Dissertação de Mestrado

Indexação de Objetos Móveis utilizando árvore TPKd

GLADSA FARIAS CASTRO

“A vitória do sucesso é metade conquistada quando você ganha o hábito de estabelecer metas e alcançá-las. Mesmo a mais entediante rotina torna-se suportável quando você marcha a cada dia convencido que toda tarefa, não importando quão chata ela seja,

lhe traz mais perto de conquistar seus sonhos.”

Ao Romildo,

Um amigo que conquistei e que fez

com que meu sonho pudesse tornar-se realidade.

Serei eternamente grata.

Agradecimentos

A Deus, por ter me dado, ao longo dessa jornada de estudo, saúde e disposição.

Aos meus pais, Gladstone e Zenaide, pelo estímulo e apoio incondicional desde a primeira hora; pela paciência e grande amizade com que sempre me ouviram, sensatez com que sempre me ajudaram e principalmente, pela importância da construção e coerência de meus próprios valores.

Aos meus irmãos Gladstone, Keilla, Karine e Castro Neto pelo amor e incentivo e pelas ajudas inestimáveis nos árduos momentos e estresse que passei durante o período deste mestrado.

A UVA, Universidade Vale do Acaraú, por permitir o meu afastamento, entendendo as razões que me levaram a fazer esta solicitação, tão necessária para a conclusão de minha pesquisa.

Ao Prof. Angelo, meu orientador, pelo apoio na definição do trabalho, pela orientação, pela compreensão e paciência com a minha situação particular.

Ao Prof. Romildo, meu co-orientador, pela sua generosidade desde o nosso primeiro contato, pelo acolhimento, pela orientação dada, bem como pela disponibilidade e amizade então demonstradas.

Ao Joselias, pelo apoio manifestado, que permitiu reunir as condições que muito me ajudaram a vencer este período de trabalho intenso.

Ao Plácido, pela confiança na minha capacidade.

Às amigas Ana Paula e Fábia, pela compreensão dos vários momentos que não pude participar devido a dedicação no desenvolvimento deste projeto.

Aos colegas Andréia, Edila, Fonseca, Melisa, Shyrlene e Stefan que tanto ouviram minhas lamentações.

Resumo

Objetos espaciais devem ser indexados por uma ou mais chaves de busca, onde cada chave de busca representa sua posição em um eixo de coordenadas. Muitas propostas têm sido publicadas para solucionar o problema de indexação de objetos espaciais. Contudo, o avanço na área de comunicação sem fio fez surgir um nicho de aplicações, como, por exemplo, monitoramento de tráfego aéreo e terrestre, comércio eletrônico móvel móvel e predição de posicionamento de telefones móveis em redes de telefonia celular, que requerem o acesso a dados de objetos espaciais que estão em contínuo movimento. Garantir um acesso eficiente a tais dados implica na existência de técnicas de indexação eficiente. As várias propostas de indexação de objetos móveis, ao invés de utilizarem a atualização contínua da posição dos objetos, utilizam uma atualização periódica da posição e da velocidade do objeto, e uma função linear de previsão de posição futura, baseada em tais informações. Isto faz com que algumas consultas a estes objetos móveis não sejam precisas o suficiente. Neste trabalho será apresentada uma técnica de indexação de objetos móveis baseada na estrutura de árvore kd, onde a função de previsão de movimento é quadrática e baseada nos vetores posição, velocidade e aceleração. A técnica proposta apresenta baixos custos computacionais para gerar e balancear árvores kd, além de garantir um acesso eficiente a entradas na árvore ($\log M$, onde M representa o número de nós folhas). Tais propriedades tornam a técnica proposta mais precisa e eficiente (quanto à localização de dados) do que árvore TPR, por exemplo, proposta por Saltenis *et al*.

Abstract.

