel de espessura.

Em verdade, entretanto, os métodos de *thinning* costumam implementar erosão considerando elementos estruturantes equivalentes a um pixel, isto é, um quadrado ao invés de um círculo. Portanto, o esqueleto obtido pelo método não coincide exatamente com o eixo medial.

Os conjuntos de bolas de aproximação da forma mostrados neste trabalho foram produzidos a partir de componentes conectados extraídos pela segmentação de imagens de documentos. Então, cada elemento é processado com uma adaptação dos algoritmos eficientes propostos por Coeurjolly et al. [46].

Em particular, tais algoritmos descrevem uma abordagem para computar o eixo medial discreto reduzido (*Reduced discrete medial axis - RDMA*) que é ilustrado pela Figura 2.11 e descrito a seguir.

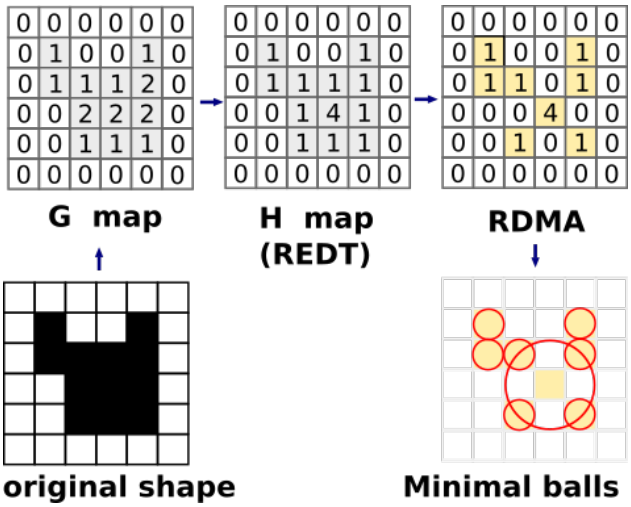


Figura 2.11: Computação do RDMA para obtenção do conjunto mínimo de bolas que aproximam a forma.

Considere a transformada de distância ao quadrado (*Squared distance transformation - SDT*) em uma imagem binária. Essa transformada mapeia cada pixel de fundo para a distância ao quadrado do pixel de primeiro plano mais próximo.

Seja a imagem P de tamanho $n \times n$; $\overline{P}$ denota o complementar de P, isto é, o conjunto de pixels do plano de fundo. Na etapa inicial, a saída do algoritmo é uma imagem $H = (i, j)$ que armazena a SDT. Para cada ponto i, j da imagem, a transformada de distância ao quadrado é dada por:

$$h(i,j) = min\{(i-x)^2 + (j-y)^2;\ \ 0 \leq x,\ y < and\ \ (x,y) \in \overline{P})\}$$

A computação de SDT é realizada em duas etapas. A primeira é computar a transformada de distância euclidiana unidimensional (*Euclidean Distance Transformation - EDT*), a partir da imagem fonte P no eixo x (veja a Figura 2.12). A EDT é denotada por $G = \{g(i,j)\}$, onde para uma dada linha j:

$$g(i,j) = \min_{x}\{|i-x|\,;\ \ 0 \leq x < n\ \ and\ \ (x,j) \in \overline{P}\}$$

Em seguida, construímos a imagem H, considerando um processo equivalente para o mapeamento no eixo y. Este é um processo de mistura que mescla os resultados das duas etapas e está associado ao cálculo da transformada da distância euclidiana reversa (REDT), onde para uma dada coluna i, têm-se:

$$h(i,j) = \min_{y}\{g(i,y)^2 + (j-y)^2;\ \ 0 \leq y < n\}$$

Ao concluir a computação do SDT nas duas etapas, observamos que a operação de minimização *min* associada à REDT corresponde à computação do envelope inferior

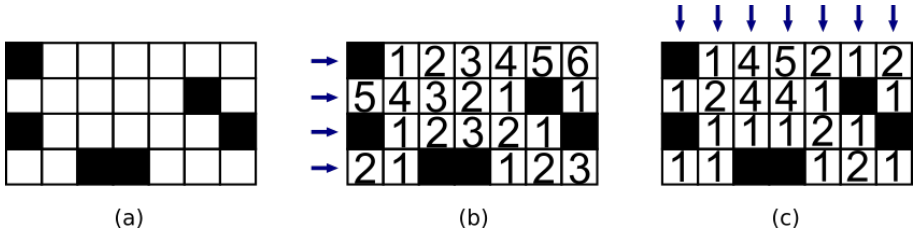


Figura 2.12: Ilustração do algoritmo SDT. (a) Imagem original P, (b) o mapa G resultante da etapa 1 (EDT), (c) o mapa H resultante da etapa 2 (REDT) equivalente a SDT final.

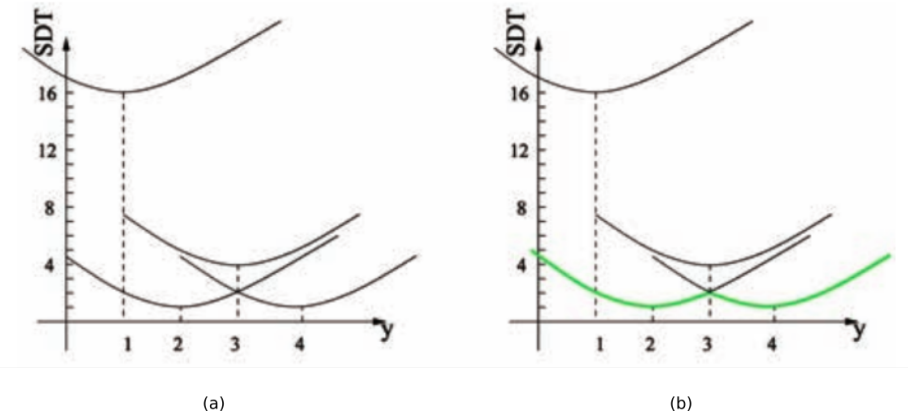


Figura 2.13: Ilustração da computação do SDT como uma extração do envelope inferior.

de um conjunto de parábolas. Mais precisamente, vamos supor que computamos o primeiro passo do algoritmo SDT (EDT, eixo x) e seja $\{g(i,y)\}$, $(0 \leq y < n)$ uma coluna de G. Se considerarmos o conjunto de parábolas $F_y^i(j) = g(i,y)^2 + (j-y)^2$, a coluna $h(i,j)$ obtida depois do segundo passo, é exatamente o envelope inferior de $\{F_y^i(j)\}$ com $0 \leq y < n$. Como exemplo, destacamos a quinta coluna de G na fig. 2.12.b, $[4,1,2,1]$. O gráfico da fig. 2.13.a representa o conjunto de parábolas $F_y^i(j)$ e no gráfico da fig. 2.13.b, a curva destacada é o envelope inferior. Assim, o resultado do processo de minimização é $[2,1,2,1]$.

Assim, assumindo que Q é o SDT de P, o RDMA é o conjunto de pontos (i,j); tal que existe ao menos uma linha em que a parábola associada a $(i,j,q(i,j))$ é preservada e não duplicada durante o processo de redução unidimensional.

Tanto a computação do REDT quanto a extração do RDMA tornam a complexidade linear no tamanho da imagem.

Embora a abordagem RDMA seja muito eficiente e produza boas aproximações, ela pode perder algumas características finas na imagem. Outra deficiência é que ele não pode ser ajustado para produzir aproximações mais finas ou grosseiras. Na Seção 4.2 descrevemos outro algoritmo baseado no SDT que trata desses problemas.

2.3 Layout de formas

A maioria das aplicações genéricas de criação 2D oferece alguma funcionalidade para colocar e organizar objetos em uma página. Isso geralmente é feito com a ajuda de interfaces WIMP (Window, Icon, Menu, Pointer), onde objetos individuais podem ser colocados arrastando com um dispositivo apontador e grupos de objetos são criados clicando individualmente em seus elementos ou selecionando todos os elementos contido ou sobreposto a uma determinada forma desenhada pelo usuário – tipicamente uma janela retangular. Entretanto, as interfaces tradicionais podem ter um desempenho insatisfatório, pois exigem vários comandos individuais, e as tarefas se tornam tediosas porque são baseadas em ações repetidas para realizar uma única operação.

Outro problema é que, após a identificação dos grupos, o usuário deve explicitamente relacioná-los para destacar a semântica de toda estrutura do agrupamento. Em geral, usa-se alguma estrutura visual, como listas ou árvores recolhíveis, onde as folhas representam os itens e os nós internos representam os grupos. Uma vez que o grupo é formado, sua caixa delimitadora é usada como um *proxy* visual para interações adicionais. Por exemplo, outra seleção usando uma janela retangular pode selecionar o grupo mesmo que nenhum elemento dentro dela se sobreponha à janela.

A tarefa de reorganizar as formas em uma página torna-se mais desafiadora à medida que seu número aumenta, devido ao esforço computacional adicional necessário e à tensão de interação induzida pela desordem visual. Nesse caso, os conteúdos são gerenciados de forma mais eficiente se organizados hierarquicamente. Em software de autoria comercial, os grupos são comumente criados pelo autor usando comandos de interface explícitos.

Além desses problemas, mídias brutas, como desenhos a mão livre ou notas manuscritas, são tratadas como imagens digitalizadas individuais. Assim, para utilizar partes deste material, primeiro é necessário empregar algum software de segmentação de imagens, seguido da coleta manual das partes úteis, o que diminui sensivelmente a eficiência de todo o processo de autoria.

Propomos vários remédios para esses problemas. Acima de tudo, sugerimos que a proximidade espacial é uma metáfora mais eficaz para transmitir relações semânticas entre elementos de conteúdo do que agregações simbólicas na forma de árvores ou listas recolhíveis. Apresentamos ainda um conjunto de técnicas de interação para construir tais hierarquias, organizá-las espacialmente em grupos semânticos.

2.3.1 Métodos de controle de layout espacial

Notadamente, os agrupamentos espaciais podem ser computados automaticamente com base em alguns critérios. Vários métodos de controle de layout espacial têm

sido propostos na literatura.

Uma técnica de interação chamada *the vacuum* é descrita em [47], onde um *widget* circular especial é proposto para criar *proxies* para representar objetos distantes na tela - a ideia é criar e usar esses manipuladores ao invés de deslocar o mouse para a região da tela onde esses objetos se encontram. Outra ideia relacionada, os *ninja cursors* [48] permitem a interação com objetos distantes através de operações de *zoom in* e *zoom out* controladas implicitamente através da velocidade de deslocamento do mouse.

Podemos ainda citar outras abordagens para melhorar a interação com objetos distantes [49, 50], subgrupos de janela [51], marcos artificiais [52] e interfaces de zoom [53, 54].

Interfaces multiescala baseadas em *zoom*

Com aglomerações de informação mais densas, uma interface intuitiva de navegação é um desafio, pois geralmente o espaço de consulta é limitado. Interfaces multiescala fornecem mecanismos alternativos eficazes para lidar com problemas associados à navegação de coleções de informações muito grandes.

Essa é a motivação primária para o trabalho de Bederson *et. al.* [55], chamado Pad++, para representar a abstração de objetos usando o que é denominado de *zoom semântico*.

Esta representação tem uma motivação natural, segundo o autor, pois é comum ver os detalhes de um objeto bem de perto, quando está ampliado (*zoom in*). No entanto, quando em *zoom out*, em vez de simplesmente ver uma versão reduzida do objeto, é potencialmente mais eficaz ver uma representação diferente dele.

Com o Pad++, além da simples escolha binária de apresentar ou omitir informações particulares, pode-se determinar a escala da informação e, talvez mais importante, os detalhes de como ela é renderizada, baseado em várias semânticas ou tarefas, a fim de obter uma filtragem das informações [54]. Pad++ produz animações suaves ao buscar os objetos desejados. As animações interpolam o nível de aproximação (*zoom*) entre uma visão panorâmica e uma visão mais estreita para trazer a exibição para o local especificado. Se o ponto final, no entanto, estiver a mais de uma largura de tela do ponto inicial, a animação será ampliada para um ponto intermediário entre os pontos inicial e final, o suficiente para que ambos os pontos fiquem visíveis. A animação então suavemente aproxima o destino. Isso dá uma sensação de contexto ao visualizador e ajuda a manter a constância do objeto. Além disso, acelera a animação, já que a maior parte do pan é executada quando o zoom é ampliado e, portanto, cobre mais distância do que a visão panorâmica enquanto o zoom é diminuído. O algoritmo faz uso de diagramas espaço-escala [3] para ajudar a analisar e construir essas trajetórias, bem como cumprir os requisitos

de renderização de uma superfície de informações mais ampla [54].

A ideia básica de um diagrama espaço-escala é simples. Considere, por exemplo, uma imagem quadrada em 2D (Figura 2.14a). O diagrama espaço-escala para esta imagem seria obtido pela criação de muitas cópias da imagem 2-D original, uma em cada ampliação possível, e empilhando-as para formar uma pirâmide invertida (Figura 2.14b). Enquanto os eixos horizontais refletem as dimensões espaciais originais, o eixo vertical representa a escala, isto é, a ampliação da imagem nesse nível.

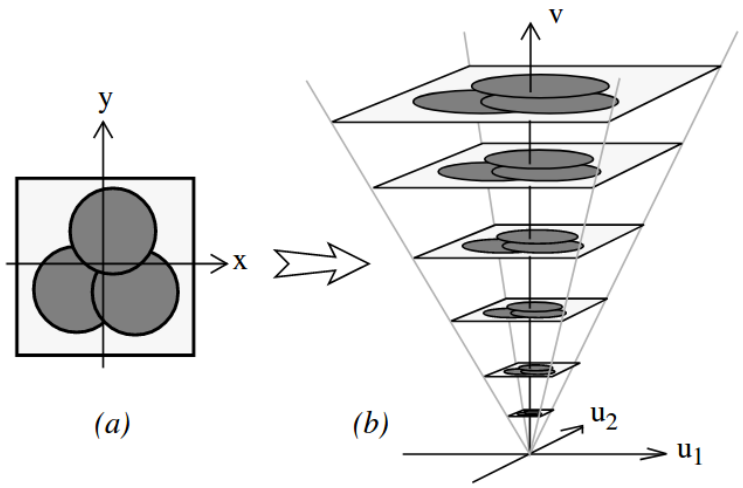


Figura 2.14: Construção de um diagrama espaço-escala de uma imagem 2D [3]

Este é um trabalho seminal interessante relacionado à organização de conteúdos em níveis de detalhe. Uma ideia importante nesse caso é que os elementos semanticamente relacionados tendem a ser posicionados próximos uns dos outros, o que é particularmente interessante como sugestão de organização hierárquica.

Interfaces Baseadas em Traços

As interfaces baseadas em traços surgiram como contraposição às interfaces gráficas tradicionais WIMP. A ideia principal é permitir ao usuário concentrar mais esforços na tarefa e menos em comandos específicos para realizá-las.

O desenvolvimento de interfaces baseadas em traços inclui o estudo de diversos problemas relacionados [56]:

1. a aquisição do traço através de algum dispositivo de entrada apropriado,

2. como processar o traço de entrada para suavizá-lo, simplificá-lo ou filtrá-lo de alguma forma,

3. como interpretar o traço no contexto de uma dada aplicação ou mesmo usá-lo como método de especificação de alguma forma pré-conhecida,

4. mecanismos para modificação de traços entrados anteriormente,

5. estruturas de dados e algoritmos para reconstrução de superfícies a partir de traços,

6. reconhecimento de gestos (a partir de traços).

Um trabalho pioneiro nesta área é o sistema de modelagem 3D Teddy [57] onde a maior parte das ações são baseadas em traços desenhados à mão livre (vide Figura 2.15). Usando uma interface minimalista, o sistema Teddy introduziu o uso de modeladores 3D a uma audiência não técnica.

Figura 2.15: Sistema *Teddy* em uso em um tablet

A partir desse trabalho, as interfaces baseadas em traços passaram a ser aplicadas em outras áreas, além de modelagem tridimensional, tais como a modelagem de textura, de animações e outras.

Segundo Igarashi [58], as interfaces baseadas em traços não são especificamente adequadas para aplicativos precisos e orientados para produção devido à sua ambigüidade e imprecisão. Entretanto, elas fornecem um ambiente de computação natural altamente interativo para atividades exploratórias e pré-produtivas em várias aplicações gráficas.

Um interface baseada em traços tem uma estrutura que aproveita o poder do desenho à mão livre para obter uma interação fluida entre usuários e computadores na execução de tarefas gráficas. Os usuários expressam suas ideias gráficas como traços de forma livre e o computador toma as ações apropriadas com base na percepção dos traços.

Duas características importantes desse tipo de interface são observadas por Igarashi. A primeira é que a interface necessita identificar se o usuário está desenhando um traço ou arrastando um objeto na tela (vide Figura 2.16). A segunda está vinculada ao que ele denomina “processamento perceptivo".

O processamento perceptivo refere-se a mecanismos que inferem informações de traços simples que são mais ricos que meros símbolos. É uma tentativa de simular a percepção humana, pelo menos em domínios limitados.

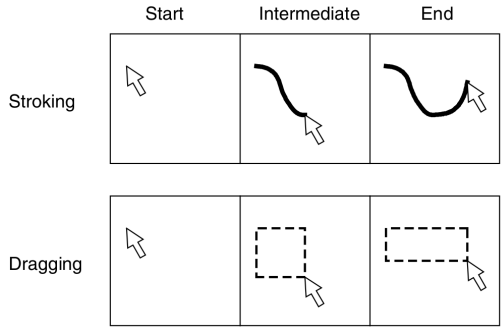


Figura 2.16: *Stroking X Dragging*

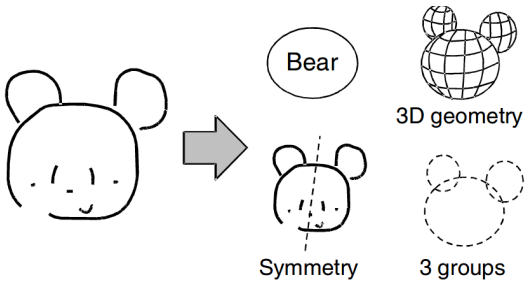


Figura 2.17: Os humanos percebem informações relevantes em um traçado simples

O objetivo do processamento perceptivo é permitir que o usuário execute tarefas complicadas com uma quantidade mínima de controle explícito.

A ideia é inspirada na observação de que os seres humanos percebem informações importantes em desenhos simples, como possíveis relações geométricas entre linhas primitivas ou formas tridimensionais de silhuetas bidimensionais (Figura 2.17).

Nielsen [59], destaca que interfaces baseadas em traços se distanciam da abordagem orientada a objetos padrão para uma abordagem orientada a usuários e orientada a tarefas. Em vez de usar uma sintaxe verbo-substantivo ou substantivo-verbo, tais interfaces são, até certo ponto, livres de sintaxe. A noção básica aqui é que a especificação da ação e do objeto é unificada em um único *token* de entrada, em vez de exigir a composição de um comando de entrada do usuário.

Sua abordagem destaca que a apresentação do conteúdo deve ser informal. O sistema deve exibir os objetos a manipular ou o resultado da computação de maneira informal, usando uma representação limpa e sem excesso de gráficos. Essa apresentação informal é importante não apenas para uma aparência esteticamente agradável, mas também para despertar expectativas adequadas na mente do usuário sobre a funcionalidade do sistema.

Se o sistema fornecer *feedback* em gráficos precisos e detalhados, o usuário naturalmente espera que o resultado da computação seja preciso e detalhado. Por outro lado, se o *feedback* do sistema estiver em apresentação informal, o usuário pode se concentrar na estrutura geral das informações sem se preocupar muito com

os detalhes. Portanto, este é um paradigma importante a ser explorado em nosso sistema.

Bubble clusters

O modelo de interação conhecido como *Bubble Clusters* [4] é uma das principais inspirações da presente proposta para agrupar ícones por distância. Ele foi originalmente proposto para ajudar os usuários a organizar coleções de ícones na tela para que eles possam ser agrupados e reagrupados espacialmente usando o mouse. O artigo também descreve um aplicativo em que traços à mão livre podem ser manipulados. A ideia é usar a proximidade entre os vários objetos na tela, reconhecendo automaticamente os objetos próximos como grupos. Esses grupos são sinalizados visualmente pelo traçado de uma curva em torno deles semelhante a uma bolha de sabão. Essas bolhas também são usadas como *proxies* para os grupos nas interações de layout.

Para compor as bolhas, o método computa uma função radial quadrática $f(r)$ baseada no conceito de *metaballs*. Cada objeto é associado a um ponto que define um campo potencial ao redor dele que diminui com a distância e desaparece além de um determinado limite. A curva em si é simplesmente um isocontorno deste campo que é traçada usando o método *Marching Squares* [60]. Se houver vários objetos no cluster, os campos potenciais de objetos próximos são somados para gerar um limite curvo suave. A função $f(r)$ satisfaz $f(r_0) = 1, f(r_1) = f'(r_1) = 0$, onde r_0 é o raio da bolha de um objeto isolado e r_1 é o tamanho limite (*threshold*) do campo (vide fig. 2.18).

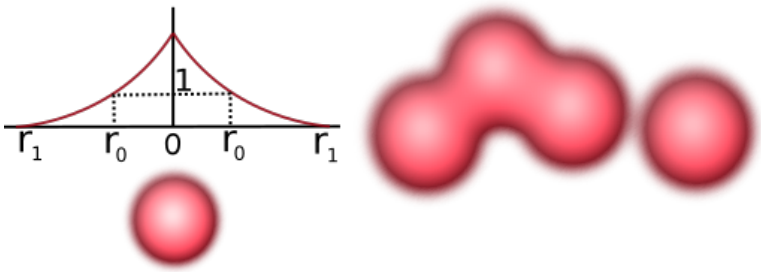


Figura 2.18: Campo potencial [4]. (esquerda) Função f(r) que compõe o campo potencial de uma bolha simples; (direita) Exemplo da soma de campos potenciais.

Essa abordagem de agrupar ícones como uma bolha não funciona apenas como uma indicação visual para indicar a existência de um grupo, mas também como um identificador para interagir com o grupo. A operação mais básica é arrastar um grupo de ícones dentro de uma distância limite: o usuário pode arrastar um grupo inteiro arrastando a bolha. Para evitar agrupamento acidental, um efeito de histerese é aplicado ao arrastar, ou seja, o usuário deve arrastar o cluster a uma

distância ainda menor do que o limite que aciona a fusão (ver Fig. 2.19). Por outro lado, o sistema fornece uma operação de corte explícito para dividir clusters de bolhas usando gestos. O usuário simplesmente desenha um traço de forma livre que atravessa a bolha e o sistema então divide o cluster original em dois clusters separados (Figura 2.20).

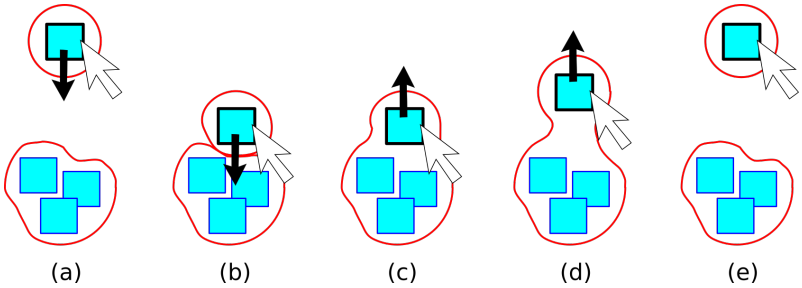


Figura 2.19: Efeito de histerese no *bubble cluster*.