Spatial objects must have to be indexed by one or more search keys, where each search key represents its position in a coordinate axis. Many proposals have been published to solve the problem of spatial objects indexing. However, the advancement in the wireless area made come up an application niche, for example, terrestrial and air traffic monitoring, mobile e-commerce and prediction of mobile phones positions in network cellular telecommunications , which demands spatial objects data access that are in continuous movement. To obtain an efficient access to those data, you should make use of efficient indexing techniques. The several proposals of mobile objects indexing, instead of using the objects position continuous update, they use a periodical update of the object position and speed, and based on that, they use a linear function to determine the future object position. In this work will be presented a technique of mobile objects indexing based in the structure of kd tree, where the movement forecast function is quadratic and based on the position, speed and acceleration vectors. This technique proposal presents low computational costs to generate and balance kd trees. Besides, it is guaranteed an efficient entry access in the tree $\log M$, where M represents the number of leaves nodes. Those properties make the proposal technique more precise and efficient (related to data location) than the TPR trees , for example, proposed by Saltenis *et al*.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

A área da computação que atende a usuários que desejam realizar operações computacionais em qualquer tempo e lugar, inclusive estando em movimento, é a computação móvel. Na computação móvel, o ambiente não requer dos usuários a manutenção de uma posição fixa e conhecida, possibilitando, dessa forma, sua mobilidade irrestrita [Bar99]. O desenvolvimento da computação móvel ocorreu em função do desenvolvimento de outras duas tecnologias: os computadores portáteis e as redes de comunicação sem fio.

Entre as aplicações que se tornaram possíveis com o advento da computação móvel estão os Bancos de Dados Móveis. A mobilidade dos Bancos de Dados pode ser vista de vários pontos. Pode-se considerar o Banco de Dados como sendo móvel, estando, portanto, em um dispositivo sem ponto fixo na rede e podendo ser acessado a partir de redes que podem ou não ser conectadas por meios físicos.

Nesse caso, há várias considerações e questões a serem resolvidas, como constantes desconexões da rede, manutenção da integridade e da consistência dos dados.

Móveis também podem ser os clientes do Banco de Dados, que, por estarem a cada momento em uma localidade diferente, podem acessar e atualizar dados a cada momento em um servidor de Banco de Dados diferente, mais adequado a sua comunicação naquele local. Considerando-se um usuário móvel de um Banco de Dados, uma grande variedade de consultas personalizadas baseadas na localização do usuário pode ser realizada. Para tanto, há uma série de estratégias para localização de usuários móveis e identificação da sua posição no espaço. Usuários ou dispositivos que reportam sua localização para realização de consultas, cujos resultados são dependentes dessa localização, são chamados de *objetos móveis* [WXCJ98].

Uma variedade de aplicações envolvendo objetos móveis surge da necessidade de se armazenar informações sobre esses objetos em um Banco de Dados, incluindo as informações sobre a sua localização. Neste caso, informações sobre localização são extremamente inconstantes, uma vez que variam durante todo o tempo em que o objeto se move.

Em sistemas de Banco de Dados convencionais presume-se que os dados armazenados no Banco de Dados permanecem constantes até que mudanças explícitas, através de operações de atualização, sejam realizadas. Este modelo serve para diversas aplicações tradicionais. Mas as aplicações cujos dados requerem atualizações contínuas tornam tal modelo inadequado, pois fazer atualizações constantes no banco é ineficiente, por provocar um declínio na performance do SGBD (Sistema Gerenciador de Banco de Dados) e sobrecargar a rede de comunicação. Por outro lado, reduzir a freqüência de atualizações pode gerar um grau indesejável de imprecisão no resultado de consultas.

Para ilustrar o problema descrito no parágrafo anterior, considere uma aplicação que manipula objetos móveis no espaço. Tais objetos caracterizam-se pela necessidade de tornar a localização espacial (posição) persistente no Banco de Dados na forma de atributo. Adicionalmente, há a necessidade por parte da aplicação em acessar tais objetos através de uma estrutura de indexação, cuja chave de busca seria a posição do objeto. Caso um objeto se encontre em movimento contínuo (um carro, por exemplo), os atributos correspondentes a sua localização (latitude, longitude e altitude, no caso de um espaço tridimensional) precisam ser constantemente atualizados. Naturalmente, isto implicaria em uma degradação de performance do SGBD. Adicionalmente, haveria ainda o problema da consistência do atributo que indica a posição do objeto, pois, quando uma atualização é efetivada, o objeto (em contínuo movimento) já pode estar em outro local, o que tornaria uma consulta ao banco obsoleta. Estes movimentos contínuos apresentam novos desafios para a tecnologia de Banco de Dados.

Um exemplo que seria bastante favorecido com o uso de Banco de Dados de objetos móveis é o setor de plano de saúde na prestação de serviços. Com informações atualizadas a respeito da localização de suas ambulâncias, seria possível praticar um atendimento mais imediato aos clientes em situação de emergência. Um grande número de consultas diferentes poderia ser submetido ao Banco de Dados. Um exemplo destas consultas envolvendo a localização de objetos móveis seria: “indique as ambulâncias mais próximas do local onde ocorreu este acidente”. Esta consulta retornará às ambulâncias que, no momento, têm seus dados no Banco indicando que estão próximas ao local indicado.

É possível, ainda, capacitar o Banco de Dados para armazenar funções que estimam a movimentação dos objetos em relação ao tempo, e, dessa forma, fazer consultas baseadas na previsão de onde deve estar o objeto em determinado momento, confiando-se em sua função de movimentação. Por exemplo:

“indique as ambulâncias que estarão em determinada área dentro de três minutos”.

Portanto, o posicionamento de dados é esperado para ser o alvo central de um grande número de aplicações móveis em lazer, segurança, turismo, tráfego de aplicações e em uma grande quantidade de serviços na internet, onde o posicionamento dos objetos é essencial. Outros campos que poderiam ser favorecidos pelo uso dessas aplicações móveis seriam empresas de frotas de táxi, transporte de cargas ou até aplicações militares, para supervisionar posicionamento de unidades de batalhas [WSX+99].

Em todos os exemplos citados, a exigência na precisão na localização atual e futura dos objetos (por exemplo, aviões e carros) torna-se fundamental para que tomadas de decisões sejam feitas de forma rápida e eficiente.

Portanto, há uma necessidade em se indexar objetos móveis pelos atributos correspondentes as suas coordenadas espaciais, para que, consultas em aplicações que utilizam estes atributos como chave de busca, tenham resultados rápidos e eficientes.

1.2 Indexação de Objetos Móveis

Uma das principais capacidades adicionadas com a computação móvel é a possibilidade de se fazer valer da localização de determinados objetos para oferecer serviços personalizados. Para isso, os sistemas devem estar cientes da constante mudança de localização dos usuários e prover um mecanismo de atualização eficiente para a manutenção da consistência nas informações referentes a essa localização, de forma a não trabalhar com dados defasados.

Quando se trata de objetos que se movimentam e de informações referentes à localização desses objetos, dois tipos de aplicações podem ser desenvolvidos. Pode-se considerar um cliente de uma rede móvel como portador de um dispositivo móvel (telefone celular, palmtop, PDA etc.) que fornece a sua posição e realiza algum tipo de consulta que será respondida baseada na localização fornecida. Por exemplo, um usuário de telefone celular realiza uma consulta para saber quais as farmácias mais próximas de sua localidade no momento da consulta. Estas aplicações são chamadas de aplicações dependente da localização.

Outro tipo de aplicação seria o armazenamento de informações desses objetos que se movimentam em um SGBD, incluindo suas informações de localidade, que deveriam ser constantemente atualizadas, para realização de consultas envolvendo relações entre mais de um objeto, ou envolvendo requisitos temporais, além dos requisitos espaciais. Aplicações como esta são o foco do nosso trabalho.

Como o número de objetos de um Banco de Dados pode ser muito grande, consultas que devem examinar o posicionamento de todos os objetos tornam-se insatisfatoriamente lentas. Estruturas de índices que tenham, como chave de procura, os atributos de localização, poderão resolver estes problemas.

Para tornar possível o acesso à informação destes objetos que estão em contínuo movimento, novas formas de atualização do atributo *posição do objeto* no Banco de Dados vêm sendo pesquisadas. Aqui apresentaremos uma forma onde, ao invés do armazenamento simples da posição, funções do tempo que expressam a posição do objeto deverão ser armazenadas.