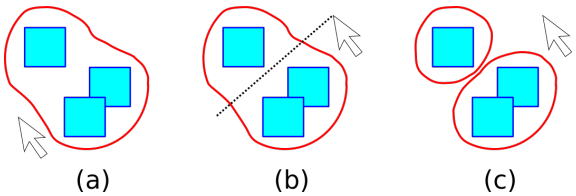


Figura 2.20: Efeito de de corte na bolha usando o desenho de uma linha à mão livre.

Inspirados pela sugestão deste modelo, implementamos um sistema visual de agrupamentos baseado em curvas fechadas suaves que limitam sua fronteira. A Seção 5.2 descreve este sistema e a Figura 2.21 ilustra um exemplo do agrupamento natural em quatro níveis distintos de hierarquia.

Bubble Sets

Quando são permitidas sobreposições de grupos, selecionar uma ou mais formas para operações adicionais também pode se tornar problemático. Uma solução típica é organizar o conteúdo em camadas ou permitir que o usuário manipule explicitamente a ordem de empilhamento entre elementos ou grupos. Outra possibilidade é permitir que o sistema mova automaticamente as formas para um estado sem sobreposição. Por exemplo, em [4], ícones no mesmo cluster podem ser distribuídos por comando para facilitar as operações de seleção.

Bubble Sets [61] é uma técnica de visualização de dados para destacar conjuntos de pontos que estão irregularmente distribuídos em um gráfico ou mapa existente. Como os *bubble clusters*, ela usa uma curva em forma de bolha para circundar os elementos pertencentes ao mesmo grupo, mas não exige que estejam próximos

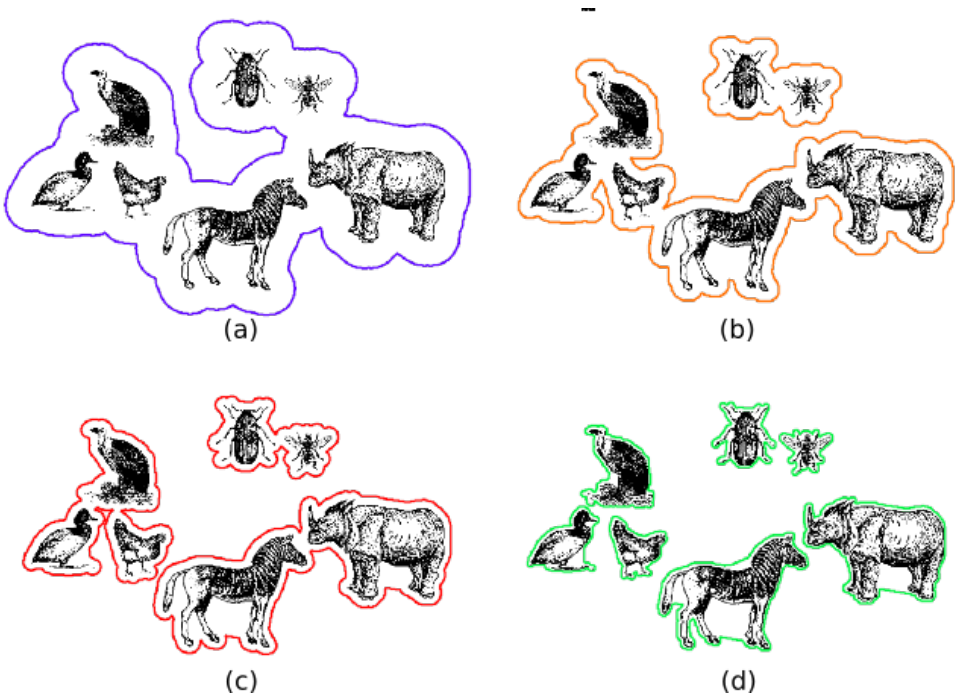


Figura 2.21: Exemplo de visualização de agrupamentos semânticos em quatro níveis de hierarquia. A curva azul (a) indica o nível mais abrangente contendo todos os *patches* com figuras de animais; em (b), as duas curvas laranjas indicam os dois filos (cordados e artrópodes); as três curvas vermelhas (c) indicam as classes (aves, mamíferos e insetos); em (d) as curvas verdes representam o nível mais baixo e contornam cada animal em separado.

uns dos outros. Os autores propõem uma abordagem heurística para construir bolhas esteticamente agradáveis. A ideia é construir primeiro uma árvore geradora conectando todos os elementos do conjunto de forma que não apenas os pontos (vértices) sejam usados como centros das *metaballs*, mas também algumas arestas selecionadas que são necessárias para manter a curva conectada. A conexão do contorno é garantida por meio de arestas virtuais que contribuem para a distribuição de energia do conjunto. Da mesma forma, algumas heurísticas são empregadas para garantir que as curvas de bolha não se cruzem ou que envolvam elementos não pertencentes ao conjunto como ilustra a Figura 2.22.

No Capítulo 5 aprofundamos o uso de *ball trees* para layout de formas e lidamos com esses problemas de sobreposições de grupos apresentando técnicas para agrupar automaticamente as formas hierarquicamente por distância e deslocar as formas na página para evitar sobreposições.

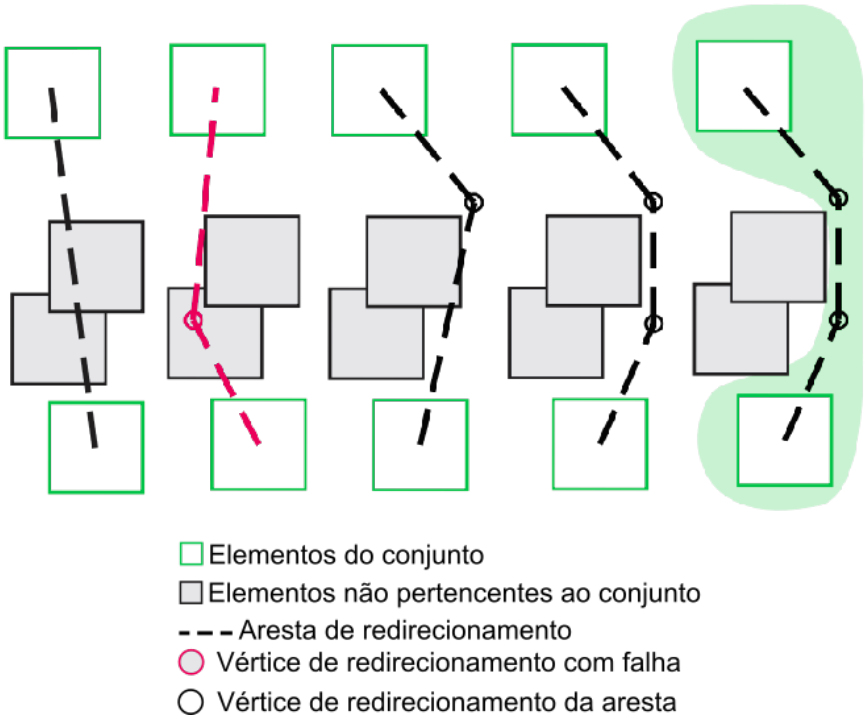


Figura 2.22: (Da esquerda para direita): Uma aresta virtual passando por um elemento é detectada. Um novo ponto de controle é criado em um canto da caixa delimitadora do obstáculo e o teste é repetido. Quando o teste falha, o canto diagonalmente oposto é usado e nenhum obstáculo é encontrado. Pontos de controle adicionais são criados nos cantos para direcionar a aresta ao redor do obstáculo. O conjunto final de arestas virtuais contribui para o cálculo da energia, permitindo que o contorno definido evite os obstáculos e permaneça conectado.

Capítulo 3

Construção de *Ball trees*

Embora as *ball trees* tenham sido propostas como um tipo de estrutura especializada na indexação de dados em espaços métricos (*metric-trees*) [62, 63] e usadas como dispositivos de decomposição espacial, não nos concentramos em algoritmos de construção propostos para esse fim. Em vez disso, focamos no seu uso em tarefas como detecção de colisões e diversas consultas relacionadas à distância com métricas euclidianas, usando o trabalho seminal de Omohundro sobre construção de *ball trees* [9] como parâmetro.

3.1 Algoritmos de construção de *ball trees*

Para os propósitos deste trabalho, seis algoritmos de construção de *ball trees* foram implementados [64, 65]. Os três primeiros seguem a discussão de Omohundro [9] e nos referimos a eles como algoritmos de construção *KD*, *Online* e *Bottom up*. Notamos que, embora outros dois algoritmos tenham sido testados por Omohundro, ele considerou esses três como os de melhor desempenho.

Nossas implementações diferem daquelas descritas por Omohundro [9] principalmente em dois aspectos: o primeiro é que os nós folha consistem de bolas, em vez de pontos. O segundo, é o uso de uma estratégia que chamamos de otimização *Enclosing Leaves (EL)*, onde propomos o uso de nós internos mais justos. Uma vez que uma *ball tree* é construída, uma etapa adicional é realizada para garantir que cada nó interno corresponda ao menor círculo contendo todas as folhas da subárvore enraizada naquele nó. Notamos que encontrar o menor círculo que contém uma coleção de círculos pode ser feito no tempo de complexidade $O(n)$ [66–69], e assim esta etapa leva $O(n^2)$ para uma árvore com n nós folha. A Figura 3.1 mostra um exemplo.

A seguir, apresentamos ainda três novos algoritmos para construção de *ball trees*. Um algoritmo quadrático (Q) e outros dois que utilizam as vizinhanças de Voronoi [70, 71], denominados *V0* e *V1*.

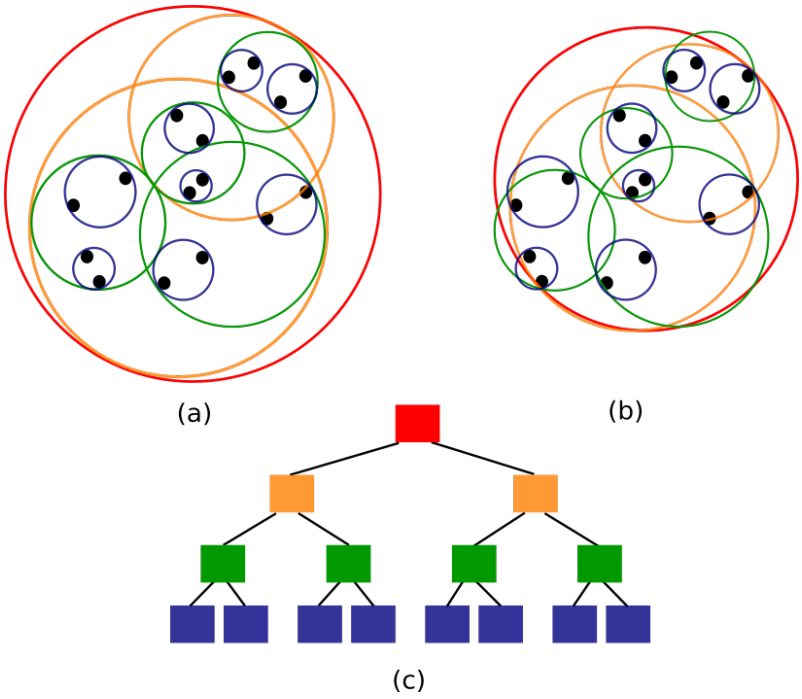


Figura 3.1: Representação típica de *ball tree* binária. (a) mostra uma *ball tree* onde os nós internos envolvem os dois filhos imediatos, enquanto em (b) eles envolvem todos os nós folha descendentes. (c) descreve a topologia da árvore, que é a mesma para ambas as árvores.

3.1.1 Algoritmo KD

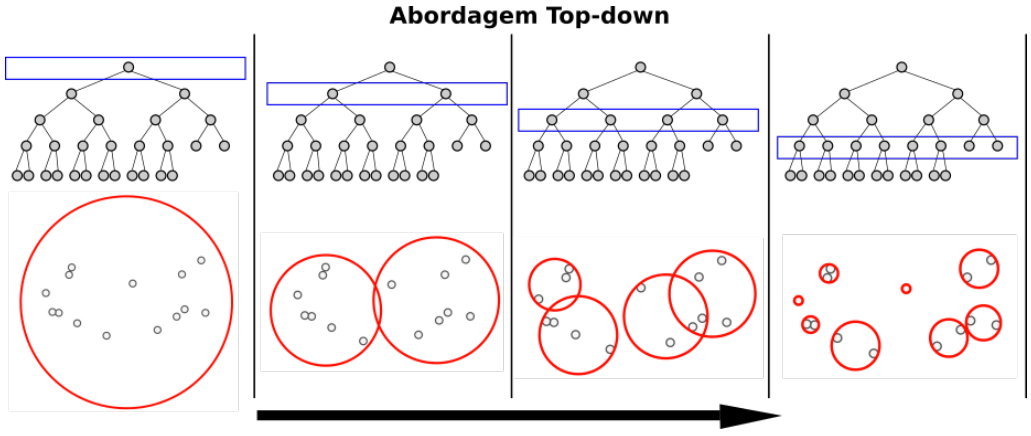


Figura 3.2: Construção de *ball tree* com a abordagem *Top-down*.

Este é um algoritmo que utiliza a abordagem *top-down* na construção da *ball tree* (fig. 3.2), semelhante ao método proposto por Friedman [13] para construir *KD trees*. De acordo com Omohundro [9], se o conjunto de dados é grande e relativamente uniforme, então a abordagem KD é rápida, simples e adequada. Mas, se os dados estiverem agrupados ou esparsos ou tiverem estrutura extra, a abordagem KD tende a não refletir essa estrutura em sua hierarquia.

Inicialmente, o algoritmo determina a bola envolvente de todo o conjunto e a

estabelece como o nó raiz da *ball tree*. A árvore é construída dividindo recursivamente cada conjunto em dois subconjuntos com aproximadamente o mesmo número de bolas.

Em cada etapa, o algoritmo determina o eixo de maior dispersão e o valor médio em relação a esse eixo é utilizado para realizar a divisão, considerando as coordenadas dos centros das bolas (vide Figura 3.3). As bolas envolventes de cada um dos dois subconjuntos gerados, são associadas a dois novos nós na *ball tree* e o processo continua até que todos os subconjuntos tornem-se unitários. As bolas correspondentes a estes elementos únicos serão as folhas da *ball tree*.

A complexidade geral deste algoritmo é $O(n \log n)$. No entanto, adicionar a otimização *EL* aumenta sua complexidade para $O(n^2)$.

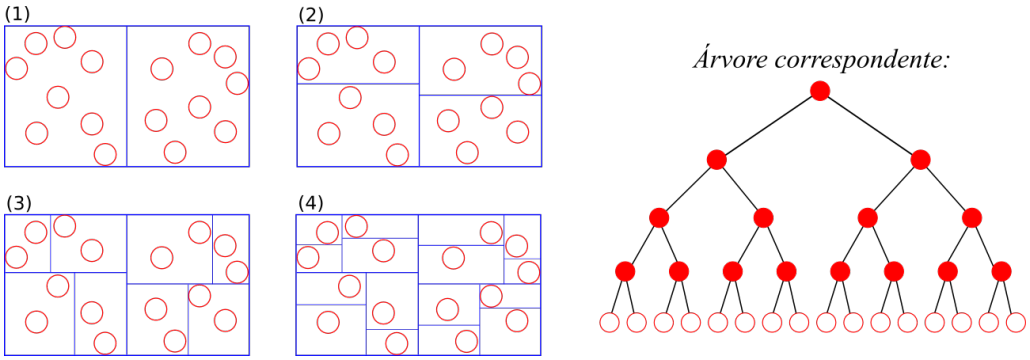


Figura 3.3: Subdivisão de um conjunto esparso de bolas induzida por uma *KD tree*.

3.1.2 Algoritmo Online

Este algoritmo constrói a árvore de forma incremental. Cada nova bola será um nó folha da árvore. O algoritmo tenta inserir cada nova bola, buscando a localização ideal, ou seja, aquela que menos contribui para o aumento da área total da árvore.

Seja N o novo nó a ser inserido e A o nó considerado pelo algoritmo como ponto ideal de inserção de N.

Então, quando o algoritmo insere a nova bola na árvore, N se tornará o irmão de A sob o novo pai P (veja a Figura 3.4). Com isso, a bola envolvente que contém P (nó ancestral) deverá ser redimensionada, considerando a expansão da área com a inserção de N. Este ajuste ocorre recursivamente até a raiz. Ao realizar esta operação, a árvore tem um aumento em sua área total.

A Figura 3.5 mostra a situação em que a bola 3 deve ser inserida. Com 1 e 2 já contidas na árvore, o ponto ideal de inserção deve ser aquele que minimiza o aumento da área total. Claramente, neste exemplo, a escolha ideal é mostrada na fig. 3.5.b que exibe a menor bola envolvente.

Além da área da nova folha, há duas contribuições para o aumento da área total

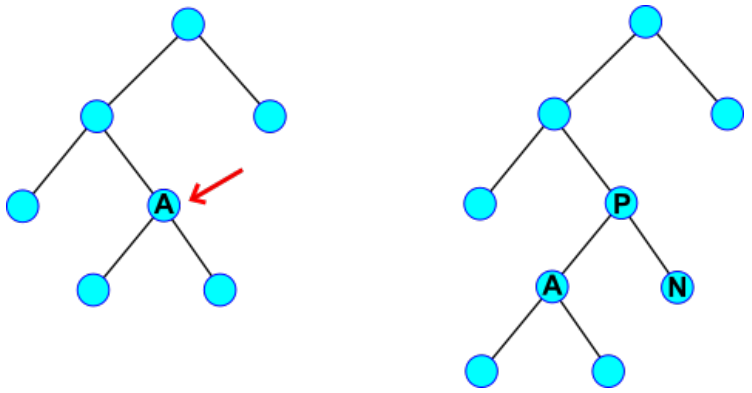
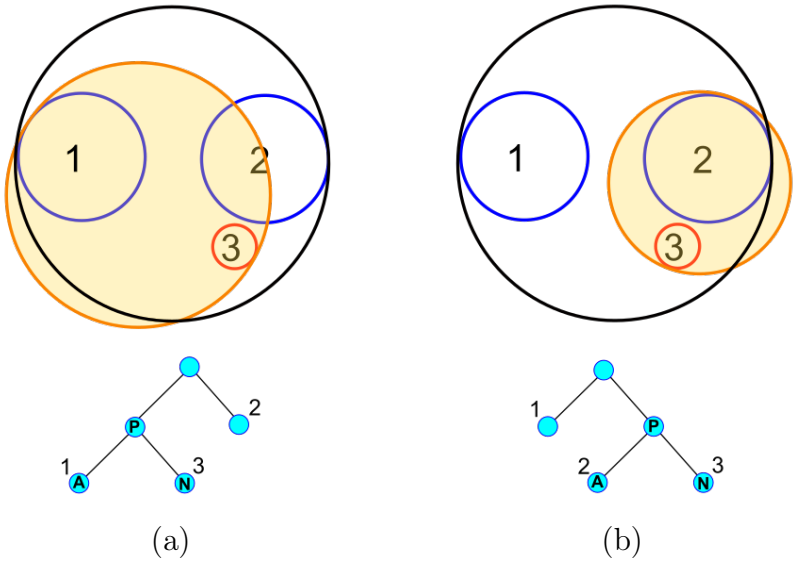


Figura 3.4: Ilustração do algoritmo *online*. A inserção de um novo nó N na posição ideal da árvore.

da árvore: a área do novo nó pai e a quantidade de expansão de área nas bolas ancestrais acima dele.



(a) (b)

Figura 3.5: Ilustração da tentativa de inserir a bola 3 na *ball tree* seguindo o critério da minimização do aumento da área total. (a–b) estrutura da árvore, caso o pareamento fosse com a bola 1 ou com a bola 2, respectivamente.

Para possibilitar a busca pelo melhor local de inserção usamos a estrutura de uma fila de prioridade ordenada por sua área de expansão ancestral.

Essa estrutura de *ball tree* tende a agrupar os nós mais próximos como irmãos e deixar aqueles que estão mais distantes ou que são muito maiores próximos ao topo da árvore e em diferentes ramos. Novas bolas que estão longe das bolas existentes também acabam perto do topo. Desta forma, a estrutura da árvore tende a refletir a estrutura de agrupamento das folhas. Quando há necessidade de remover um nó, removemos seu pai e corrigimos as bolas envolventes dos ancestrais. A *ball tree* é construída em $O(n(\log n)^2)$ e aumenta para $O(n^2)$ com otimização *EL*.

3.1.3 Algoritmo Bottom up

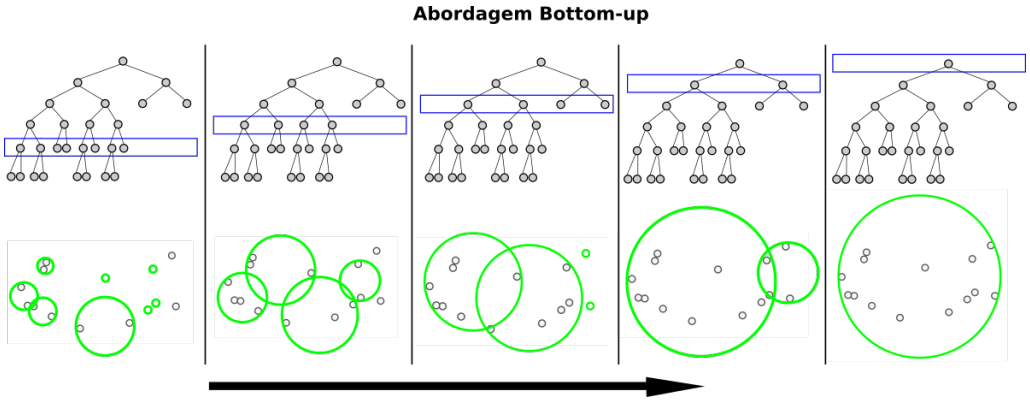


Figura 3.6: Construção de *ball tree* com a abordagem *Bottom up*.

O algoritmo *bottom up* utiliza uma abordagem de construção da *ball tree* de baixo para cima. A ideia é partir do conjunto de bolas iniciais que aproximam a forma do objeto, ou seja, as folhas da árvore (fig. 3.6).

A heurística *bottom up* encontra repetidamente as duas bolas cuja bola envolvente tem a menor área, as torna irmãs e insere a bola pai de volta no conjunto. Este novo conjunto agora é menor, pois considera as bolas ainda não pareadas e apenas as bolas envolventes dos pares formados.

Sabemos que *ball tree* é uma estrutura ideal para determinar o melhor par para cada bola [9]. Portanto, utilizamos uma *ball tree* dinâmica auxiliar construída com o algoritmo *online* para manter as bolas ainda não pareadas.

Inicialmente, o algoritmo encontra para cada nó, o par de bolas que produz a menor bola envolvente. Esses pares são mantidos em uma fila de prioridade e a cada iteração os melhores parceiros são recalculados com o auxílio da árvore de inserção auxiliar. Os nós que já foram pareados são removidos da árvore de inserção e quando resta apenas um nó na árvore de inserção, a construção é concluída em $O(n^2(\log n)^2)$.

A Figura 3.7 ilustra um exemplo didático sem considerar a otimização *EL*. Um dado conjunto constituído de 8 bolas (Fig. 3.7.a) e alguns dos primeiros pares a serem considerados na fila de prioridades ranqueada por ordem crescente dos círculos envolventes (Fig. 3.7.b). Na sequência, as iterações na árvore auxiliar para compor a *ball tree* (Fig. 3.7.c). Note que os nós pareados segundo o critério do menor círculo envolvente estão em vermelho. Ao final, sobra somente um nó: a raiz da *ball tree* construída.