As estruturas de índices deverão ser utilizadas para tornar a localização mais eficiente em um posterior acesso. As estruturas de índices convencionais, como árvores B , $B+$ e índices hash, são eficientes para indexar dados unidimensionais (com uma chave de acesso). Em consultas dinâmicas, é necessário

que mais de uma chave de acesso sejam utilizadas, tendo em vista que os dados a serem recuperados estão em um espaço d-dimensional. Diversos tipos de estruturas [GG98, NST98] estão disponíveis para este tipo de consulta, cada uma com uma peculiaridade diferente. A estrutura que iremos usar para indexar o posicionamento dos objetos será baseada em uma *árvore Kd* [Ben75]. A árvore kd é uma árvore binária, homogênea e multidimensional. É uma estrutura caracterizada pela robustez da variedade das consultas, sendo assim mais eficiente em situações onde a natureza da consulta é pouco conhecida [BF79].

1.3 Estrutura da Dissertação

Esta dissertação está organizada da seguinte forma: No capítulo 2, são apresentados conceitos de algumas estruturas de indexação. Na capítulo 3, discutimos propostas de indexação de diversos autores e que utilizam as estruturas descritas no capítulo 2. No capítulo 4, nós conceituamos o problema, apresentamos alguns tipos de consultas que podem ser feitas utilizando o posicionamento dos objetos; introduzimos nosso método de indexação e os algoritmos por ele utilizados. No capítulo 5, nós analisamos o comportamento da estrutura e a comparamos com a estrutura de *árvore TPR* [SJLL00]. No capítulo 6, nós concluímos com uma discussão dos nossos resultados e alguns trabalhos futuros.

Capítulo 2

Estruturas de Indexação Espacial

2.1 Introdução

Neste capítulo citamos características sobre objetos espaciais, alguns tipos de consultas que podem ser feitas para localização destes objetos e algumas estruturas de indexação para objetos móveis com suas características principais e que são necessárias ao capítulo 3.

Objetos no espaço podem ser representados por pontos, linhas ou polígonos. São referenciados geograficamente por coordenadas cartesianas e supõe-se que não sofrerão mudanças e não conterão qualquer informação sobre variação temporal ou espacial. Possuem geometria e características próprias, não estão necessariamente associados a qualquer fenômeno geográfico e podem inclusive ocupar a mesma localização geográfica.

2.2 Propriedades das Estruturas

Algumas aplicações que manipulam objetos espaciais podem ser suportadas pelos SGBD's convencionais, mas alguns destes SGBDs não são equipados para armazenar atributos de dados que mudam continuamente, como a posição de objetos móveis. O motivo é que, nestes Bancos de Dados, os pontos são assumidos para permanecerem constantes até que uma alteração explícita os modifique.

O número de objetos móveis em um Banco de Dados pode ser grande (regiões com grande número de carros por exemplo). Desta forma, torna-se extremamente ineficiente uma consulta que implique em percorrer todos os elementos do banco. Indexar é um mecanismo usado para aumentar a velocidade de acesso a dados. O uso de estruturas de indexação pode minimizar ou solucionar este problema provendo a localização do objeto pelos atributos armazenados.

As consultas sobre dados espaciais podem ter as seguintes características:

- Regiões no espaço euclidiano permitem relações do tipo "em frente de", "contém" e "intercepta", que não têm equivalentes em dados não espaciais;
- Quando a consulta envolve proximidade, a forma e a localização dos objetos devem ser consideradas;
- Para uma consulta por proximidade ser eficiente, os objetos armazenados em um bloco da estrutura de dados devem estar próximos no espaço;

Os esforços para otimizar o desempenho de consultas espaciais concentram-se em desenvolver estruturas de dados adequadas para armazenar e recuperar a posição dos objetos, pois, para o manuseio

eficiente de dados espaciais, um sistema de Banco de dados necessita de um mecanismo que ajude a recuperar os itens de dados rapidamente de acordo com suas localizações espaciais.