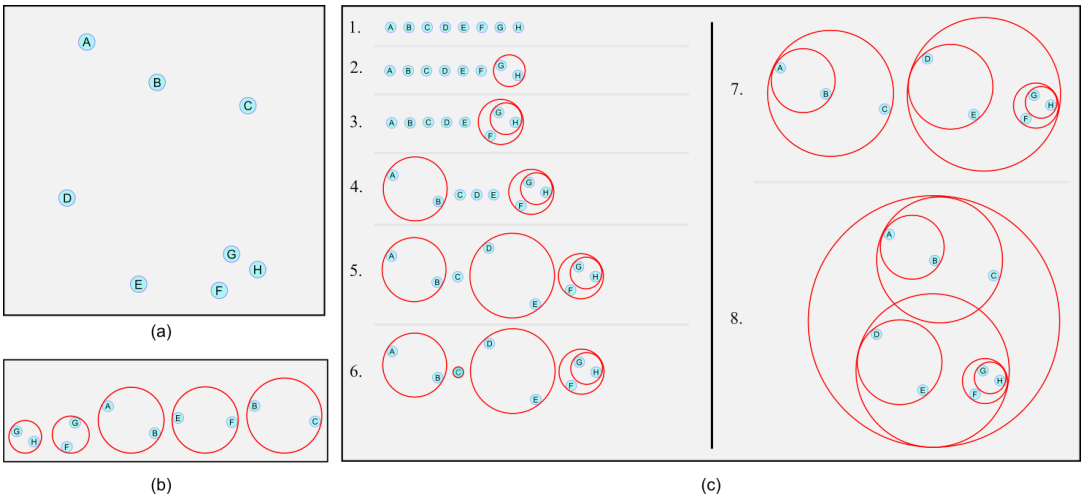


Figura 3.7: Ilustração da estratégia *bottom up*. (a) Exemplo de um conjunto simples com apenas 8 bolas; (b) Alguns dos primeiros pares a serem considerados na fila de prioridades; (c) Construção da *ball tree* com *bottom up*. Nós da árvore auxiliar disponíveis para o pareamento a cada uma de 8 iterações. Os nós vermelhos são selecionados e inseridos na *ball tree*.

3.1.4 Algoritmo Q

Neste algoritmo, a coleção de bolas de entrada B é inicialmente arranjada em ordem crescente de raio. Então, o par ótimo para cada elemento nesta ordem é determinado, ou seja, para B_i, todos os elementos B_j tal que $j > i$, e tal que j ainda não foi pareado, são examinados e o elemento que produz o menor círculo envolvente é escolhido.

Os nós internos formados pelo pareamento de bolas de B são novamente ordenados por raio e uma nova coleção B' é criada, e apenas uma fração $0 < \alpha \leq 1$ destas com os menores raios é mantida, enquanto os outros pares são dissolvidos e seus filhos imediatos também são colocados em B'. Todo o processo é então repetido, agora considerando a nova coleção B', até que B' contenha uma única bola: a raiz da árvore.

A ideia de fracionar B, a partir de α, é priorizar o posterior pareamento entre bolas de menor dimensão, descartando aquelas que contribuíram muito com o aumento da área total. Este comportamento tende a ser favorável quando o conjunto de bolas é muito compacto, ou seja, quando há muitas bolas próximas umas das outras.

A complexidade deste algoritmo é $O(n^2)$, mas a constante depende de α: valores maiores resultam em um número menor de repetições.

A Figura 3.8 exibe um exemplo com algumas iterações do algoritmo Q para a construção de uma *ball tree* com otimização *EL*. O quadro da Fig. 3.8.a exibe o conjunto com apenas 6 bolas de entrada. Inicialmente, o algoritmo ordena todo o conjunto pelo raio (como visto no diagrama da Fig. 3.8.b) e, posteriormente, inicia

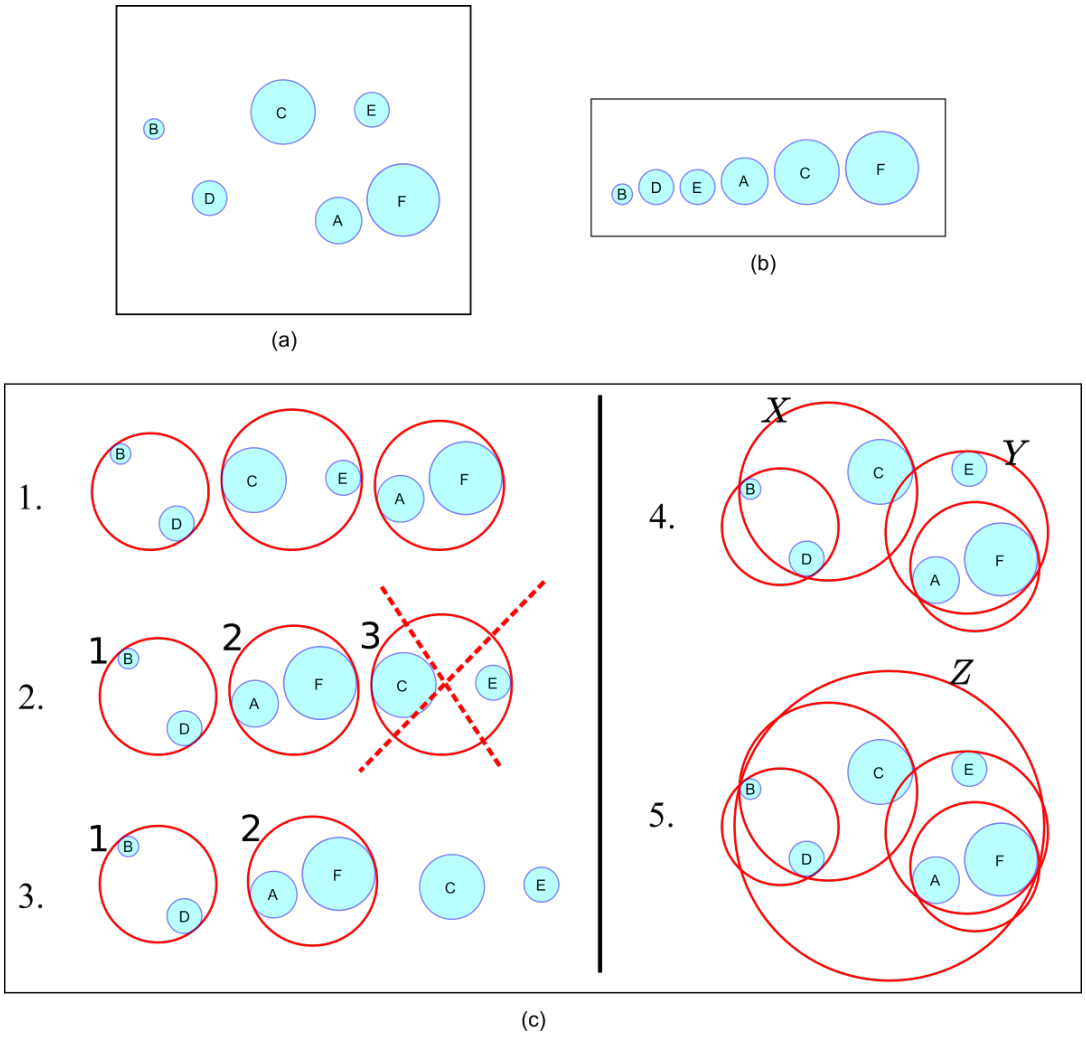


Figura 3.8: Exemplo de construção de *ball tree* com o algoritmo Q e $\alpha = 0,6$. (a) Conjunto de bolas inicial; (b) Ordenação do conjunto; (c) Ilustração das iterações do algoritmo.

o pareamento segundo o critério da menor bola envolvente. Observe a Figura 3.8.c e considere as iterações do algoritmo Q com $\alpha = 0.6$. Na iteração 1 os pareamentos realizadas são: (B,D), (C,E) e (A,F). Após a reordenação no passo 2, somente as duas primeiras bolas resultantes dos pareamentos (1 e 2) são mantidas. O passo 3 exibe o conjunto B' com as bolas {1,2,C,E}. Na etapa 4 as bolas X e Y são formadas a partir do pareamento das bolas remanescentes do conjunto. E, finalmente, a bola Z torna-se a raiz da *ball tree* gerada.

3.1.5 Algoritmo V0

De modo semelhante ao algoritmo Q, as bolas de entrada $\in B$ são colocadas em uma lista L ordenada por raio.

Entretanto, ao invés de considerar todos os possíveis pareamentos entre as bolas (como em Q), a tentativa de parear uma dada bola B_i é feita apenas com as bolas

que estão mais próximas a ela. Para tanto, usamos o critério de vizinhança, a partir da construção de um diagrama de Voronoi.

Então, este algoritmo constrói o diagrama de Voronoi V, cujos sítios correspondem aos centros das bolas $\in L$.

A Figura 3.9.a mostra um exemplo onde as bolas ordenadas são $L = \{A, B, C, D, E\}$.

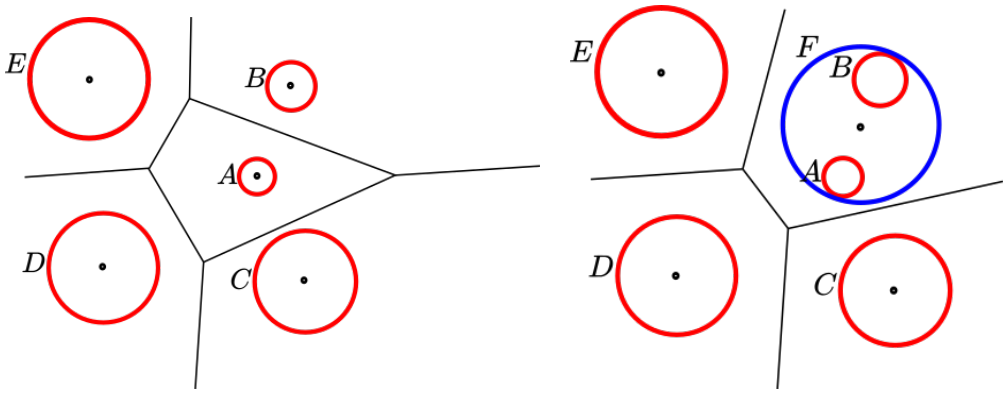


Figura 3.9: Exemplo de inserção de um nó na *ball tree* com o algoritmo V0. (a) O conjunto de bolas inicial, (b) a iteração seguinte, quando o novo par é formado e o diagrama de Voronoi é reconstruído.

Para cada elemento $V_i \in V$, existe um conjunto de pontos que representa sua vizinhança $N(i)$. Então, para cada elemento V_i, todos os sítios $j \in N(i)$ são examinados, ou seja, as respectivas bolas correspondentes em L. O par ótimo é determinado ao produzir seu menor círculo envolvente. As bolas i e j são removidas da lista L e a nova bola pai é inserida.

Na fig. 3.9.b observe que V_1 corresponde a bola A e $N(1) = \{B, C, D, E\}$. Então, o algoritmo determina F, como a menor bola envolvente entre os possíveis pares com A. Esta nova bola é inserida na *ball tree* e o algoritmo prossegue.

Assim, a lista L é ordenada novamente e o algoritmo reconstrói V. Enquanto houver bolas em L, o algoritmo continua até terminar de construir a *ball tree* a partir da última bola.

A complexidade do algoritmo *V0* é $O(n^2 \log n)$.

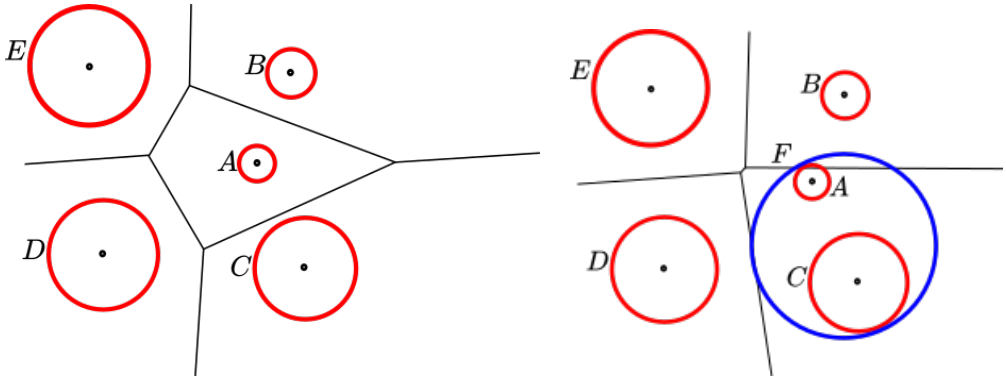
3.1.6 Algoritmo V1

Usando a mesma abordagem de *V0*, este algoritmo seleciona o par ótimo de bolas na lista L a partir da comparação com vizinhança de Voronoi em cada iteração, ou seja, examinando todos os elementos de $N(i)$ para cada V_i .

No entanto, os elementos em L não são ordenados. Em vez disso, eles são embaralhados apenas uma vez no início, para que os elementos sejam sempre tomados em

ordem aleatória. A ideia por trás desta heurística é evitar os casos em que algumas bolas maiores sejam pareadas com outras bolas muito grandes. Isto pode ocorrer por estarem localizadas na periferia do conjunto, ou por falta de melhor opção nos estágios iniciais do pareamento.

No exemplo da Figura 3.10), $L = \{C, B, E, D, A\}$, ou seja, V_1 equivale ao centro de C. Desta forma, o pareamento realizado é (C,A), onde F é a menor bola envolvente.



(a) (b)

Figura 3.10: Exemplo de inserção de um nó na *ball tree* com o algoritmo V1. (a) O conjunto de bolas inicial, supondo $L = \{C, B, E, D, A\}$ (b) na iteração seguinte, a bola F é inserida na *ball tree*.

O procedimento para atualizar a lista L é o mesmo usado em *V0* e a *ball tree* é construída em $O(n^2 \log n)$

3.2 Algoritmos de consulta

Tipicamente, a avaliação de um algoritmo de construção de *ball tree* particular inclui: análise de sua complexidade e medições de certas características das árvores construídas a partir de entradas variadas. Como mencionado anteriormente, o padrão *de facto* para este tipo de medida é a área total de todas as bolas da árvore. No presente estudo, no entanto, adicionamos outras medidas empíricas a essa mistura, ou seja, realizamos experimentos onde as árvores construídas com cada algoritmo listado na Seção 3.1 são usadas para responder a uma variedade de consultas de proximidade.

3.2.1 Estratégia de processamento de consultas

Consultas relacionadas à distância entre duas árvores A e B são rotineiramente computadas por uma descida combinada das duas árvores, analisando um par de nós (a, b) de cada vez, onde $a \in A, b \in B$. O processo claramente deve começar com

os dois nós raiz, mas a partir deste ponto, a escolha do próximo par a ser examinado deve ser informada por alguma heurística. A ideia é priorizar pares de nós com maior chance de melhorar a solução para a consulta.

Nossos algoritmos de consulta envolvendo duas árvores bolas A e B seguem uma estratégia *branch-and-bound*, onde, a cada passo, um par (a, b) de nós é considerado apenas se pareamentos de seus descendentes (se houver) ainda podem melhorar a função objetivo da busca. A travessia do espaço de busca é realizada com a ajuda de uma fila de prioridade P ordenada em relação a alguma função $f_P(a, b)$. Esta função classifica um par de nós (a, b) de acordo com sua chance de conter a solução entre pares construídos com a e b ou seus descendentes. Como requisito adicional, $f_P(a, b)$ deve suportar um término antecipado da busca, ou seja, deve ser possível determinar se P ainda contém pares dignos de exame após examinar o par à frente da fila.

Algorithm 1 - Tree Distance

Input: A,B {ball trees}
Input: f_P {função de ranqueamento da fila de prioridade}

```
1:  P ← fila de prioridade ordenada por f_P
2:  P.push((root(A), root(B)))
3:  minDist ← ∞
4:  while P ≠ ∅ do
5:     (a, b) ← P.pop()
6:     if dist(a, b) > minDist then
7:        return minDist
8:     end if
9:     if isLeaf(a) and isLeaf(b) then
10:       minDist ← dist(a, b)
11:    else if isLeaf(a) and not isLeaf(b) then
12:       P.push((a, b.left))
13:       P.push((a, b.right))
14:    else if not isLeaf(a) and isLeaf(b) then
15:       P.push((a.left, b))
16:       P.push((a.right, b))
17:    else
18:       P.push((a.left, b.left))
19:       P.push((a.left, b.right))
20:       P.push((a.right, b.left))
21:       P.push((a.right, b.right))
22:    end if
23: end while
24: return minDist
```

Como exemplo de algoritmo para computar consultas de proximidade, considere o Algoritmo 1, que computa a menor distância entre A e B, ou seja, a menor distância entre duas folhas de $a \in A, b \in B$. Claramente, a cláusula de término

antecipado expressa na linha 6 implica que nenhum par de bolas (a', b') ainda na fila pode produzir um par de folhas cuja distância seja menor que $minDist$. Em outras palavras, se a' é o conjunto de todas as folhas da árvore enraizadas em a (e analogamente para b'), e $dist(a, b)$ representa a distância euclidiana entre a e b, então definimos

$$d_{min}(a, b) = \min\{dist(a'_i, b'_j) \mid a'_i \in a, b'_j \in b\},$$

e requer que $f_P(a, b) \geq d^+_{min}(a, b) \geq d_{min}(a, b)$, onde d^+_{min} é alguma heurística que limita o valor real de d_{min}.

Por outro lado, se dois pares (a_1, b_1) e (a_2, b_2) são tais que $d^+_{min}(a_1, b_1) = d^+_{min}(a_2, b_2)$, pode ser vantajoso atribuir uma prioridade diferente a eles. Isso pode ser feito adicionando um segundo critério para desempate. Propomos usar uma estimativa de distância máxima $d^+_{max}(a, b)$ para esse propósito, ou seja, uma heurística que limita

$$d_{max}(a, b) = \max\{dist(a'_i, b'_j) \mid a'_i \in a, b'_j \in b\}.$$

A lógica é que um par com d^+_{max} mais baixo deve ser considerado antes de outro com d^+_{max} mais alto.

Alguns autores propõem usar uma função de ranqueamento baseada na área de interseção entre os dois nós [72], mas esse esquema não estabelece uma distinção entre pares que não se cruzam. Uma escolha mais natural para $d^+_{min}(a, b)$ é $dist(a, b)$, mas podemos fazer melhor se a e/ou b forem nós internos considerando a distância mínima entre os pares possíveis de seus filhos. Da mesma forma, $d^+_{max}(a, b)$ pode ser estimado pela distância máxima entre a e b, mas uma estimativa melhor (mais justa) pode ser a menor das distâncias máximas para até quatro pares de seus filhos. Em nossos experimentos, uma função de ranqueamento f_P usando a distância mínima entre os filhos e desempates com a distância máxima entre os filhos deu os melhores resultados.

3.2.2 Outras consultas

Além da consulta Tree Distance (TD), também realizamos testes para as seguintes consultas:

- ***Tree intersection (TI)*** - Encontra um par de nós folha (a, b) tal que $a \cap b \neq \emptyset$. Este algoritmo é análogo ao Algoritmo 1, exceto que (1) pares de nós retirados de P que não se intersectam são descartados imediatamente; (2) se for encontrado um par de nós folha que se intersectam, o algoritmo retorna `true`; e (3) quando P fica vazio, `false` é retornado.

- ***Point Intersection (PtI)*** - Dado um ponto p, encontra um nó folha a, tal que $a \supset p$. Isso é análogo ao ***TI***, considerando p como uma árvore com um único nó.

- ***Distance to Point (DtP)*** - Dado um ponto p, encontra um nó folha a, tal que $dist(a, p)$ é mínima. Isso é análogo ao ***TD***, considerando p como uma árvore com um único nó.

- ***Line intersection (LnI)*** - Dada uma reta r, encontra um nó folha a, tal que $a \cap r \neq \emptyset$. Este algoritmo é semelhante ao ***TI***, exceto que (1) f_P usa estimativas de distância entre as bolas e a reta r; (2) um nó retirado de P que não intercepta r é descartado imediatamente; (3) se um nó folha cruzando r for encontrado, o algoritmo retornará `true`; e (4) quando P fica vazio, `false` é retornado.

- ***Distance to Line (DtL)*** - Dada uma reta r, encontra um nó folha a tal que $dist(a, r)$ é mínima. Este algoritmo adapta ***TD*** da mesma forma que ***LnI*** é uma adaptação de ***TI***.

Deve ser mencionado que Omohundro [9] também propõe um algoritmo para consultas DtP que não requer uma fila de prioridade. Em vez disso, é uma simples busca recursiva em profundidade onde, em cada etapa, o filho mais próximo do ponto de consulta é percorrido primeiro. No entanto, em nossos experimentos, o critério de seleção global de nós fornecido por uma fila de prioridade produz melhores resultados em termos de número de nós visitados. É claro que o custo de manter uma fila de prioridades para esse tipo de consulta pode compensar os ganhos assim obtidos.

No entanto, a adoção de filas de prioridade em todos os algoritmos de consulta fornece um padrão uniforme para julgar o impacto do algoritmo de construção nos resultados da consulta.

3.3 Experimentos de construção e consulta

Para avaliar os algoritmos de construção descritos na Seção 3.1, realizamos uma série de experimentos diferentes. Além de avaliar diretamente esses algoritmos medindo o tempo de construção e a área total da árvore de bola, também realizamos experimentos para avaliar quão bem as árvores construídas se comportam em relação às consultas de proximidade descritas na Seção 3.2. Como mais de $3.5M$ de dados foram coletados em 218 diferentes tipos de experimentos, aqui comentamos apenas as descobertas mais relevantes [73].

Todos os algoritmos foram codificados em JavaScript e executados em um navegador Chrome v90.04 em uma estação de trabalho com processador Intel i7 1.8GHz (8 núcleos) com 12 GB de memória principal, rodando Ubuntu Linux v20.04 64 bits.

3.3.1 Dados de teste

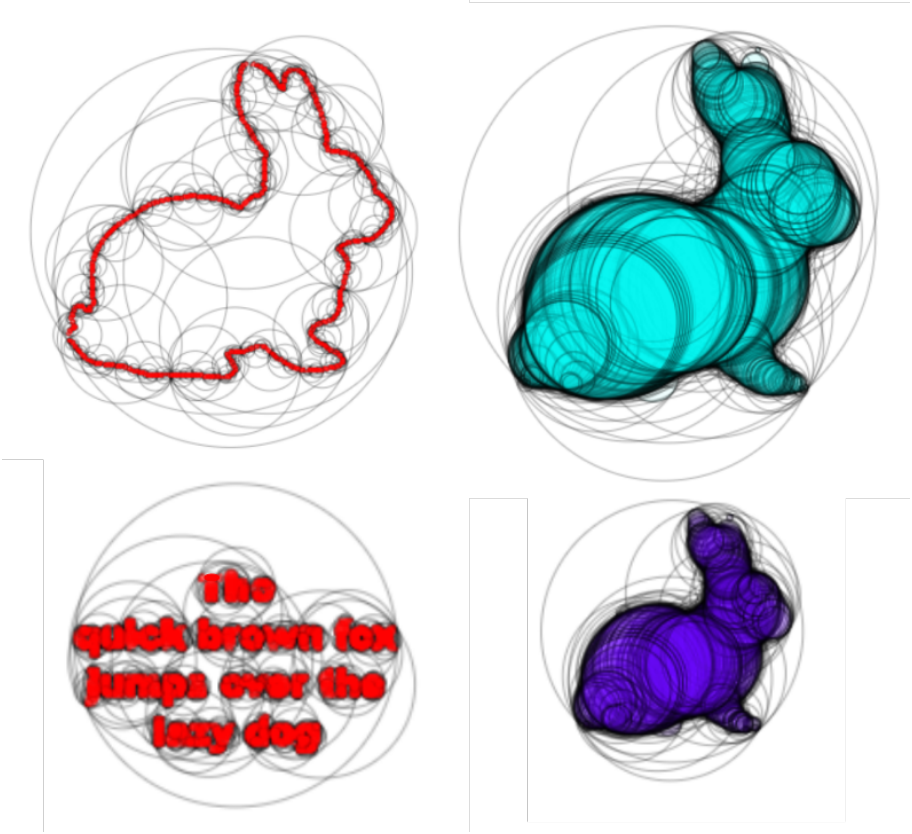
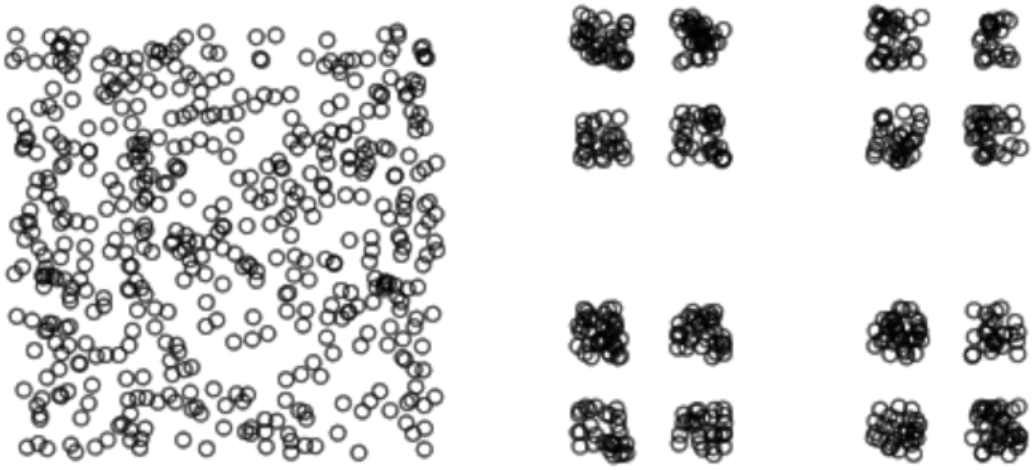


Figura 3.11: Dados de teste: *Ball trees* de Aproximação de formas - bolas coloridas (folhas) cobrem a forma do objeto.

Para os experimentos, usamos conjuntos de bolas de entrada que se enquadram em duas classes gerais de distribuições: (1) *uniforme (aleatório e distribuições de Cantor)*, semelhantes aos usados em [9]; e (2) *aproximações de forma*, ou seja, conjuntos de bolas que aproximam formas 2D conforme discutido na Seção 2.2 (veja fig. 3.11).

As bolas nas coleções aleatórias têm raio 0.01 distribuídas dentro de um quadrado de tamanho unitário. Foram utilizadas duas coleções com 500 bolas cada, denominadas 500-R e 500-C de acordo com a distribuição (uniforme ou Cantor) conforme Figura 3.12.

As quatro coleções de bolas do tipo *aproximações de forma* são *proxies* de imagens, símbolos e texto que podemos extrair de documentos: *Pangram*, *Border*, *Rabbit 128* e *Rabbit 256*. As bolas em cada coleção foram produzidas a partir de imagens



(a) (b)

Figura 3.12: Exemplo de distribuições com 500 bolas uniformes. (a) Aleatória, (b) Cantor.

originais usando nosso próprio algoritmo de aproximação de bola com $\epsilon = 0.01$ (veja Seção 2.2) e são mostradas na Fig. 3.11, enquanto as estatísticas relevantes são mostradas em Table 3.1.

Tabela 3.1: Coleções de bolas *aproximação de forma*

Formas	Rabbit 128	Rabbit 256	Pangram	Border
Resolução da imagem	124×122	255×250	144×81	247×243
# Bolas	190	391	997	$1,254$

3.3.2 Metodologia de teste de consulta

Para fornecer uma amostragem justa das consultas, estas foram realizadas considerando uma área de teste fixa equivalente a um quadrado de lado 8. Este espaço é subdividido em uma grade regular de 40×40 (*query grid*), resultando em 41×41 pontos (fig. 3.13).

Todas as árvores de bolas são dimensionadas para que seu nó raiz tenha raio unitário. Para cada tipo de consulta, um objeto é colocado no centro da grade, enquanto o objeto de comparação varia de posição, gerando um total de 1681 consultas individuais. Para consultas envolvendo uma árvore e um ponto, o ponto permanece fixo no centro enquanto a árvore é transladada para que o centro da raiz coincida com cada ponto da grade. Da mesma forma, para consultas envolvendo uma linha, esta é colocada centralizada no eixo vertical da grade.

Também distinguimos amostras de consulta obtidas quando os objetos estão próximos daqueles obtidos quando estão distantes. Usamos o termo *close* para nos

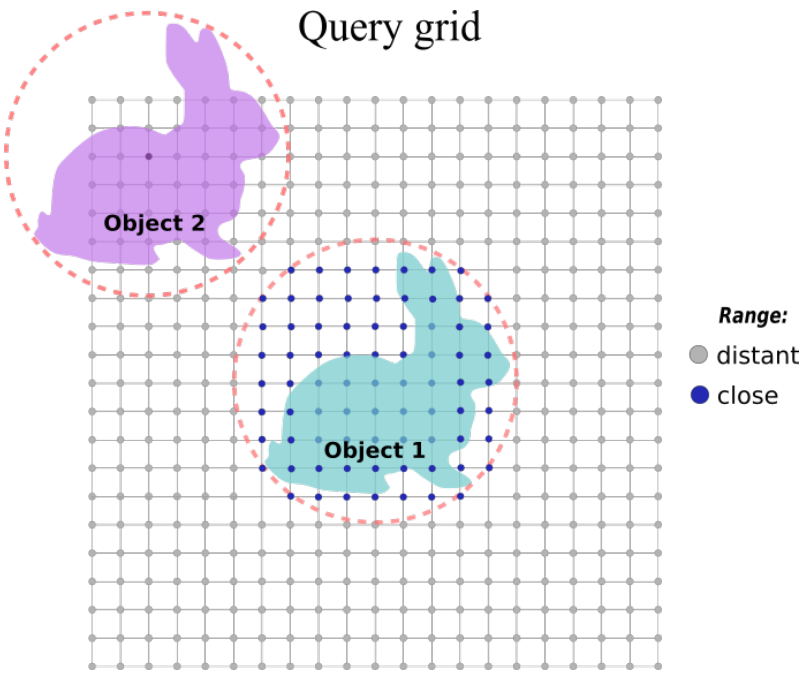


Figura 3.13: Query grid - Exemplo de consulta para dois objetos do mesmo tipo.

referirmos a consultas feitas quando há pelo menos alguma chance de interseção entre os dois objetos, caso contrário classificamos a consulta como *distant*. Essa distinção é útil, pois a distância geral entre os objetos afeta claramente a complexidade de tempo da consulta. Por exemplo, uma consulta PtI *distant* pode ser respondida com apenas uma comparação.

Como todos os algoritmos de consulta dependem de uma fila de prioridade, a complexidade de tempo de execução de cada consulta individual é medida em termos do número total de operações *push* e *pop* executadas na fila (*total Ops*). Isso é razoável, pois essas são as operações mais caras executadas para cada iteração do loop (veja Algoritmo 1), tendo complexidade $O(\log n)$ cada, onde n é o tamanho da fila.

3.3.3 Construção da árvore

Como realizamos experimentos multivariados e abundantes em dados, a maioria das apresentações utiliza uma série de pequenos gráficos conhecida como *small multiples* que respondem diretamente por meio de comparações visuais de mudanças, das diferenças entre objetos e do escopo das alternativas empregadas (Vide exemplo na Fig. 3.14).

Inicialmente, apresentamos um estudo do tempo de execução de algoritmos de construção de *ball trees*. Foram utilizadas duas versões do algoritmo Q, com $\alpha = 0.2$ e 0.5, respectivamente. Todos os algoritmos, exceto o Bottom up, exibem uma velocidade similar, com a construção KD rodando mais rápido na maioria dos testes

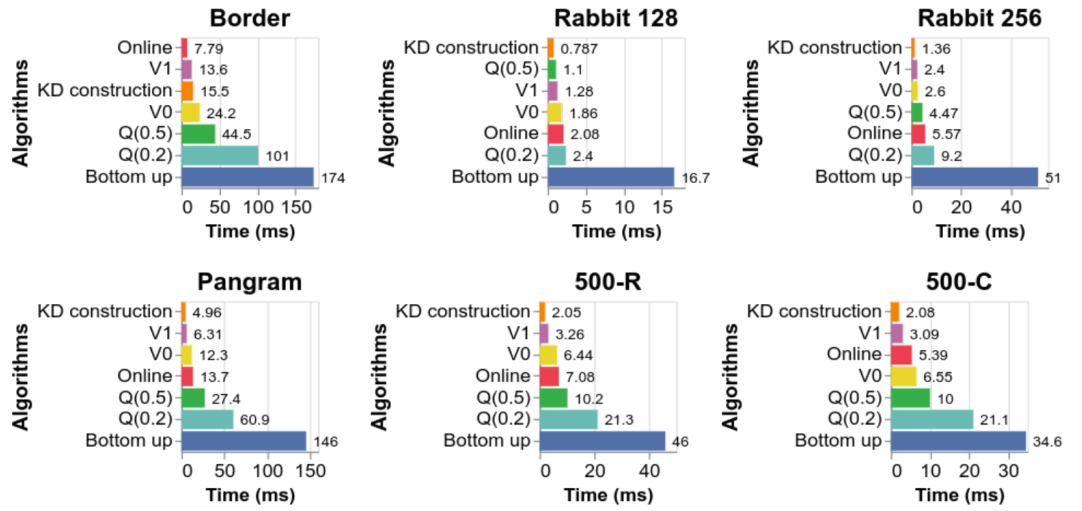


Figura 3.14: Algoritmos de construção de *ball tree* por *run time* (sem otimização *EL*).

(veja a Figura 3.14). O algoritmo Bottom up é significativamente mais lento do que todos os outros algoritmos. Por exemplo, para o conjunto de dados Rabbit 256, ele é executado aproximadamente 37 vezes mais lento que KD (o algoritmo mais rápido) e 5 vezes mais lento que Q(0.2), que por sua vez, é o segundo algoritmo mais lento. Observe também que diminuir o valor α de 0.5 para 0.2 afeta severamente o tempo de execução de Q, mais do que dobrando essa medida para todos os conjuntos de dados.

Os tempos de execução mostrados na Fig. 3.14 excluem a etapa de otimização *EL*, que, embora quadrática, é bastante rápida para os conjuntos de dados testados (ver Tabela 3.2). De fato, mesmo quando emparelhado com o algoritmo de construção mais rápido (KD), não leva mais do que 12% do tempo total, mesmo para o maior conjunto de dados (Border). Claramente, quando executado após um algoritmo lento, seu impacto no tempo total de execução torna-se insignificante.

Tabela 3.2: Tempo de otimização *EL* (ms)

	Bottom up	KD	Online	Q(0.2)	Q(0.5)	V0	V1
500-C	0.69	0.62	0.71	0.67	0.68	0.69	0.68
500-R	0.72	0.66	0.69	0.65	0.63	0.67	0.69
Border	1.74	1.92	2.22	1.69	1.59	1.62	1.61
Pangram	1.47	1.36	1.60	1.35	1.32	1.48	1.37
Rabbit 128	0.24	0.22	0.26	0.22	0.19	0.21	0.19
Rabbit 256	0.47	0.39	0.53	0.46	0.44	0.45	0.46

Com relação à área total, confirmamos que o algoritmo Bottom-up produz árvores com a menor área total para a maioria dos conjuntos de dados (ver Fig. 3.15). Essa vantagem é maior no caso do conjunto de dados 500-C, onde KD produz uma ár-

vore com área total mais de 2.2 vezes maior que a produzida pelo algoritmo Bottom-up. Também notamos que os algoritmos de construção Q, V0 e V1 se saem muito bem para todos os conjuntos de dados. Por exemplo, o pior resultado obtido com o algoritmo V0 foi para o conjunto de dados Pangram, com uma área apenas 20% maior que a do Bottom up, enquanto para todos os outros conjuntos de dados, seu resultado foi no máximo de 2% a 6% maior.

Ainda para o Pangram, o melhor resultado foi obtido com o Q(0.2) com área 6% menor que a do Bottom up. Observamos ainda que a redução de α de 0.5 para 0.2 produz árvores com áreas menores, mas o ganho é bastante modesto em geral.

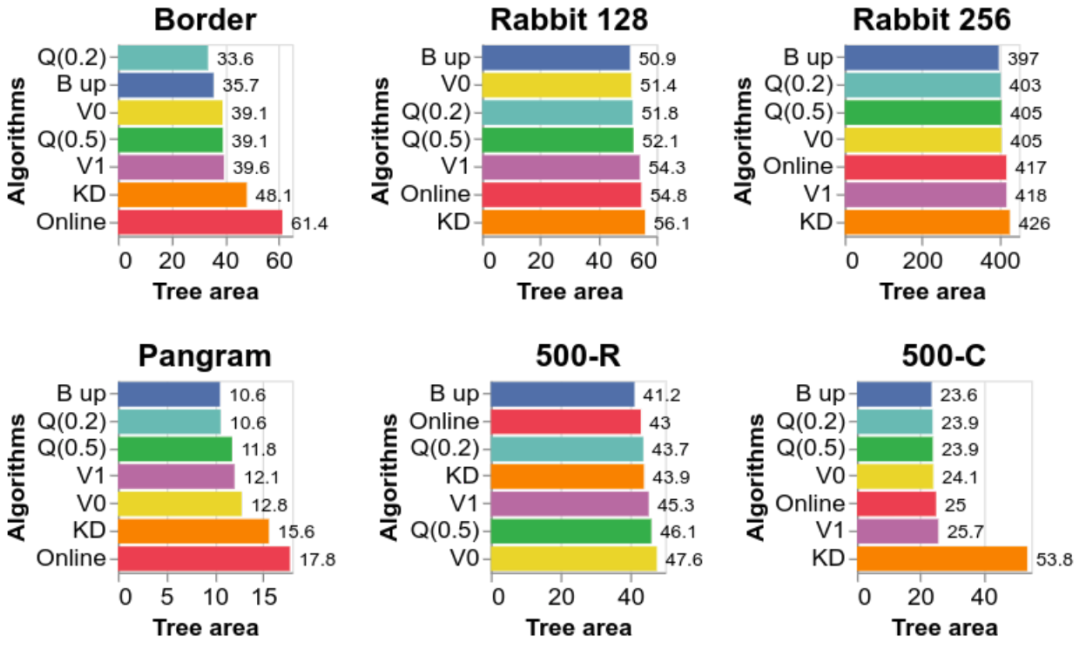


Figura 3.15: Algoritmos de construção de *ball tree* por área total (com otimização EL)

O efeito da otimização *EL* na área total da árvore também depende do tipo de conjunto de dados. Conjuntos de dados com bolas grandes, como os Rabbits, são os menos beneficiados. Por exemplo, enquanto a árvore Rabbit 128 construída com o algoritmo Bottom-up teve sua área melhorada em 4% (de 53 para 50.9), a árvore para o conjunto de dados Border construído com o algoritmo Q(0.2) melhorou mais de 61% (de 54.4 a 33.6).

3.3.4 Funções de ranqueamento da fila de prioridades

Conduzimos todos os testes de consulta com quatro variantes de f_P, conforme discutido na Seção 3.2.1. Estas variantes correspondem a utilizar apenas a distância mínima, ou utilizar as distâncias mínimas como critério principal, mas desvinculando-se da distância máxima. As distâncias mínima e máxima, por sua vez, podem ser es-

timadas a partir da distância entre os próprios nós, ou pela menor das distâncias entre seus filhos.

A Figura 3.16 mostra um gráfico para consultas TD entre um par de árvores Pangram construídas com o algoritmo Bottom up, onde o número de operações de fila é mostrado em função da distância calculada entre as duas árvores para cada uma de quatro variantes f_P. Observe que, em geral, usar os filhos para estimar distâncias resulta em menos operações de fila necessárias. Além disso, usar distâncias mínimas como chave primária e distâncias máximas como chave secundária faz com que as consultas a distância zero exijam muito menos operações. Como essas descobertas são repetidas para todas as árvores e todas as consultas, nossa discussão a partir de agora considerará apenas os resultados calculados com esta última variante f_P, denominada "*children min+max distance*".

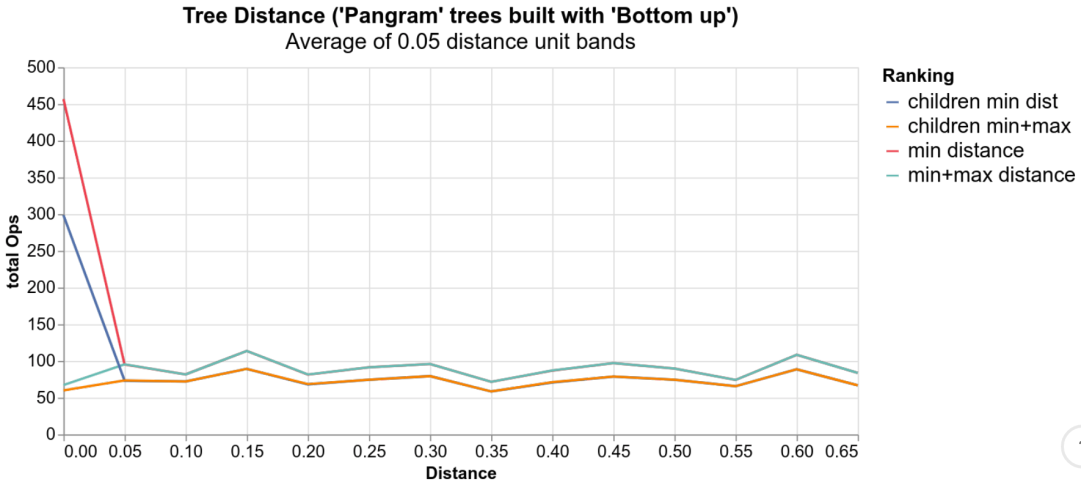


Figura 3.16: Tree distance (TD) – total Ops por distância na faixa *close*. *Ball trees* Pangram construídas com algoritmo Bottom up.

3.3.5 Desempenho da consulta

Nós nos concentramos em seguida no desempenho da consulta TD para os vários conjuntos de dados e algoritmos de construção. Um primeiro teste foi realizado para avaliar o efeito da otimização *EL* no desempenho geral desta consulta.

Conforme mostrado na Figura 3.17, a otimização é claramente benéfica para árvores construídas com todos os algoritmos de construção para o conjunto de dados Rabbit 256, mas essa tendência é verificada para todos os conjuntos de dados testados, como por exemplo 500R (fig. 3.18). Nestes gráficos, "Mean Ops" é o número médio de operações para consulta TD em todas as faixas de distância.

A Tabela 3.3 mostra o valor médio do número total de operações na fila de prioridades, necessárias para responder a essa consulta, considerando sempre um par de cada conjunto de dados.

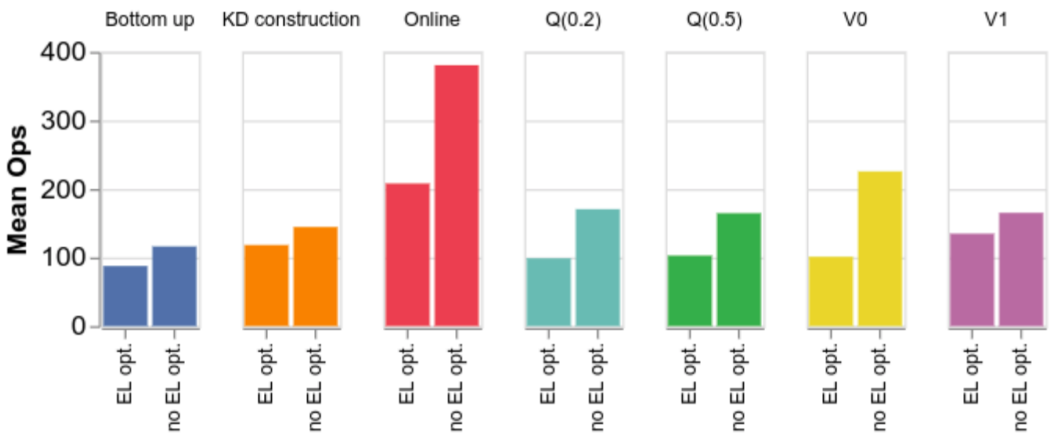


Figura 3.17: Efeito da otimização *EL* na consulta Tree Distance (*TD*) sobre o conjunto de dados *Rabbit 256*.

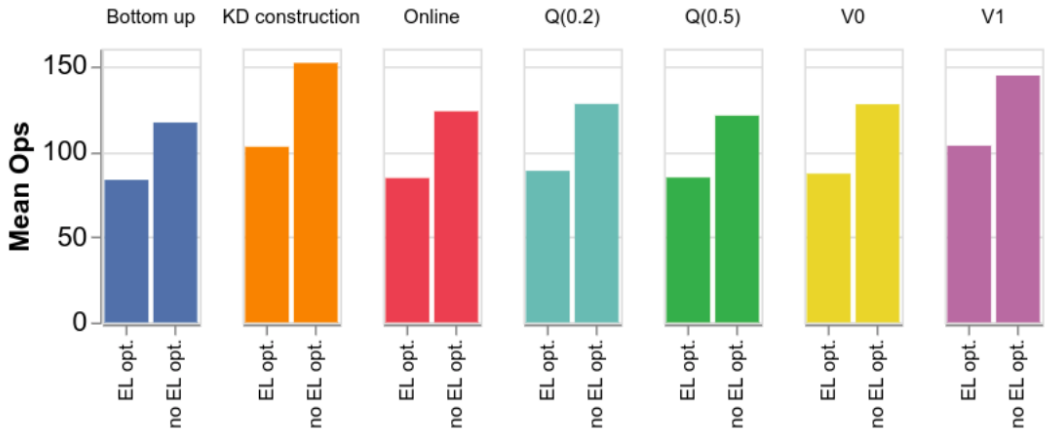


Figura 3.18: Efeito da otimização *EL* na consulta Tree Distance (*TD*) sobre o conjunto de dados *500R*.

Novamente observamos que as árvores construídas com o algoritmo Bottom up se comportam melhor, seguidas de perto por alguns dos outros algoritmos, notadamente Q. O pior do lote parece ser o algoritmo Online.

Notamos que diminuir o valor de α para o algoritmo Q não melhora os resultados em todos os casos, com Q(0.2) e Q(0.5) tendo desempenho semelhante.

Tabela 3.3: Número médio de operações de fila para consulta TD

	B up	KD	Online	Q(0.2)	Q(0.5)	V0	V1
500-C	**66.79**	218.36	70.48	69.34	68.10	72.47	72.69
500-R	**83.38**	102.71	84.45	88.67	84.76	87.04	103.33
Border	**149.17**	195.20	822.74	160.32	167.02	225.43	214.95
Pangram	**68.02**	93.26	170.46	68.80	72.97	91.00	81.08
Rabbit 128	**65.55**	78.22	138.18	77.90	68.35	66.10	82.68
Rabbit 256	**87.68**	118.35	208.16	98.94	102.95	101.25	134.95

Quanto às consultas DtP, o algoritmo Online surpreendentemente se sai muito bem, superando o algoritmo Bottom up para os conjuntos de dados 500-C e 500-R, conforme indicado na Tabela 3.4, que mostra o número médio de operações de fila para todas as combinações de algoritmo de construção/conjunto de dados.