Estruturas de dados de vários tipos, tais como árvores *B*, árvores B^+ e tabelas Hash foram projetadas para indexar dados através de uma chave de busca única. Contudo, há um número de aplicações que requerem o uso de mais de uma chave de busca, onde intervalos de pesquisa com o uso destas chaves são operações comuns. Como exemplo, podemos citar: "encontre todos os funcionários que têm idade maior que 40 anos e salário entre 5.000 e 10.000". A pesquisa será efetuada levando-se em consideração os atributos da chave *idade* e da chave *salário*.

Uma operação comum em dados espaciais é a pesquisa por todos os objetos em uma área, como por exemplo: *procure todas as viaturas policiais que se encontram a dois quilômetros de um ponto particular*. Este tipo de pesquisa é importante por ser capaz de recuperar objetos eficientemente de acordo com suas localizações espaciais. As estruturas de dados usadas para pesquisas em uma dimensão tais como árvores *B*, árvores B^+ e Hash não são eficientes quando aplicadas a consultas onde mais de uma chave de busca é requerida.

Para lidar com os objetos de extensão diferente de zero, os métodos de acesso a pontos podem ser modificados usando uma das seguintes técnicas:

- Transformação - os objetos são transformados para uma outra representação, e então são indexados por métodos de acesso para pontos. Eles são mapeados para pontos de um espaço de dimensão maior ou para intervalos no espaço unidimensional.

- Sobreposição de Regiões - o espaço de dados é particionado em regiões. As regiões podem se

sobrepor de modo que cada objeto esteja totalmente contido dentro de uma região.

- Regiões Disjuntas ou Clipping - o espaço de dados é particionado em regiões disjuntas, porém pode acontecer do objeto não estar totalmente contido em uma região. Neste caso existem duas soluções: duplicar ou decompor o objeto. No caso de duplicação, cada região que intercepta o objeto tem uma referência para ele. No caso de decomposição, o objeto é decomposto em pedaços de forma que cada pedaço esteja totalmente dentro de uma região.

2.3 Classificação das Estruturas

As estruturas de dados para suporte de consultas em dados espaciais estão, geralmente, dentro de duas categorias:

- Abordagem *Hash table* que engloba *grid files* e partições *Hash*:

  Em partições *Hash*, a organização dos dados nesta estrutura é obtida com a utilização de um vetor. Dessa forma, é possível acessar qualquer posição de tal estrutura através de um índice. A idéia é ter uma função que converta chaves em índices do vetor. A dificuldade fundamental do uso de um Hash consiste no fato de que o conjunto dos possíveis valores das chaves é muito maior do que o conjunto dos índices de vetor, ou de endereços de memória disponíveis.

  Os grid files funcionam através da correspondência de células de uma grade com o seu devido *bucket* por meio de ponteiros. Um *bucket* é uma unidade de armazenamento físico em disco, onde são mantidos os registros referentes a dados espaciais. Grid files requerem muitos I/Os de disco em vários tipos de consultas. Um problema maior ocorre quando se tem casos de grandes dimensões,

pois o número de *buckets* cresce exponencialmente com a dimensão. Se grandes partições do espaço estão vazias então deveremos ter muitos *buckets* vazios.

- Abordagem de árvores tais como índices multi-chaves, árvore *Kd*, árvore *R*, árvore *Quadrante* entre outras.

  Árvore é um conjunto finito de um ou mais Nós (dados) de tal maneira que existe um Nó denominado raiz e os Nós restantes estão desdobrados em conjuntos separados maior ou igual a zero e cada um destes conjuntos constitui uma árvore. Na árvore, a relação entre os Nós é de hierarquia ou de composição, onde um conjunto de nós é hierarquicamente subordinado a outro. As subárvores não podem ser interligadas.