Outro resultado inesperado é a pequena diferença nos tempos de consulta observados para as árvores obtidas com o algoritmo Q quando α é reduzido de 0.5 para 0.2.

Claramente, então, $\alpha = 0.5$ é indicado na maioria dos casos, pois permite custos de construção significativamente inferiores a $\alpha = 0.2$. De forma complementar, a Figura 3.19 ilustra o número de operações para a consulta DtP com o conjunto de dados Rabbit 256. Note que V0 e V1 tem bom desempenho frente aos demais. Para esta consulta, observamos ainda que o algoritmo Q(0.5) é mais eficiente que Q(0.2).

Tabela 3.4: Número médio de operações de fila para consulta DtP

	B up	KD	Online	Q(0.2)	Q(0.5)	V0	V1
500-C	29.31	37.28	**29.07**	30.28	29.91	30.31	31.75
500-R	30.11	33.27	**29.44**	32.12	30.55	30.85	33.63
Border	**48.21**	52.13	67.13	48.93	48.56	51.37	50.45
Pangram	**33.07**	37.23	44.57	33.47	34.57	35.69	34.47
Rab. 128	32.98	**31.35**	36.30	35.42	32.28	31.65	32.12
Rab. 256	39.58	38.64	45.15	42.64	38.95	37.01	**35.22**

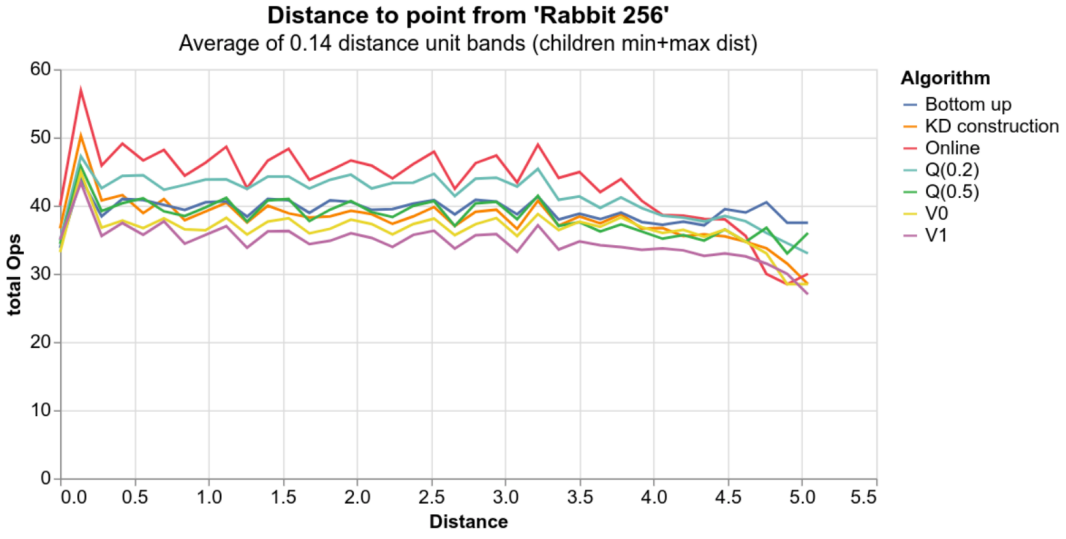


Figura 3.19: Consulta ***DtP*** sobre Rabbit256.

Experimentos com consultas TI (*tree intersection*) para distâncias *close* revelam que as árvores construídas com a abordagem Bottom up levam aos melhores resultados em quase todos os casos, exceto para o conjunto de dados Border, onde Q(0.2) produz um resultado ligeiramente melhor (fig. 3.20).

Tanto KD quanto a abordagem Online produzem os piores resultados em todos os conjuntos de dados, exigindo em alguns casos mais que o dobro do número de operações. Os algoritmos Q, V0 e V1, propostos aqui, se saem relativamente bem. Por exemplo, as árvores construídas com Q(0.5) requerem entre 4% e 27% mais operações do que aquelas construídas com a abordagem Bottom up.

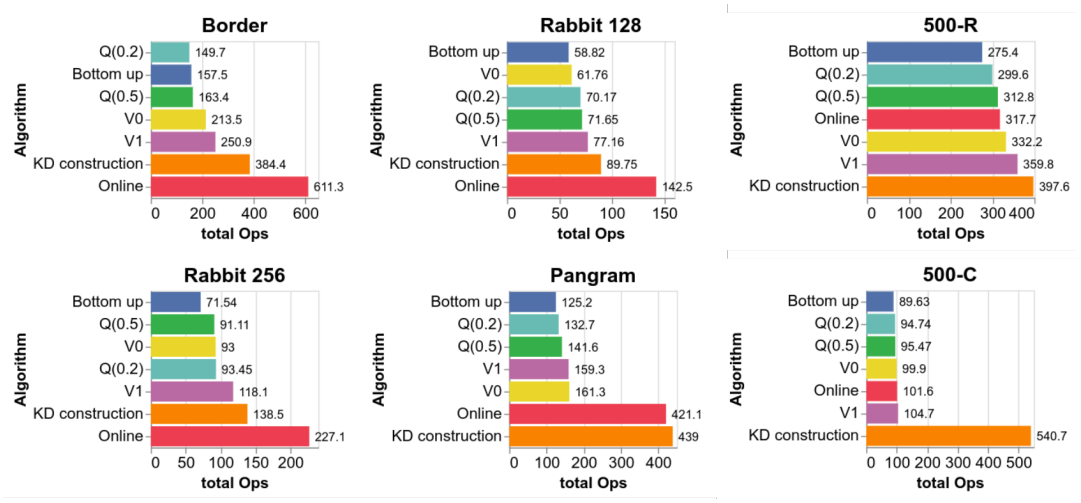


Figura 3.20: Consulta ***TI*** entre todos os conjuntos de dados.

Focamos agora em experimentos com consultas PtI (*point intersection*). Na Figura 3.21, observamos que para distâncias *close* os algoritmos se comportam de forma semelhante, principalmente se observarmos os resultados obtidos para os 5 algoritmos de melhor desempenho.

Por exemplo, para a árvore Rabbit 256, V1 requer apenas 31% mais operações do que o melhor resultado obtido pela árvore Bottom up. No caso do conjunto de dados 500-C, observamos que o resultado para KD é significativamente pior do que o obtido pelos outros algoritmos. Isso pode ser atribuído ao fato de que (1) nossa amostragem de consulta não detectou nenhuma interseção com ponto, e (2) KD é o único algoritmo de construção top-down, o que leva a muitas bolas grandes nos níveis superiores da árvore.

As consultas do tipo LnI (*Line Intersection*) obtiveram resultados bem próximos para todos os conjuntos de dados entre os diversos algoritmos de construção de *ball tree*. Esta consulta pode ser útil, por exemplo, em operações que utilizam comandos realizados por traços à mão livre que podem interceptar objetos.

A Figura 3.22 mostra esta consulta na faixa de distância *close*. Na maioria dos casos os algoritmos obtiveram bom desempenho, exceto KD e Online. Os algoritmos Q, V0 e V1 superaram o Bottom up para os conjuntos de dados Border, Rabbit 128 e Rabbit 256.

A consulta DtL (*Distance to Line*) é útil para diversas operações comuns, como alinhamento de objetos e prevenção de colisão com a borda da tela, por exemplo. So-

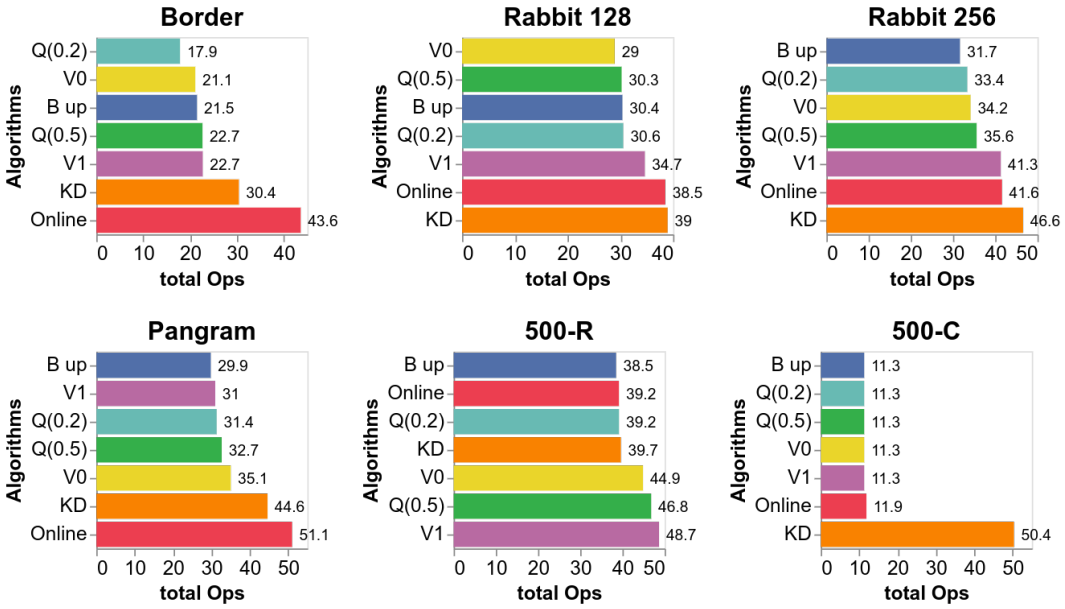


Figura 3.21: Consulta *Point intersection (PtI)* - Algoritmos de construção de *ball tree* por total Ops (filtrado por faixa de distância *Close*).

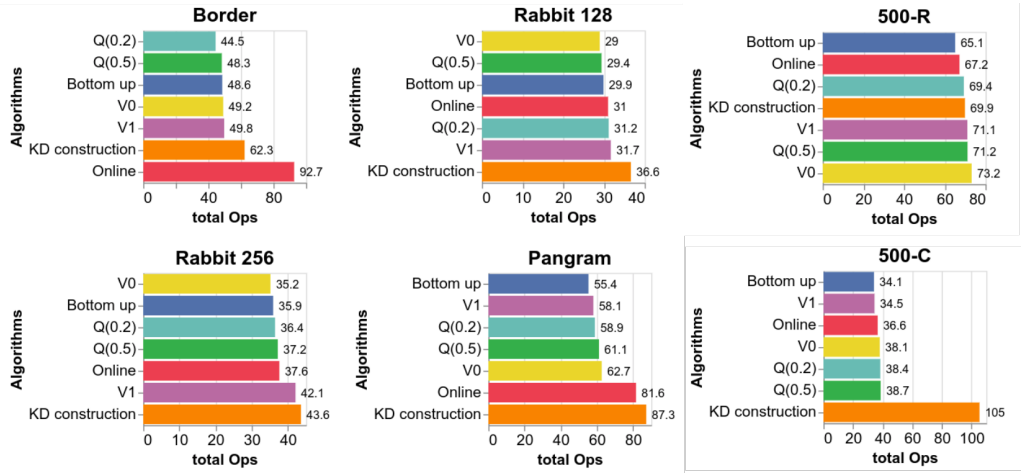


Figura 3.22: Consulta *Line Intersection (LnI)* - Algoritmos de construção de *ball tree* por total Ops (filtrado por faixa de distância *Close*).

bre este tipo de consulta podemos observar que os resultados foram muito variados, dependendo do conjunto de dados.

Na maior parte dos experimentos, os algoritmos V0, V1, Q(0.2) e Q(0.5) foram os mais eficientes. As Figuras 3.23 e 3.24 ilustram estes bons resultados para dois casos: 500C e Border, respectivamente.

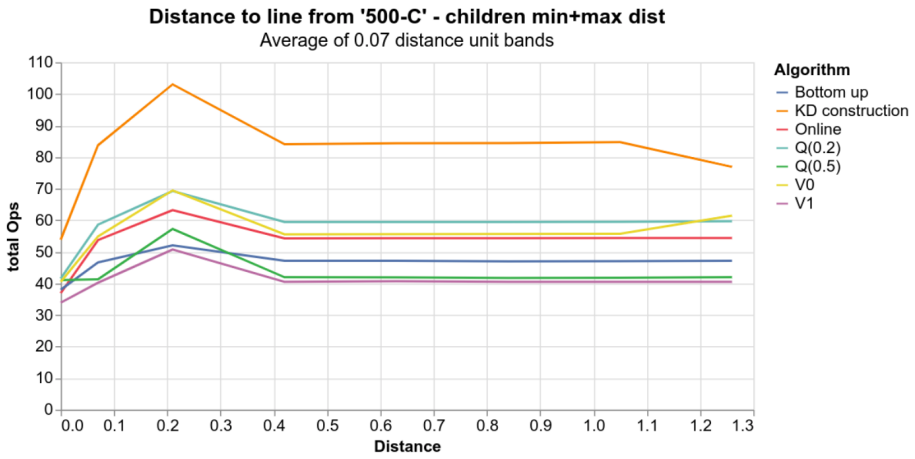


Figura 3.23: Consulta *Distance to Line (DtL)* para 500C - Número de operações por distância (filtrado por faixa de distância *Close*).

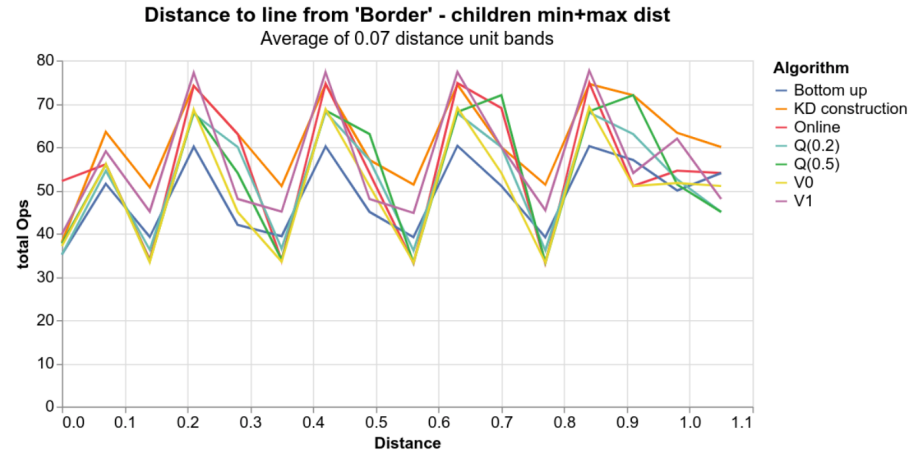


Figura 3.24: Consulta *Distance to Line (DtL)* para Border - Número de operações por distância (filtrado por faixa de distância *Close*).

Capítulo 4

Aproximação de formas com bolas

Como destacado no Capítulo 2, podemos aproximar uma determinada forma usando eficientemente uma coleção de bolas (círculos). Na Seção 2.2, foi demonstrado que o algoritmo RDMA introduzido em [46] é um algoritmo eficiente para produzir uma coleção de bolas aproximando uma dada forma rasterizada como uma imagem binária. Apesar da abordagem RDMA ser muito eficiente e produzir boas aproximações, ela pode perder algumas características finas na imagem. Outra deficiência é que ela não pode ser ajustada para produzir aproximações mais finas ou grosseiras. Neste capítulo vamos apresentar uma solução para estes problemas.

Antes porém, para justificar a escolha por coleções de bolas, bem como analisar a eficiência de consultas realizadas sobre estas coleções, investigamos aproximação de formas bidimensionais usando uma coleção de retângulos com o objetivo de responder a duas questões importantes:

- Quão bem podemos aproximar uma determinada forma usando círculos ou retângulos?
- Quando essas coleções de círculos ou retângulos são organizadas em um BVH, com que eficiência cada hierarquia responde a consultas de proximidade?

Para determinar a fidelidade das aproximações com bolas/retângulos à forma original, desenvolvemos o seguinte procedimento:

1. A mesma forma é rasterizada novamente em uma imagem maior. O objetivo de usar uma imagem maior é compensar o efeito de *aliasing* introduzido pelo processo de rasterização. Ela serve como referência para determinar o erro da aproximação;
2. a coleção de bolas/retângulos também é rasterizada em uma imagem binária aumentada (do mesmo tamanho da referência);
3. pixels diferentes entre as duas imagens aumentadas são contados.

A Figura 4.1 ilustra o procedimento acima, aplicado para uma coleção de bolas, mas o processo é semelhante para coleção de retângulos.

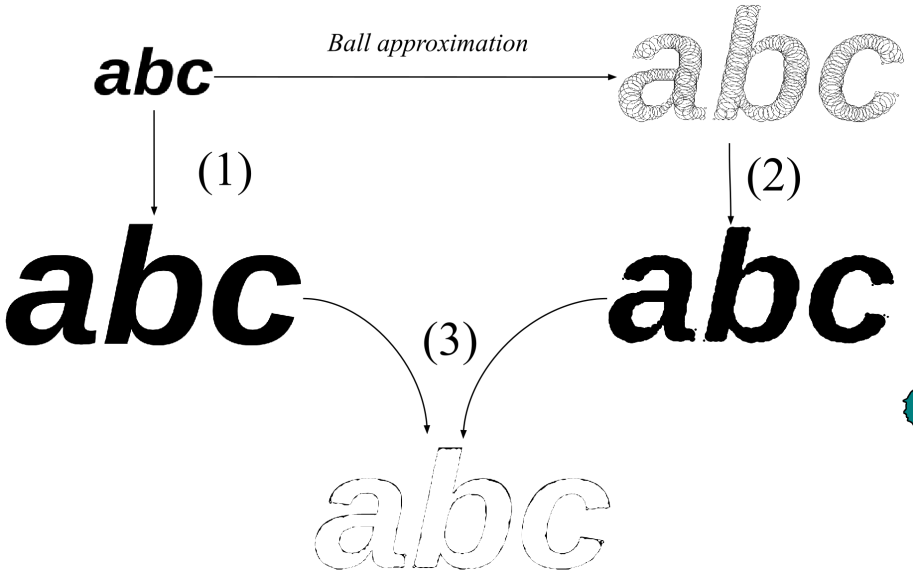


Figura 4.1: Procedimento de determinação da fidelidade da aproximação por um conjunto de bolas.

4.1 RA trees

Para a construção de coleções de retângulos aproximando formas, implementamos o que chamamos de *Rectangle Approximation (RA) trees* [33], que além de prover um meio de aproximar a forma, também podem ser usadas para responder consultas de proximidade.

Claramente, há uma relação entre o número de retângulos usados para aproximar a forma e a precisão da resposta a uma consulta de proximidade. Para algumas aplicações, no entanto, um pequeno erro na resposta é tolerável se esta for calculada rapidamente.

A *RA tree* é uma adaptação do esquema de subdivisão *KD tree* para imagens em preto e branco. Cada nó associa um subconjunto de pixels com uma caixa delimitadora alinhada aos eixos *(axis-aligned bounding box - AABB)*.

Assumindo que um pixel é um quadrado unitário, se a caixa de um nó com lados w e h contém exatamente $w \times h$ pixels, então ele é um nó folha. Caso contrário, a caixa é dividida no meio em relação ao maior lado, e os pontos que caem dentro de cada caixa resultante são processados recursivamente para construir as subárvores esquerda e direita.

Assim, uma aproximação de forma pode ser obtida primeiro convertendo-a em uma imagem binária, criando uma *RA tree* para os pixels e, em seguida, coletando as

caixas delimitadoras dos nós folha. Uma aproximação mais grosseira usando menos caixas pode ser obtida usando duas heurísticas:

- Criar um corte, ou seja, truncar a árvore abaixo de um determinado nível;
- Usar uma taxa de preenchimento de nó $F < 1$. Isso significa que um nó interno com área A contendo n pixels é promovido a um nó folha se $FA >= n$, ou seja, a árvore é simplificada para ignorar as divisões de nós aproximadamente completos.

A Figura 4.2 mostra a aproximação da forma tipográfica "abc" considerando a heurística de corte para um determinado nível da *RA tree* correspondente.

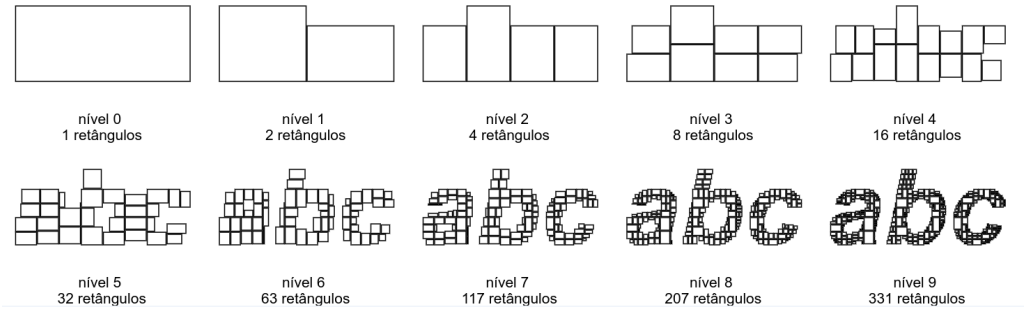


Figura 4.2: Ilustração da heurística de corte da *RA tree*. Os números indicam o número total de nós do corte em cada nível.

Para determinar a fidelidade da aproximação com a *RA tree* à forma original, utilizamos o procedimento descrito anteriormente. A Figura 4.3 ilustra um exemplo desta análise sobre a forma tipográfica "&#$". Neste caso, a imagem da forma original tem resolução de 187×102 contendo 6.909 pixels pretos (fig. 4.3.a). A imagem de referência é 5 vezes maior (fig. 4.3.b) com 171.981 pixels. A proximação é feita com 810 retângulos (fig. 4.3.c) cuja contagem de pixels diferentes é igual a 10.338 (fig. 4.3.d), ou seja, um erro de aproximação de 6%.

(a) (b) (c) (d)

Figura 4.3: Ilustração do procedimento de fidelidade da aproximação por um conjunto de retângulos. A *RA tree* foi cortada no nível 14.

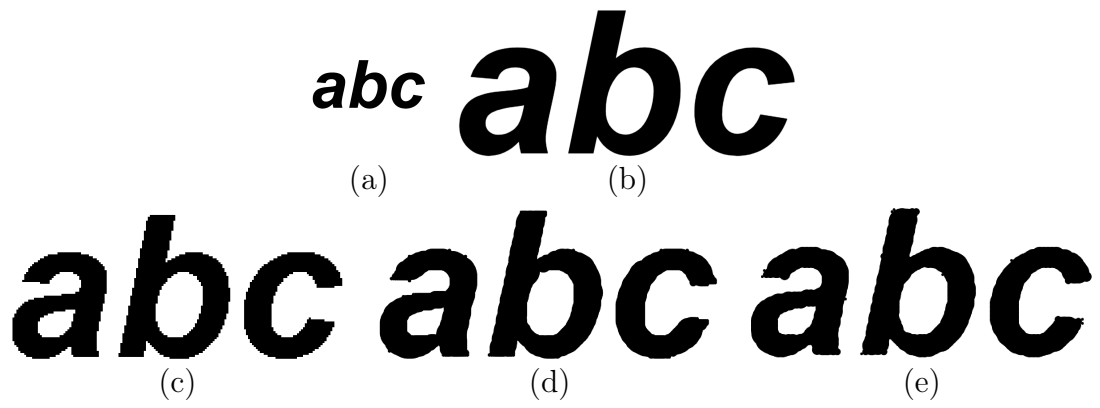
4.2 A fidelidade da aproximação com bolas

4.2.1 Algoritmo *SDT+filter*

Analogamente, passemos agora a determinar a fidelidade da aproximação com bolas à forma original a partir da abordagem RDMA.

Considere o exemplo onde a forma renderizada mostrada na Figura 4.4.a tem 5082 pixels, enquanto a imagem em escala 5× maiores (Fig. 4.4.b) usada como referência tem 127048 pixels. Por outro lado, o conjunto de bolas obtido com a abordagem RDMA possui 456 círculos que, quando renderizados na mesma escala, (Fig 4.4.d), produzem 6774 pixels diferentes, ou 5.3% do total. Em comparação, quando os pixels da imagem de amostra são renderizados 5× maior (Fig. 4.4.c) eles produzem 6438 pixels diferentes, ou 5.1% do total.

Para obter melhores aproximações usando menos bolas, propomos um novo algoritmo que, como a abordagem RDMA, parte dos círculos da transformada de distância ao quadrado (SDT). Em vez de computar um eixo medial, no entanto, esses círculos são primeiro classificados por tamanho em ordem decrescente e testados quanto à contenção aproximada em qualquer um dos círculos considerados anteriormente. Os círculos que passam no teste são descartados, enquanto os que falham são adicionados à coleção final. Um círculo c_1 com raio r_1 é considerado aproximadamente contido em um círculo c_2 com raio r_2 se estiver totalmente contido em um círculo com o mesmo centro que c_2, mas com raio $r_2 + \epsilon$, onde $\epsilon \geq 0$ é uma margem de erro.



(a) (b) (c) (d) (e)

Figura 4.4: Aproximação por conjunto de bolas: (a) imagem com forma renderizada a ser aproximada; (b) renderização 5× maior (referência); (c) imagem original com pixels 5× maiores; (d) renderização da coleção de bolas obtida com o algoritmo RDMA; (e) renderização da coleção de bolas obtida com o algoritmo SDT+filter e $\epsilon = 1$.

Esse novo algoritmo, que chamamos de *SDT+filter*, é executado mais lentamente que a abordagem RDMA, pois cada elemento do SDT deve ser testado em relação a todos os outros. Felizmente, é possível usar uma *ball tree online* (veja Seção 3.1.2)

para acelerar a busca por bolas envolventes, ou seja, apenas bolas que intersectam a bola de consulta expandida por ϵ devem ser testadas para contenção parcial. As bolas que falharem no teste são adicionadas à *ball tree* e, assim, suas folhas correspondem à coleção final. A Figura 4.4.e mostra o resultado do algoritmo para $\epsilon = 1$, contendo 266 bolas e gerando 5600 pixels diferentes, ou 4.4% do total. Notamos que este resultado contém menos bolas e produz um erro menor do que o obtido com a abordagem RDMA (Fig. 4.4.d), embora não seja tão visualmente agradável, pois usa cerca de $\frac{1}{3}$ da quantidade de bolas. De fato, o número de pixels diferentes é ainda menor do que o obtido para a imagem com pixels com escala superdimensionada (Fig. 4.4.c).

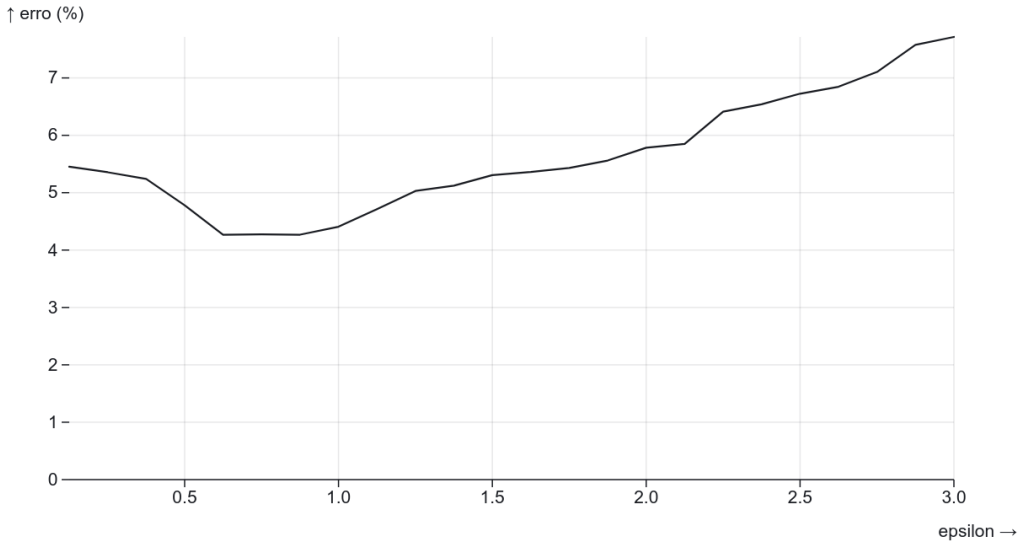


Figura 4.5: Erro (número de pixels diferentes em relação à Fig. 4.4.b) para a abordagem de SDT+filter para valores variados de ϵ.

Curiosamente, quando testada com várias imagens obtidas a partir de formas tipográficas, a abordagem SDT+filter alcança boas aproximações dentro de uma ampla faixa de valores ϵ [31]. Para o exemplo na Figura 4.4.a, $0.4 \leq \epsilon \leq 1.2$ produz erros abaixo de 5% (veja fig. 4.5).

4.3 Comparação entre aproximação de formas com *RA trees* e *ball trees*

Tentamos responder à pergunta sobre fidelidade de aproximação testando formas geradas a partir de texto renderizado com fontes e tamanhos personalizáveis.

O processo de comparação é realizado a partir destas fontes selecionadas conforme os parâmetros oferecidos no protótipo da aplicação (vide Figura 4.6).

Uma imagem binária (preto e branco) é então produzida e algoritmos para aproximação usando bolas e retângulos são aplicados. O erro de aproximação é avaliado

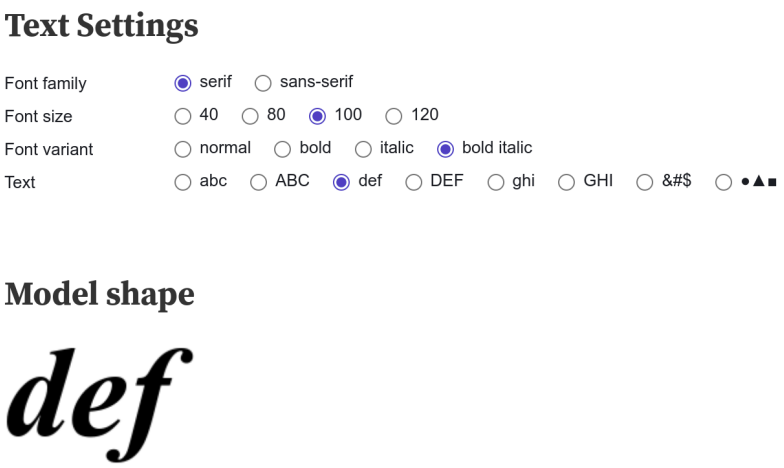


Figura 4.6: Interface da aplicação de comparação de aproximações de forma.

usando uma versão em escala da forma como referência. Renderizações em escala similar da coleção de bolas e da coleção de retângulos são então produzidas. O erro é medido contando os diferentes pixels.

A Figura 4.7 mostra os parâmetros de ajuste para os algoritmos de produção de coleções de bolas e a Figura 4.8 para coleções de retângulos.

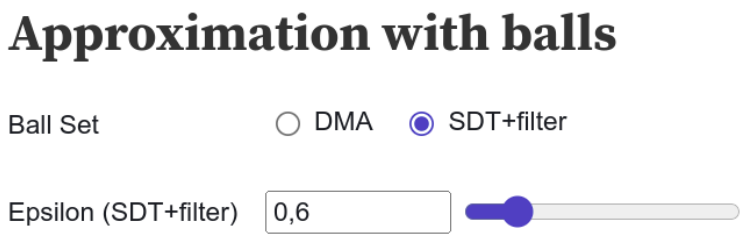


Figura 4.7: Seleção dos algoritmos de aproximação com bolas e margem de erro ϵ.

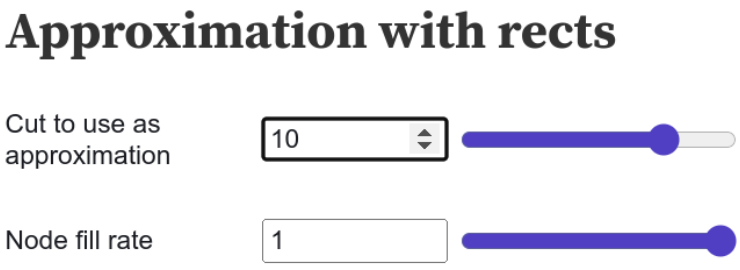
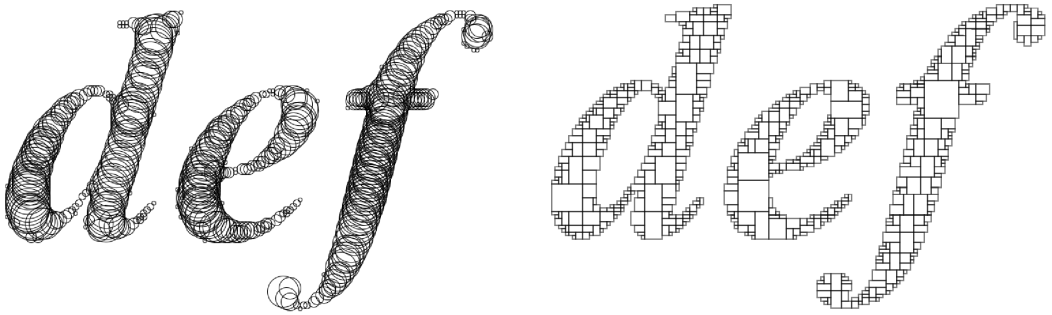


Figura 4.8: Interface de seleção dos parâmetros de ajuste para algoritmo de aproximação com retângulos.

Para as coleções de bolas e retângulos produzidas, é construída uma *ball tree/RA tree* correspondente. Avaliamos a questão da eficiência da consulta de proximidade contando o número de etapas necessárias pela *ball tree* e *RA tree* para responder a consultas *Distance to Point (DtP)*. Mais precisamente, amostramos uma região ao

redor da forma que é três vezes maior, tanto no eixo x quanto em y com $30 \times 30 = 900$ pontos. O número de iterações é o número de nós visitados por um algoritmo *branch-and-bound* recursivo simples implementado para ambos os tipos de árvore.

Diversos experimentos foram realizados. Por exemplo, tomamos a forma tipográfica "def" e aplicamos a aproximação de forma com SDT+filter e $\epsilon = 0.6$; e a aproximação com retângulos com corte no nível 10 da árvore. Estes parâmetros se justificam para manter o número igual de primitivas que neste caso foi de 481, tanto de bolas quanto retângulos. O detalhe da Figura 4.9 exibe ambas aproximações.



(a) (b)

Figura 4.9: Aproximações da forma tipográfica "def" com 481 primitivas. (a) com bolas ($\epsilon = 0.6$), (b) com retângulos (corte no nível 10 da *RA tree*).

Para este experimento específico, podemos ver na Figura 4.10 que o erro de aproximação por número de primitivas é bem mais baixo na aproximação com SDT+filter em relação a *RA tree*. Mais especificamente, 5707 pixels diferentes (6.3%) para SDT+filter contra 7749 (8.6%) para *RA tree*. Este é um comportamento geral. Exceções a esta regra podem ser observadas em formas contendo muitas arestas estritamente horizontais ou verticais. Nesses casos, é possível usar relativamente poucos retângulos com baixas taxas de erro, mas são necessárias muitas bolas para reproduzir fielmente as mesmas arestas.

A Figura 4.11 mostra que é possível reduzir ainda mais o número de bolas obtidas com a abordagem SDT+filter aumentando o valor de epsilon (tolerância de aproximação). Embora o número de retângulos também possa ser reduzido usando cortes com níveis mais rasos e/ou modificando os limites da taxa de preenchimento (consulte a Seção 4.1, vide fig 4.12), o erro de aproximação resultante aumenta muito mais acentuadamente. Este fato acontece de maneira menos acentuada com bolas.

Ao considerar o desempenho da consulta *DtP* (fig. 4.13), as *ball trees* geralmente têm um desempenho pior para pontos próximos do que retângulos.

Essa tendência se inverte no caso de pontos distantes. Isso pode ser atribuído ao fato de que as *RA trees* se comportam como hierarquias de particionamento de espaço, de modo que o tamanho das folhas tende a diminuir logaritmicamente com o nível. Por outro lado, o conjunto de bolas calculado pela abordagem SDT+filter

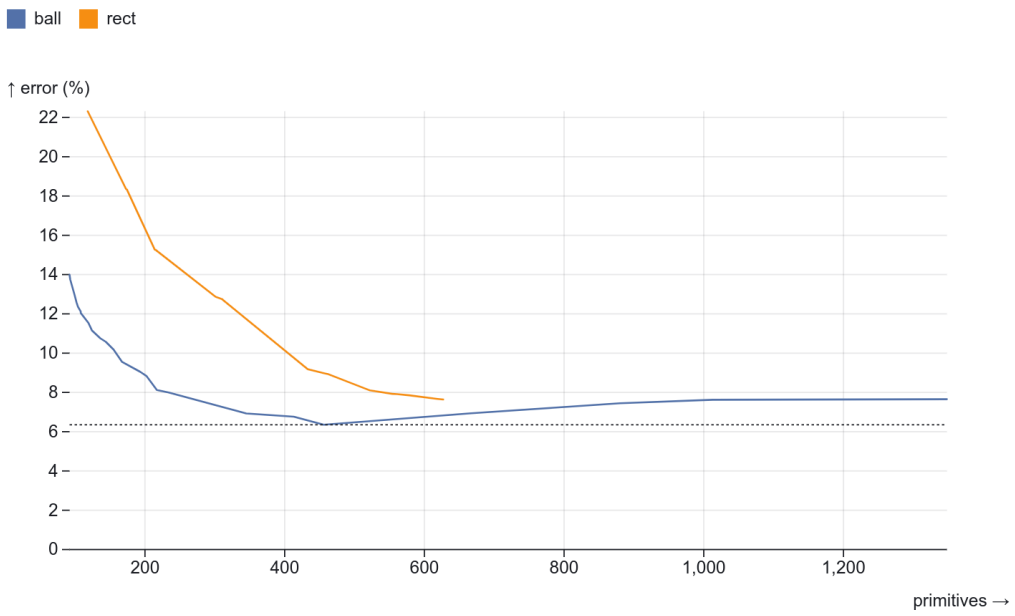


Figura 4.10: Erro de aproximação por quantidade de primitivas.

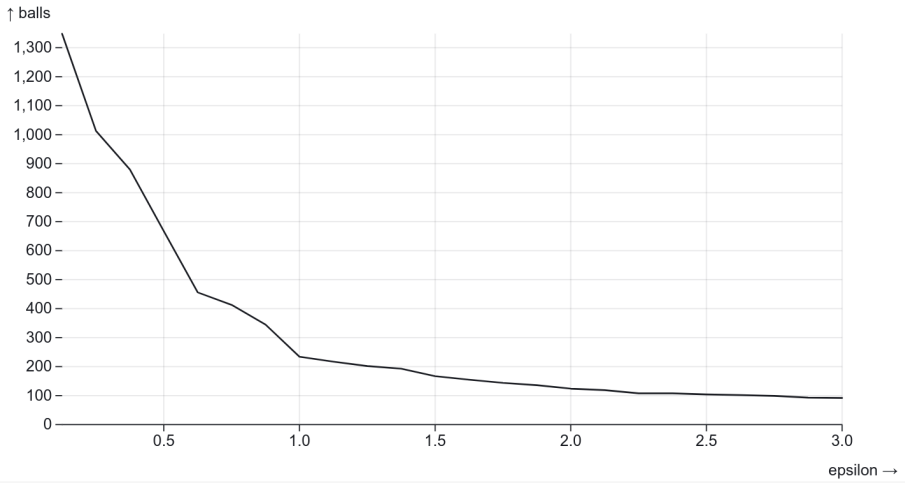


Figura 4.11: Quantidade de bolas por tolerância de aproximação (epsilon).

tende a ser relativamente grande em média e assim a busca deve examinar folhas em níveis mais profundos para pontos próximos à forma. Como um todo, exceto para os casos mencionados acima, onde as formas têm longas bordas horizontais ou verticais, a contagem mais baixa de primitivas nas aproximações com bolas parece favorecer *ball trees* em vez de *RA trees*.

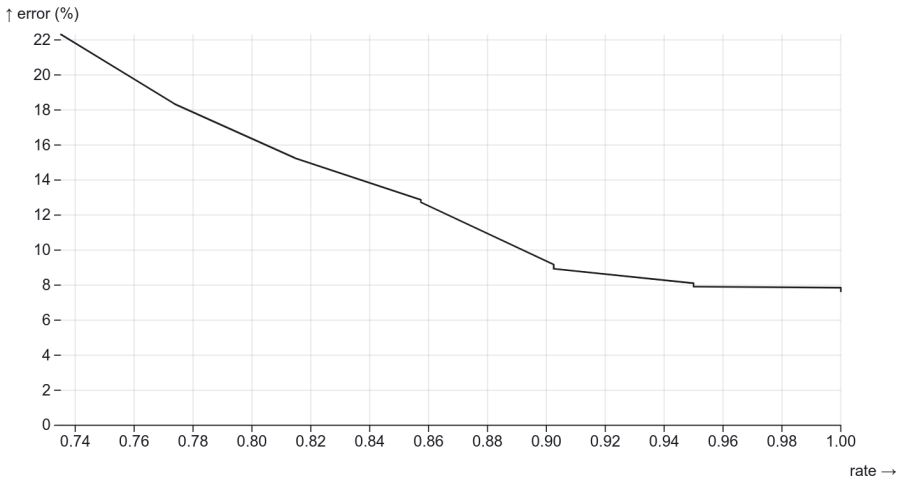


Figura 4.12: Erro de aproximação de retângulos por tacha de preenchimento.

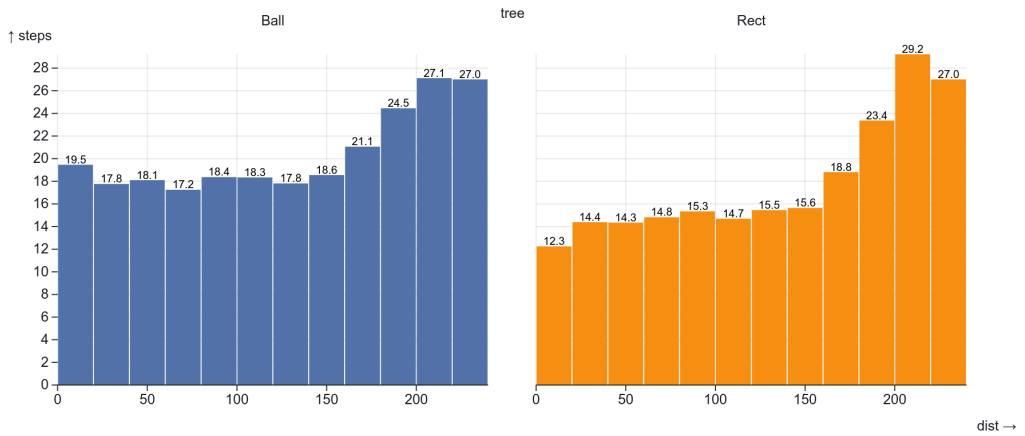


Figura 4.13: Média do número de iterações na consulta *DtP* pela distância.

Capítulo 5

Layout de formas com *ball trees*

Na Seção 2.3 mencionamos brevemente como usar *ball trees* para layout de formas. Aqui nós desenvolvemos essas ideias detalhando um conjunto de operações de layout e algoritmos para computá-las com a ajuda de *ball trees*. Essas operações foram implementadas em uma aplicação web de prova de conceito [74, 75], para que o leitor interessado possa interagir diretamente com ela.

5.1 Preparação dos Patches

Em nosso protótipo, as formas são fragmentos de imagem (*patches*) obtidos pela digitalização e segmentação de documentos. Empregamos um algoritmo simples de segmentação de imagens que considera todos os pixels suficientemente semelhantes a uma determinada cor como plano de fundo. *Patches* são então formados pelo reconhecimento de grupos de pixels de primeiro plano através da vizinhança 4-conexa. Para cada *patch*, um conjunto de bolas é criado usando o algoritmo descrito na Seção 2.2, e estas são organizadas em ball trees usando o algoritmo Q, com $\alpha = 0.5$ e otimização *EL*. A Figura 5.1 mostra um exemplo da estrutura obtida a partir da imagem Border (consulte a tab. 3.1).

5.2 Agrupamento baseado em distância

Por construção, todos os *patches* obtidos de uma imagem não se sobrepõem. No entanto, ao considerar os *patches* a uma distância $d > 0$ um do outro, podemos inferir agrupamentos naturais decorrentes do layout original do documento. Em outras palavras, podemos descobrir que a distância entre as letras de uma palavra é menor que a distância entre palavras, que por sua vez é menor que a distância entre linhas, etc. Isso sugere que agrupamentos *ad hoc* podem ser construídos especificando um limite de distância.

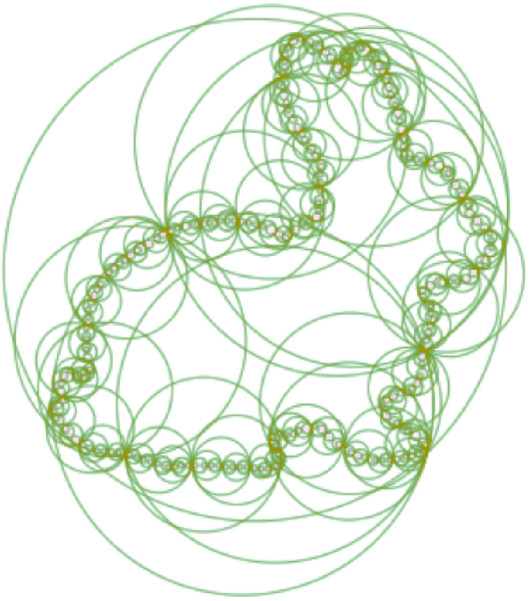
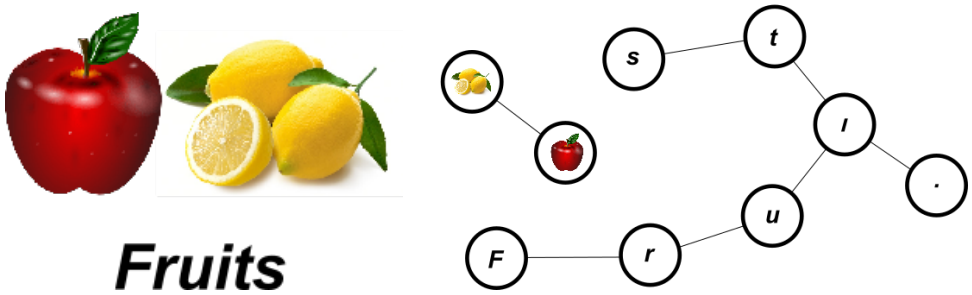


Figura 5.1: *Patch* para imagem Border. *Ball tree* formada a partir do algoritmo Q e aproximação da forma com SDT+filter.