2.4 Estruturas de Árvores

As estruturas de árvores *R* [Gut84] e R^* [BKSS90] utilizam uma estrutura hierárquica de retângulos d-dimensionais aninhados e é permitida também a sobreposição de regiões. As estruturas de dados de árvores *kd* e *quadrante* são estruturas de indexação para memória principal onde informações preliminares sobre os dados estarão disponíveis sem acesso a disco. Apresentaremos a seguir como pesquisar e como responder a consultas quando a estrutura de dados utilizada é uma árvore.

a) **Consulta por pontos exatos** - é especificado um valor e são procurados os pontos que possuem valores iguais aos especificados.

b) **Consulta por intervalos** - é perguntado por todos os conjuntos de pontos que pertencem ao intervalo ou, se formas são representadas, pelo conjunto de formas que pertencem parcial ou totalmente

ao intervalo.

c) **Consulta vizinhos próximos** - é perguntado pelos vizinhos próximos de um ponto. Por exemplo: se pontos representam cidades, nós podemos querer encontrar as cidades com mais de 100.000 habitantes mais próximas de uma cidade X.

A indexação de um objeto está relacionada com sua posição inicial de referência no espaço, sua velocidade e aceleração iniciais. Algumas estruturas de dados tais como: árvore *R*, R^*, *TPR* e árvore *Kd* serão apresentadas aqui, pois estas estruturas permitem intervalos de pesquisas e são usadas para indicar objetos como pontos ou regiões em um espaço multidimensional.

2.4.1 Árvore R

A estrutura proposta por Guttman [Gut84], chamada árvore *R*, é uma estrutura que representa o espaço de dimensão *n* como uma hierarquia de intervalos n-dimensionais (retângulos no caso bidimensional), provendo um mecanismo de indexação que ajuda a recuperar(consultar) rapidamente os dados de acordo com sua posição espacial.

Uma árvore *R* é uma árvore balanceada, similar a uma árvore *B*, que contém, em seus Nós folhas, índices que apontam para os objetos no BD, ou os Nós correspondem a páginas em disco se o índice estiver armazenado em disco e a estrutura for desenhada de modo que uma consulta espacial necessite visitar apenas um pequeno número de Nós. O índice é completamente dinâmico; inclusões e exclusões podem ser intercaladas com consultas e nenhuma reorganização periódica se faz necessária.

O primeiro passo da construção de uma árvore *R* consiste em envolver cada objeto com um retângulo

menor possível. Esse retângulo é chamado de *MBR (minimum bounding retangle)*. Cada *MBR* é representado na árvore *R* como uma folha. Depois, os *MBR's*, que estão próximos, são colocados dentro de outro *MBR* maior e esse *MBR* é representado, na árvore *R*, como um Nó pai dos *MBR's* que ele contém. O processo se repete até que se tenha um *MBR* que envolva todos os *MBR's*. Este *MBR* irá representar o Nó raiz.