Mais formalmente, se P for um conjunto de *patches*, gostaríamos de particionar P em um conjunto de clusters C em relação ao predicado $dist(p_a, p_b) < d$, onde p_a e p_b são *patches* em P. Assim, se criarmos um grafo G onde os *patches* são vértices e uma aresta é definida entre dois *patches* p_a e p_b sempre que $dist(p_a, p_b) < d$, então C corresponde aos componentes conectados de G (veja a fig. 5.2).



(a) (b)

Figura 5.2: Ilustração de agrupamento automático baseado em uma distância d. (a) o conjunto de *patches* P. (b) O grafo G e os dois clusters C representados pelos componentes conectados.

Em nosso protótipo, a implementação da operação de agrupamento é uma aplicação direta do algoritmo *union-find* para G. Embora o conjunto de arestas candidatas seja quadrático, testar o predicado de distância é trivial apenas para um subconjunto relativamente pequeno, uma vez que as *ball trees* são capazes de descartar arestas associadas a pares de *patches* distantes em tempo constante. Uma vez que os fragmentos pertencentes a cada cluster são determinados, uma única *ball tree*

para cada cluster é construída a partir do conjunto de bolas formado pelo nó raiz de cada *ball tree* associada a cada *patch*. A Figura 5.3 mostra um esquema simplificado com apenas dois clusters com dois *patches* cada e suas respectivas ball trees.

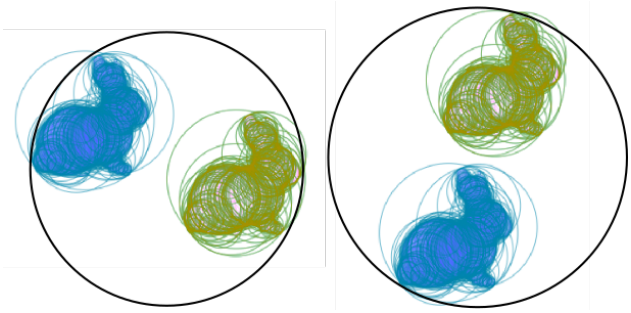


Figura 5.3: Aglomeração baseada na distância. Dois clusters gerados e suas respectivas *Ball trees* representadas pelos círculos pretos.

Para a interface com o usuário, a representação visual de cada cluster é realizada desenhando uma curva de contorno usando um subconjunto das bolas na *ball tree* do cluster expandida por d.

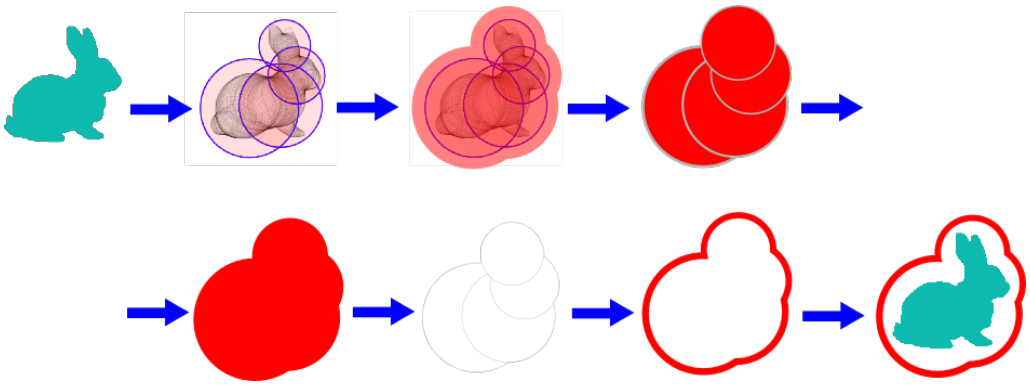


Figura 5.4: Exemplo de composição da curva de contorno para um único *patch*, com $l_{max} = 3$

Simplesmente pintamos os discos com um raio maior e depois novamente na cor de fundo com o raio real. Este subconjunto é controlado por um limite de profundidade l_{max}, tal que uma bola no nível l é desenhada se for uma folha no nível $l < l_{max}$ ou for um nó no nível l_{max}. Por padrão, nosso protótipo usa $l_{max} = 10$, para que não seja necessário renderizar mais de 2^{10} círculos para cada cluster.

A Figura 5.4 ilustra o processo da composição da curva de contorno para um cluster com apenas um *patch* Rabbit, considerando $l_{max} = 3$.

5.3 Prevenção de colisão

Ao posicionar um elemento em uma página, geralmente é desejável evitar sobreposições com outros elementos já presentes. As aplicações de layout comuns devem,

portanto, abrir espaço para o novo elemento por meio de interações adicionais, como arrastar elementos intermediários para alguma parte vazia da página. Essa tarefa pode ser automatizada empregando técnicas de detecção e resposta de colisão.

Como as *ball trees* são muito eficientes na detecção de sobreposições, tudo o que é necessário é um algoritmo para mover os elementos intermediários para fora do caminho.

O algoritmo pode ser acionado enquanto um elemento está sendo movido para sua nova posição ou no final do gesto de arrastar. Um exemplo é mostrado na Figura 1.1.

Implementamos a prevenção de colisões com base em um processo de relaxamento iterativo. O algoritmo 2 descreve como uma única etapa de relaxamento é executada. Para garantir que o limiar de distância d seja mantido entre todos os clusters, o algoritmo deve ser chamado repetidamente até que nenhuma colisão seja detectada ou o número de iterações atinja um determinado valor máximo. A razão para adotar um limite de iteração é evitar um *loop* infinito que pode surgir em layouts com formas não convexas interligadas. Observe que a primeira chamada para Algorithm 2 tem $M = \{s\}$, de modo que apenas colisões com s são testadas. Para as iterações subsequentes M é o conjunto de clusters que foram encontrados em colisão na iteração anterior.

Algorithm 2 - Prevenção de colisão

Input: C {todos os clusters}
Input: M {clusters movidos na iteração anterior}
Input: s {cluster sendo arrastado}
Input: d {limiar de distância}

$M' \leftarrow \emptyset$
for all $(c_i, c_j) \in M \times C$ with $i \neq j$ **do**
  $(t_i, t_j) \leftarrow$ ball trees for (c_i, c_j)
  if $t_i \neq t_j \wedge dist(t_i, t_j) < d$ **then**
    $\vec{v} \leftarrow Penetration(t_i, t_j, d)$
    $M' \leftarrow M' \cup \{t_i, t_j\}$
    if $c_i = s$ **then**
      translate c_j by $\vec{v}$
    else if $c_j = s$ **then**
      translate c_i by $-\vec{v}$
    else
      translate c_i by $-\vec{v}/2$
      translate c_j by $\vec{v}/2$
    end if
  end if
end for
return M'

O Algoritmo 2 requer a função $Penetration(t_i, t_j, d)$, que é uma adaptação do

algoritmo *Tree intersection (TI)* (veja Seção 3.2.2), onde duas bolas são consideradas sobrepostas se sua distância for menor que d. O resultado desta função é um vetor de penetração, ou seja, um vetor de translação que deve ser aplicado às duas árvores para separar o par de nós folha detectado pelo algoritmo de interseção.

5.4 Interface

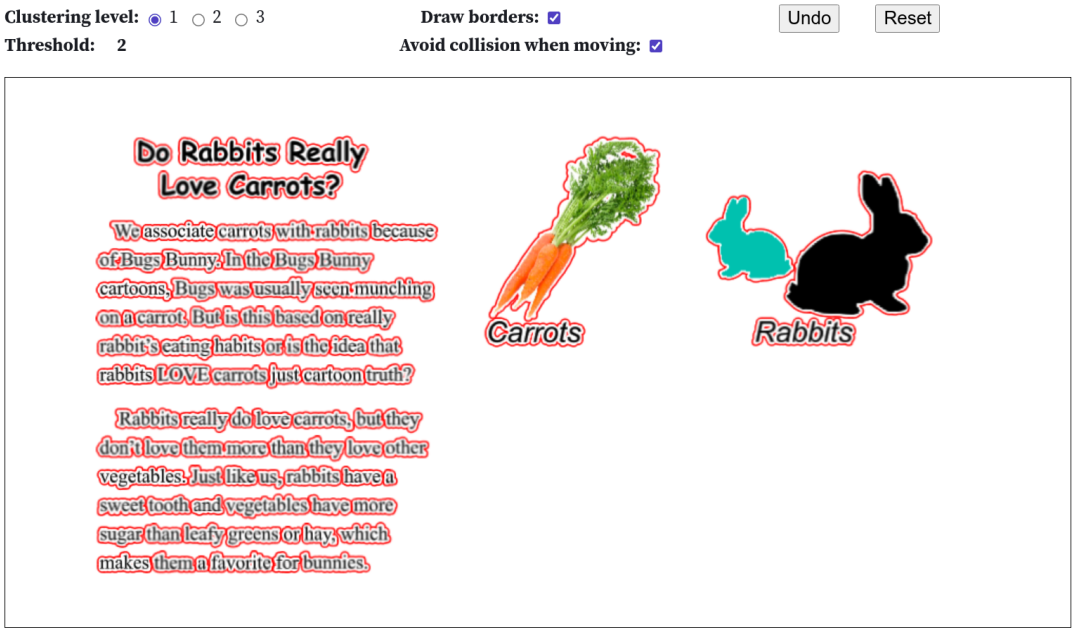
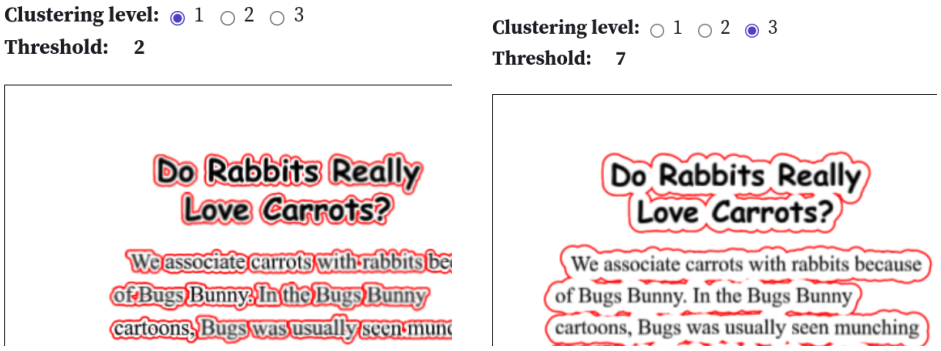


Figura 5.5: Interface de layout de formas: além da tela principal, podemos encontrar os seguintes controles: botões de opção de nível de clusterização, caixas de seleção para ativar/desativar modos de interação, bem como botões de desfazer/resetar.

Nosso protótipo de layout de forma foi criado na plataforma Observable [74] como um web notebook interativo. Ele contém uma breve explicação de uso, a interface principal, bem como todo o código-fonte usado na aplicação.

A interface é mostrada na Fig. 5.5. As formas podem ser organizadas na tela principal arrastando clusters de patch. Os clusters são recriados de acordo com o nível selecionado somente quando muda o estado dos botões de opção "clustering level" (fig. 5.6).

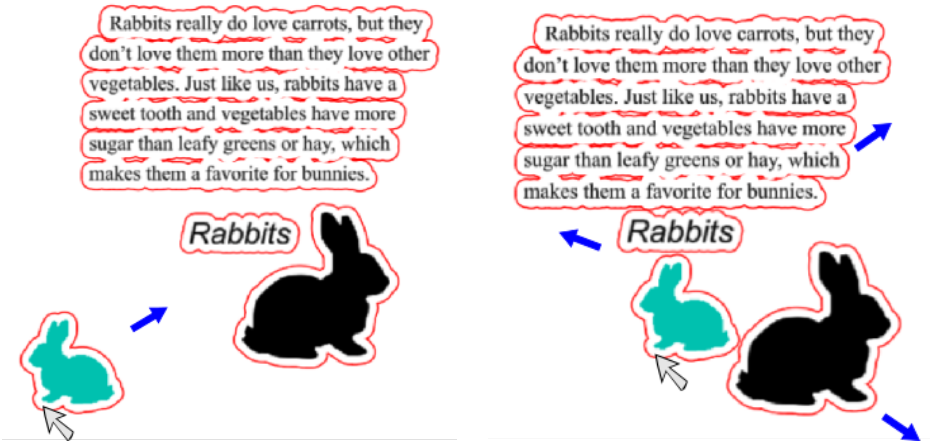
Os clusters são destacados por meio de um contorno vermelho, mas isso pode ser desabilitado desmarcando o botão "draw borders". Ao arrastar um cluster, o algoritmo de prevenção de colisões é chamado repetidamente, a menos que o botão "Avoid collision when moving" esteja desmarcado. Neste caso, o processo de prevenção de colisões é executado apenas no final do gesto de arrastar, ou seja, quando o botão do mouse é liberado. A Figura 5.7 mostra o efeito de prevenção de colisão durante o arraste de um objeto. Para exemplo de ajuste apenas no final veja a



(a) (b)

Figura 5.6: Detalhe da interface em dois estágios de clusterização distintos. (a) nível 1, com limiar de distância = 2, (b) nível 3, com limiar de distância = 7.

fig. 1.1).



(a) (b)

Figura 5.7: Prevenção de colisão durante o arraste. (a) a forma é selecionada e arrastada na direção da seta, (b) os objetos intermediários são empurrados.

5.5 Avaliação empírica

5.5.1 Testes preliminares

A fim de refinar a usabilidade da interface, submetemos a aplicação a um conjunto de experimentos preliminares, antes de avaliar em definitivo a funcionalidade do layout de formas. A interface deste protótipo preliminar foi adaptada para a execução de tarefas específicas para um grupo de usuários de teste [76]. Assim, medimos o tempo gasto para a execução de algumas tarefas, bem como o número de comandos executados pelos usuários. Todas as tarefas tinham um objetivo claro e direto: modificar o layout de um documento de entrada, reajustando a posição dos elementos para um novo layout pré-determinado.

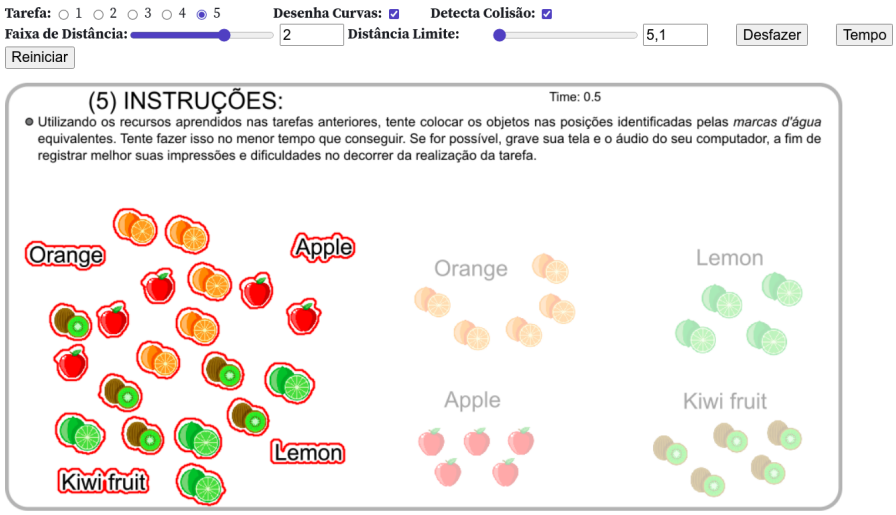


Figura 5.8: Exemplo da interface preliminar durante o exercício de preparação para a execução de uma das tarefas.

O protótipo da interface foi construído em Javascript e testado em um navegador Chrome v92.0.4, sendo disponibilizado para um grupo de 16 usuários de teste que realizaram as tarefas em suas próprias estações de trabalho. A Figura 5.8 mostra a tela da aplicação durante um exercício para a execução de uma dessas tarefas. O perfil selecionado foram de usuários frequentes ou especialistas em softwares de edição gráfica. A interface que serviu de veículo de comparação foi o software de desenho vetorial conhecido do grupo selecionado de usuários: o Inkscape v1.11 [77]. A interface de seleção e agrupamento de elementos desse software segue o método tradicional usado em aplicações desse gênero (veja Seção 2.3).

Cada usuário teve suas tarefas cronometradas e gravadas em video. Uma única tela continha espaço para os dois layouts: o original e a sugestão proposta foram alocados lado a lado, para evitar mudança ou rolamento de tela, preservando o foco do usuário.

Os usuários de teste participaram de um breve treinamento de uso da nova interface, sem limite de tempo, para testar e aprender os comandos: selecionar e arrastar objetos, alternar entre níveis de agrupamento, ativar a clusterização, modificar a distância mínima no nível corrente, copiar e colar clusters, separar objetos de um cluster e detectar colisão. Nenhuma dificuldade foi relatada nas 4 tarefas de treinamento, exceto 3 comentários sobre o comando "copiar e colar". Segundo esses relatos, a dificuldade encontrada nesta ação foi a identificação do ponto exato do alvo neste protótipo preliminar.

Nesse contexto, 3 tarefas específicas de layout foram preparadas para serem executadas no menor tempo possível. Essas tarefas, denominadas respectivamente por T1, T2 e T3, foram estabelecidas com grau de dificuldade distintos. Em T1, o documento de entrada contém somente imagens de alguns ícones (Fig. 5.9), contendo

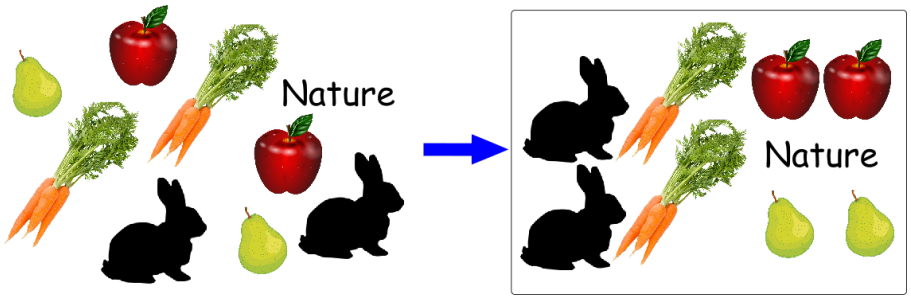


Figura 5.9: Tarefa T1 - Imagem de entrada e layout sugerido.

um total de 14 patches. Em T2, há mais imagens do que texto, num total de 47 patches (Fig. 5.10). Note que em T3 (Fig. 5.11), com 259 patches, há maior quantidade de texto, porém o alvo exige pouca modificação. Todas as imagens de entrada foram digitalizadas com a resolução máxima de 520×400 a 300 DPI.

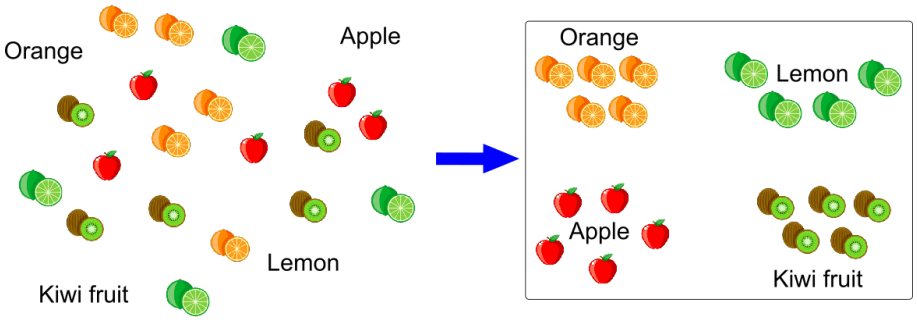


Figura 5.10: Tarefa T2 - mais imagem do que texto.

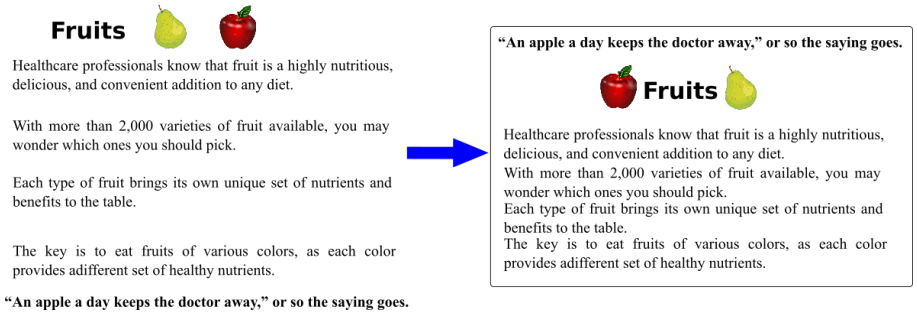


Figura 5.11: Tarefa T3 - mais texto, porém menos modificações sugeridas no layout.

Antes da apuração dos resultados, realizamos o acompanhamento em tempo real da execução das tarefas com 3 usuários experientes. As observações destes usuários especiais, ao longo da execução da tarefa, forneceram parâmetros importantes para algumas melhorias na interface, como posicionamento de alguns ícones, filtro para detecção de colisão e mecanismo usado para separação de objetos.

Inicialmente, analisamos o tempo de execução das tarefas. A Tabela 5.1 compara nossa abordagem de clusterização hierárquica com o método tradicional. Note que o tempo médio de execução utilizando o método tradicional foi 15%, 22% e 13%

mais rápido em T1, T2 e T3, respectivamente. Dado que todos são especialistas em Inkscape, é razoável esperar que o tempo em uma interface nova seja maior, mesmo que o treinamento tenha sido eficiente.

Tabela 5.1: Tempo médio de execução das tarefas (em segundos)

Interface	T1	T2	T3
Clusterização Hierárquica	28	112	39
Tradicional	24	88	34

Apesar do tempo de execução mais elevado, quando analisamos os comandos dos usuários, observamos que o fluxo de trabalho da interface proposta é eficiente e fornece uma perspectiva satisfatória. São dois aspectos importantes que nos conduzem à essa conclusão. Primeiro, o volume de ações e comandos emitidos pelos usuários. Esse total foi medido pela quantidade de cliques com o mouse. A Tabela 5.2 mostra a quantidade média de comandos executados para cada tarefa.

Tabela 5.2: Quantidade média de comandos executados por tarefas

Interface	T1	T2	T3
Clusterização Hierárquica	10	27	9
Tradicional	12	30	15

Observe que os usuários executam menos comandos com a interface proposta. Por exemplo, na tarefa T3, a clusterização hierárquica facilita bastante a seleção de parágrafos inteiros de maneira mais intuitiva e imediata, sem a necessidade de criar uma caixa envolvente. Esta observação foi relevante, pois onde há mais texto a dificuldade de seleção aumenta, quer seja pelo tamanho reduzido dos objetos (letras), quer pela sua disposição longitudinal.

O segundo aspecto importante a ser considerado quanto à eficiência da interface são os tipos de comandos executados por cada usuário. Tanto na interface proposta quanto na tradicional, há maneiras diferentes de executar comandos e se obter o mesmo resultado. Alguns usuários, propositadamente ou não, escolheram o caminho mais complexo ou menos trivial. Especificamente na tarefa T2, usando o método tradicional, o trivial seria selecionar os elementos um a um. Ao invés disso, 4 usuários decidiram: (a) selecionar os objetos do mesmo grupo, (b) movê-los na direção desejada e (c) selecionar individualmente cada um, ajustando-o para a posição alvo. Este processo parece mentalmente mais organizado, pois considera o significado dos objetos e a relação destes com sua disposição no layout. Entretanto, toda a rotina percorre um caminho mais demorado e tedioso. Por outro lado, se considerarmos o sentido inverso, ou seja, documentos que já estejam montados considerando a relação de proximidade entre objetos de um mesmo grupo semântico, a

interface proposta seria mais eficaz, porque já considera conceitualmente estas relações. Assim, tanto do ponto de vista da execução, quanto do processo cognitivo do usuário, nos parece mais simples agrupar para depois ajustar ao *layout*.

Apesar de superficiais, estes testes preliminares foram importantes para estabelecer o protótipo definitivo.

5.5.2 Usabilidade da interface proposta

Testamos nosso protótipo definitivo com dois conjuntos de dados simples, o documento "Rabbit" (mostrado na Fig. 5.5) e o documento "Fruits" (fig. 5.12).

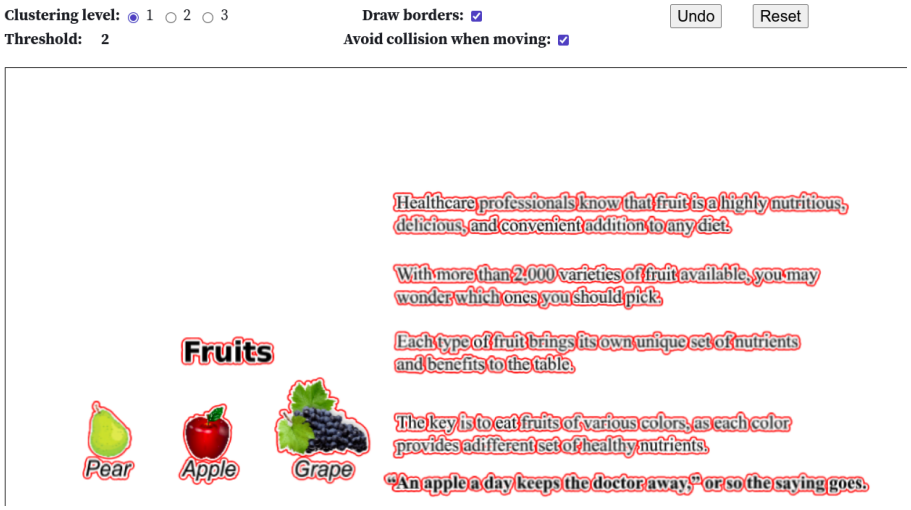


Figura 5.12: Documento "Fruits" utilizado nos testes de interface.

Estatísticas relevantes para cada nível de cluster são mostradas na Tabela 5.3. Observe que esses números correspondem ao layout inicial, pois clusters em níveis maiores que 1 podem mudar em resposta às interações do usuário. A interface permite apenas a seleção de 3 limites de distância (2.0, 3.5 e 7.0), correspondendo aproximadamente à distância entre palavras, entre linhas de texto e entre parágrafos, respectivamente.

Tabela 5.3: Estatísticas dos conjuntos de dados.

	clusters por nível			média de bolas por clusters por nível		
	1	2	3	1	2	3
Rabbit	89	23	5	55.8	215.9	993.4
Fruits	84	16	9	51.5	270.5	481.0

A operação de agrupamento é relativamente custosa, pois requer pelo menos o tempo $O(n^2 \log n)$, considerando que a obtenção das arestas de G requer que todos os pares de *patches* sejam testados para interseção dentro de uma dada distância d e supondo que cada teste seja executado em $O(\log n)$.

Em comparação, o custo de executar o algoritmo *union-find* em G é insignificante. Os tempos necessários para a operação de clusterização em nosso equipamento de teste (consulte a Seção 3.3) estão resumidos em Table 5.4, e observamos um impacto relativamente pequeno no desempenho ao aumentando o limiar de distância, pelo menos para os valores testados.

Tabela 5.4: Tempos para clusterização.

	tempo total (ms) por nível		
	1	2	3
Rabbit	49.8	47.1	48.2
Fruits	46.2	46.5	55.3

O tempo para computar um passo da prevenção de colisão também é limitado por $O(n^2 \log n)$ no pior caso, já que em princípio todos os clusters podem estar em colisão. Em interações típicas, no entanto, um conjunto limitado de clusters é afetado. Para avaliar o desempenho de nossa implementação desta operação para os dois conjuntos de dados, desativamos a prevenção de colisões durante o arrastar e mover de uma forma grande para o meio de uma região contendo texto no nível 1 de agrupamento. Para o conjunto de dados Rabbit, a "cenoura" foi movida, enquanto para o conjunto de dados Fruits, a "uva" foi movida.

A tabela 5.5 mostra o número total de etapas para evitar colisões, o número total de clusters deslocados em todas as etapas e o tempo total necessário. Achamos que todo o processo é rápido o suficiente para garantir uma boa experiência do usuário, embora o grande número de etapas sugira que a convergência seja lenta. Isso pode ser atribuído ao fato de que este esquema de resposta de colisão não faz uso de nenhuma estratégia específica para lidar com o que é conhecido como "empilhamento", situação que surge quando o grafo de dependência de colisão está fortemente conectado.

Tabela 5.5: Estatística de prevenção de colisão.

	etapas	clusters deslocados	tempo (ms)
Rabbit	39	759	118.3
Fruits	24	354	59.1

Capítulo 6

Conclusões

Ball trees são estruturas de dados extremamente flexíveis que encontram aplicação em muitas áreas.

Neste trabalho procuramos investigar a sua utilização para reordenar o conteúdo de um documento digitalizado, examinando todos os aspectos centrais desta tarefa, desde a criação de *ball trees* que aproximam as formas dos *patches* até operações de layout que podem beneficiar-se da relativa facilidade com que as *ball trees* lidam com consultas baseadas em distância.

Investimos um esforço considerável na criação de algoritmos para construir árvores que equilibram o tempo de construção e a eficiência das árvores em relação a um conjunto de consultas baseadas em distância.

Além da implementação original de três algoritmos clássicos propostos por Omohundro[9] (*KD construction*, *Online* e *Bottom up*), foram apresentados três novos algoritmos de construção de ball tree (*Q*, *V0* e *V1*). Além de avaliar diretamente esses algoritmos, também realizamos um extenso conjunto de experimentos para avaliar quão bem as árvores construídas se comportam em relação a consultas de proximidade. Todos os algoritmos foram estendidos com *Otimização EL*, garantindo uma melhoria nos tempos de processamento das consultas. Entre os novos algoritmos de construção propostos, concluimos que o algoritmo Q é o mais prático, pois pode ser ajustado para obter maior fidelidade geométrica em detrimento de maiores tempos de construção ou vice-versa.

Sentimos que ainda há espaço para melhorar os algoritmos apresentados neste trabalho. Por exemplo, o algoritmo ajustável descrito na Seção 4.2 para criar coleções de bolas a partir de imagens pode ser aprimorado com critérios que garantam uma determinada margem de erro, em vez de usar ϵ como uma indicação geral de fidelidade. Da mesma forma, o algoritmo de prevenção de colisões pode se beneficiar de uma atenção mais cuidadosa para o empilhamento de artefatos.

As operações de layout de forma e sua implementação de exemplo, conforme discutido no Capítulo 5, podem ser vistas como um ponto de partida para uma apli-

cação completa com um conjunto mais abrangente de funcionalidades. Por exemplo, *ball trees* podem ser úteis na implementação de laço de seleção, divisão de cluster usando curvas desenhadas à mão livre, espaçamento uniforme de *patches* dentro de uma curva esboçada e muito mais. Há também algumas limitações claras para essas ideias. Por exemplo, a hierarquia implícita fornecida pelos elementos de agrupamento de acordo com os limites de distância pode não ser útil para algumas entradas. Além disso, o desempenho satisfatório que observamos com os dois conjuntos de dados reconhecidamente pequenos pode não se repetir em cenários mais complexos. Uma avaliação mais definitiva dessas preocupações é reservada para trabalhos futuros.

Referências Bibliográficas

[1] WELLER, R., ZACHMANN, G. "Inner sphere trees for proximity and penetration queries." In: *Robotics: science and systems*, v. 2, 2009.

[2] BROUTTA, A., COEURJOLLY, D., SIVIGNON, I. "Hierarchical discrete medial axis for sphere-tree construction". In: *International Workshop on Combinatorial Image Analysis*, pp. 56–67. Springer, 2009.

[3] FURNAS, G. W., BEDERSON, B. B. "Space-scale diagrams: Understanding multiscale interfaces". In: *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 234–241. ACM Press/Addison-Wesley Publishing Co., 1995.

[4] WATANABE, N., WASHIDA, M., IGARASHI, T. "Bubble clusters: an interface for manipulating spatial aggregation of graphical objects". In: *Proceedings of the 20th annual ACM symposium on User interface software and technology*, pp. 173–182. ACM, 2007.

[5] SCHERP, A. "Authoring of multimedia content: A survey of 20 years of research". In: *Semantic Multimedia Analysis and Processing*, CRC Press, pp. 375–398, 2017.

[6] ANGELIDES, M. C. "Multimedia content modeling and personalization", *Encyclopedia of Multimedia*, pp. 510–515, 2008.

[7] BULTERMAN, D. C., HARDMAN, L. "Structured multimedia authoring", *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, v. 1, n. 1, pp. 89–109, 2005.

[8] JOURDAN, M., ROISIN, C., TARDIF, L. "Constraint techniques for authoring multimedia documents", *Constraints*, v. 6, n. 1, pp. 115–132, 2001.

[9] OMOHUNDRO, S. M. *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[10] KÄLLBERG, L., LARSSON, T. "Ray Tracing using Hierarchies of Slab Cut Balls." In: *Eurographics (Short Papers)*, pp. 69–72, 2010.

[11] TATARCHENKO, M., DOSOVITSKIY, A., BROX, T. "Octree generating networks: Efficient convolutional architectures for high-resolution 3d outputs". In: *Proceedings of the IEEE international conference on computer vision*, pp. 2088–2096, 2017.

[12] RETONDARO, L. C. S. C. *Divisão espacial de dados volumétricos regulares para renderização out-of-core*. Dissertação de mestrado, UNIVERSIDADE FEDERAL DO RIO DE JANEIRO, 2008.

[13] FRIEDMAN, J. H., BENTLEY, J. L., FINKEL, R. A. "An algorithm for finding best matches in logarithmic expected time", *ACM Transactions on Mathematical Software (TOMS)*, v. 3, n. 3, pp. 209–226, 1977.

[14] HELD, M., KLOSOWSKI, J. T., MITCHELL, J. S. "Evaluation of collision detection methods for virtual reality fly-throughs". In: *Canadian Conference on Computational Geometry*, pp. 205–210. Citeseer, 1995.

[15] NAYLOR, B., AMANATIDES, J., THIBAULT, W. "Merging BSP trees yields polyhedral set operations", *ACM Siggraph Computer Graphics*, v. 24, n. 4, pp. 115–124, 1990.

[16] IZE, T., WALD, I., PARKER, S. G. "Ray tracing with the BSP tree". In: *2008 IEEE Symposium on Interactive Ray Tracing*, pp. 159–166. IEEE, 2008.

[17] TAKESHITA, D. "AABB Pruning: Pruning of Neighborhood Search for Uniform Grid Using Axis-Aligned Bounding Box", *The Journal of the Society for Art and Science*, v. 19, n. 1, pp. 1–8, 2020.

[18] WALD, I., BOULOS, S., SHIRLEY, P. "Ray tracing deformable scenes using dynamic bounding volume hierarchies", *ACM Transactions on Graphics (TOG)*, v. 26, n. 1, pp. 6–es, 2007.

[19] GOTTSCHALK, S., MANOCHA, D., LIN, M. C. *Collision queries using oriented bounding boxes*. Tese de Doutorado, University of North Carolina at Chapel Hill, 2000.

[20] DONG, X., WANG, P., ZHANG, P., et al. "Probabilistic Orientated Object Detection in Automotive Radar", *arXiv preprint arXiv:2004.05310*, 2020.

[21] ABU-ATA, M., DRAGAN, F. F. "Metric tree-like structures in real-world networks: an empirical study", *Networks*, v. 67, n. 1, pp. 49–68, 2016.

[22] LIU, S.-G., WEI, Y.-W. "Fast nearest neighbor searching based on improved VP-tree", *Pattern Recognition Letters*, v. 60, pp. 8–15, 2015.

[23] MANOLOPOULOS, Y., NANOPOULOS, A., PAPADOPOULOS, A. N., et al. *R-trees: Theory and Applications*. Springer Science & Business Media, 2010.

[24] GUTTMAN, A. "R-Trees: A Dynamic Index Structure for Spatial Searching". In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, p. 47–57, New York, NY, USA, 1984. Association for Computing Machinery. ISBN: 0897911288. doi: 10.1145/602259.602266. Disponível em: <https://doi.org/10.1145/602259.602266>.

[25] DOLATSHAH, M., HADIAN, A., MINAEI-BIDGOLI, B. "Ball*-tree: Efficient spatial indexing for constrained nearest-neighbor search in metric spaces", *arXiv preprint arXiv:1511.00628*, 2015.

[26] WAN, W., LEE, H. J. "Deep feature representation and ball-tree for face sketch recognition", *International Journal of System Assurance Engineering and Management*, pp. 1–6, 2019.

[27] CHENG, C., XIAOLI, L., LINFENG, W., et al. "Algorithm for k-anonymity based on ball-tree and projection area density partition". In: *2019 14th International Conference on Computer Science & Education (ICCSE)*, pp. 972–975. IEEE, 2019.

[28] LIU, T., MOORE, A. W., GRAY, A., et al. "New Algorithms for Efficient High-Dimensional Nonparametric Classification." *Journal of Machine Learning Research*, v. 7, n. 6, 2006.

[29] KONG, M., BAI, Y. "An Efficient Collision Detection Algorithm for the Dual-Robot Coordination System". In: *2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, pp. 1533–1537. IEEE, 2018.

[30] SULAIMAN, H. A., BADE, A. "Bounding volume hierarchies for collision detection", *Computer Graphics*, pp. 39–54, 2012.

[31] ESPERANÇA, C., RETONDARO, L. C. S. C. "Approximating shapes with ball and rectangle trees". https://observablehq.com/@esperanc/approximating-shapes-with-ball-and-rectangle-trees, 2021.

[32] RAMER, U. "An iterative procedure for the polygonal approximation of plane curves", *Computer Graphics and Image Processing*, v. 1, n. 3, pp. 244–256, 1972. ISSN: 0146-664X.

[33] ESPERANÇA, C. "Rectangle approximation trees". `https://observablehq.com/@esperanc/rectangle-approximation-trees`, 2021.

[34] BLUM, H., OTHERS. *A transformation for extracting new descriptors of shape*, v. 4. MIT press Cambridge, 1967.

[35] ATTALI, D., BOISSONNAT, J.-D., EDELSBRUNNER, H. "Stability and computation of medial axes-a state-of-the-art report". In: *Mathematical foundations of scientific visualization, computer graphics, and massive data exploration*, Springer, pp. 109–125, 2009.

[36] AMENTA, N., CHOI, S., KOLLURI, R. K. "The power crust, unions of balls, and the medial axis transform", *Computational Geometry*, v. 19, n. 2-3, pp. 127–153, 2001.

[37] DEY, T. K., ZHAO, W. "Approximate medial axis as a voronoi subcomplex". In: *Proceedings of the seventh ACM symposium on Solid modeling and applications*, pp. 356–366. ACM, 2002.

[38] BORGEFORS, G. "Distance transformations in digital images", *Computer vision, graphics, and image processing*, v. 34, n. 3, pp. 344–371, 1986.

[39] FABBRI, R., COSTA, L. D. F., TORELLI, J. C., et al. "2D Euclidean distance transform algorithms: A comparative survey", *ACM Computing Surveys (CSUR)*, v. 40, n. 1, pp. 2, 2008.

[40] SAITO, T., TORIWAKI, J.-I. "New algorithms for euclidean distance transformation of an n-dimensional digitized picture with applications", *Pattern recognition*, v. 27, n. 11, pp. 1551–1565, 1994.

[41] CHAZAL, F., SOUFFLET, R. "Stability and finiteness properties of medial axis and skeleton", *Journal of Dynamical and Control Systems*, v. 10, n. 2, pp. 149–170, 2004.

[42] RONG, G., TAN, T.-S. "Variants of jump flooding algorithm for computing discrete Voronoi diagrams". In: *Voronoi Diagrams in Science and Engineering, 2007. ISVD'07. 4th International Symposium on*, pp. 176–181. IEEE, 2007.

[43] YUAN, Z., RONG, G., GUO, X., et al. "Generalized Voronoi diagram computation on GPU". In: *Voronoi Diagrams in Science and Engineering (ISVD), 2011 Eighth International Symposium on*, pp. 75–82. IEEE, 2011.

[44] CUNTZ, N., KOLB, A. "Fast Hierarchical 3D Distance Transforms on the GPU." In: *Eurographics (Short Papers)*, pp. 93–96, 2007.

[45] HARALICK, R. M., STERNBERG, S. R., ZHUANG, X. "Image analysis using mathematical morphology", *IEEE transactions on pattern analysis and machine intelligence*, , n. 4, pp. 532–550, 1987.

[46] COEURJOLLY, D., MONTANVERT, A. "Optimal separable algorithms to compute the reverse euclidean distance transformation and discrete medial axis in arbitrary dimension", *IEEE transactions on pattern analysis and machine intelligence*, v. 29, n. 3, pp. 437–448, 2007.

[47] BEZERIANOS, A., BALAKRISHNAN, R. "The vacuum: facilitating the manipulation of distant objects". In: *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 361–370. ACM, 2005.

[48] KOBAYASHI, M., IGARASHI, T. "Ninja Cursors: Using Multiple Cursors to Assist Target Acquisition on Large Screens". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, p. 949–958, New York, NY, USA, 2008. Association for Computing Machinery. ISBN: 9781605580111. doi: 10.1145/1357054.1357201. Disponível em: <https://doi.org/10.1145/1357054.1357201>.

[49] WHITLOCK, M., HARNNER, E., BRUBAKER, J. R., et al. "Interacting with Distant Objects in Augmented Reality". In: *2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pp. 41–48. IEEE, 2018.

[50] NEWN, J., VELLOSO, E., CARTER, M., et al. "Multimodal Segmentation on a Large Interactive Tabletop: Extending Interaction on Horizontal Surfaces with Gaze". In: *Proceedings of the 2016 ACM International Conference on Interactive Surfaces and Spaces*, pp. 251–260. ACM, 2016.

[51] TAN, D. S., MEYERS, B., CZERWINSKI, M. "WinCuts: manipulating arbitrary window regions for more effective use of screen space". In: *CHI'04 extended abstracts on Human factors in computing systems*, pp. 1525–1528. ACM, 2004.

[52] UDDIN, M. S., GUTWIN, C., COCKBURN, A. "The effects of artificial landmarks on learning and performance in spatial-memory interfaces". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pp. 3843–3855. ACM, 2017.

[53] PERLIN, K., FOX, D. "An alternative approach to the computer interface", *Proceedings of Computer Graphics and Interactive Techniques (SIGGRAPH'93)*, pp. 57–64, 1993.

[54] BEDERSON, B., MEYER, J. "Implementing a zooming user interface: experience building Pad++", *Softw., Pract. Exper.*, v. 28, n. 10, pp. 1101–1135, 1998.

[55] BEDERSON, B. B., HOLLAN, J. D. "Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics". In: *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology*, UIST '94, pp. 17–26, New York, NY, USA, 1994. ACM. ISBN: 0-89791-657-3. doi: 10.1145/192426.192435. Disponível em: <`http://doi.acm.org/10.1145/192426.192435`>.

[56] OLSEN, L., SAMAVATI, F. F., SOUSA, M. C., et al. "Technical section: Sketch-based modeling: A survey". In: *Comput. Graph., 33(1):*, pp. 85–103, February 2009.

[57] IGARASHI, T., MATSUOKA, S., TANAKA, H. "Teddy: a sketching interface for 3D freeform design". In: *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pp. 409–416. ACM Press/Addison-Wesley Publishing Co., 1999.

[58] IGARASHI, T. "Freeform user interfaces for graphical computing". In: *International Symposium on Smart Graphics*, pp. 39–48. Springer, 2003.

[59] NIELSEN, J. "Noncommand user interfaces", *Communications of the ACM*, v. 36, n. 4, pp. 83–99, 1993.

[60] LORENSEN, W. E., CLINE, H. E. "Marching cubes: A high resolution 3D surface construction algorithm". In: *ACM siggraph computer graphics*, v. 21, pp. 163–169. ACM, 1987.

[61] COLLINS, C., PENN, G., CARPENDALE, S. "Bubble sets: Revealing set relations with isocontours over existing visualizations", *IEEE Transactions on Visualization and Computer Graphics*, v. 15, n. 6, pp. 1009–1016, 2009.

[62] UHLMANN, J. K. "Metric trees", *Applied Mathematics Letters*, v. 4, n. 5, pp. 61–62, 1991.

[63] MOORE, A. "The Anchors Hierarchy: Using the Triangle Inequality to Survive High Dimensional Data", *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, 01 2013.

[64] RETONDARO, L. C. S. C. "Optimized 2D ball trees". `https://observablehq.com/@lretondaro/optimized-2d-ball-trees`, 2021.

[65] RETONDARO, L. C. D. S. C., ESPERANÇA, C. "Optimized 2D Ball Trees". In: *2021 34th SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*, pp. 33–40. IEEE, 2021. doi: 10.1109/SIBGRAPI54419.2021.00014. Disponível em: <https://doi.org/10.1109/SIBGRAPI54419.2021.00014>.

[66] MATOUŠEK, J., SHARIR, M., WELZL, E. "A subexponential bound for linear programming", *Algorithmica*, v. 16, n. 4-5, pp. 498–516, 1996.

[67] WELZL, E. "Smallest enclosing disks (balls and ellipsoids)". In: *New results and new trends in computer science*, Springer, pp. 359–370, 1991.

[68] MEGIDDO, N. "Linear-time algorithms for linear programming in R^3 and related problems", *SIAM journal on computing*, v. 12, n. 4, pp. 759–776, 1983.

[69] FISCHER, K. *Smallest enclosing balls of balls*. Tese de Doutorado, Citeseer, 1975.

[70] AURENHAMMER, F. "Power diagrams: properties, algorithms and applications", *SIAM Journal on Computing*, v. 16, n. 1, pp. 78–96, 1987.

[71] GAVRILOVA, M. L. *Generalized voronoi diagram: a geometry-based approach to computational intelligence*, v. 158. Springer, 2008.

[72] WELLER, R., ZACHMANN, G. "Inner sphere trees and their application to collision detection". In: *Virtual realities*, Springer, pp. 181–201, 2011.

[73] RETONDARO, L. C. S. C. "Optimized 2D ball trees results". https://observablehq.com/@lretondaro/optimized-2d-ball-trees-results, 2021.

[74] RETONDARO, L. C. S. C., ESPERANÇA, C. "Shape Layout". https://observablehq.com/d/9f3143bee1bbd0e3, 2021.

[75] RETONDARO, L. C., ESPERANÇA, C. "Optimized 2D ball trees for shape layout applications", *Computers & Graphics*, v. 103, pp. 129–139, 2022. doi: 10.1016/j.cag.2022.02.006. Disponível em: <https://doi.org/10.1016/j.cag.2022.02.006>.

[76] RETONDARO, L. C. D. S. C., ESPERANÇA, C. "Layout Semântico de Documentos com Interface Baseada em Agrupamento Hierárquico". In: *Revista de Sistemas de Informação da FSMA*, n. 28, pp. 35–44. FSMA, 2021. Disponível em: <http://www.fsma.edu.br/si/edicao28/FSMA_SI_2021_2_Principal_02.html>.

[77] INKSCAPE. “Inkscape 1.1.1 released”. 2021. Disponível em: <http://www.inkscape.org>.