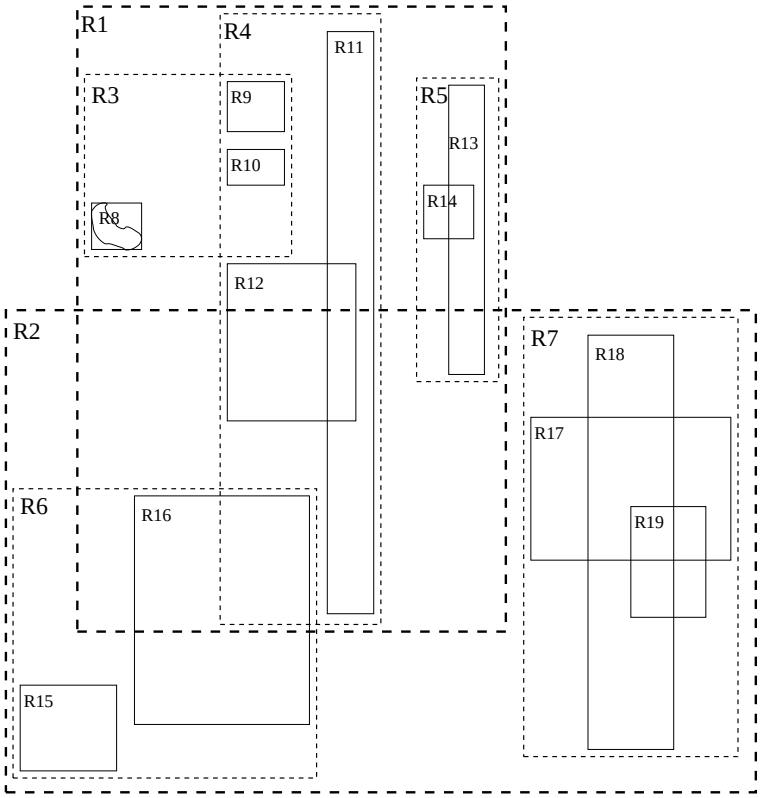


Figura 2.1: Árvore *R* através de retângulos.

Em uma árvore *R* um Nó *n* não folha contém entradas da forma *(retângulo, cp)*, sendo *cp* os endereços dos Nós filhos na árvore *R* e *retângulo*, o retângulo mínimo que envolve todos os retângulos que têm entrada naquele Nó filho, ou seja, que envolve todos os retângulos dos descendentes deste Nó. Um Nó folha contém entradas da forma *(retângulo, oid)* onde *oid* refere-se

a um registro no Banco de Dados descrevendo um objeto espacial e *retângulo* é o retângulo que contém o objeto espacial. Na figura 2.1 pontos e regiões são agrupados em retângulos envolventes mínimos e estes retângulos são agrupados dentro de outros retângulos mínimos, mapeando todos os pontos e regiões de um determinado espaço. Outra forma de visualização deste espaço mapeado pode ser vista na figura 2.2 que mostra, através de uma estrutura de árvore, os pontos e regiões mapeados na figura 2.1.

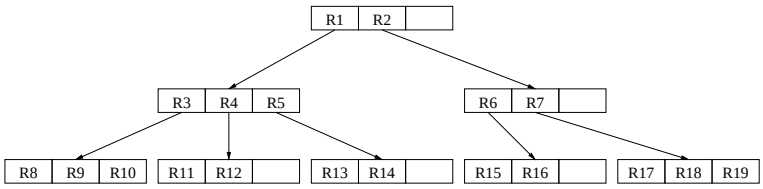


Figura 2.2: Árvore *R* em formato de árvore.

A seguir, as regras de formação de uma árvore *R* de ordem *n*: sejam M e m o número máximo e mínimo de entradas de um Nó respectivamente, tal que $m \leq M/2$. Uma árvore *R* satisfaz às seguintes propriedades:

1. Cada Nó não folha contém entre *M* e m entradas, exceto a raiz.

2. Em um nó folha, para cada entrada *(Ret, oid)*, *Ret* é o menor retângulo que espacialmente contém os dados n-dimensionais armazenados, pela tupla identificada por *oid* se a estrutura de índice for densa, ou, armazenados por uma página se a estrutura de índice for esparsa.

3. Em um nó interno, para cada entrada *(Ret, filho)*, *Ret* é o menor retângulo que espacialmente contém os retângulos descendentes.

4. O Nó raiz tem pelo menos dois filhos, a menos que seja um Nó folha.

5. Todas as folhas aparecem num mesmo nível.

Divisão*(split)* de Nó : para adicionar uma nova entrada a um Nó cheio é necessário dividir as entradas em dois Nós. A divisão deve ser feita de modo que seja improvável que ambos os Nós sejam examinados em pesquisas subseqüentes. Uma vez que a decisão de visitar um Nó dependerá da condição de seu retângulo sobrepor-se à área pesquisada, a área total dos dois retângulos deve ser minimizada para que não seja necessária a pesquisa em múltiplos caminhos na árvore.

2.4.1.1 Algoritmos da árvore R

Algoritmo de Busca - Pesquisa(Nó)

Inicia-se na raiz de forma similar a uma árvore *B*; entretanto, mais de uma subárvore pode ser visitada. Dada uma árvore *R*, façamos *T* igual ao Nó raiz. Encontre todos as entradas de índices cujos retângulos sobrepõem *(overlap)* o retângulo de busca *S*.

Passo 1. Pesquisa Subárvores. Se *T* não é folha, verifique cada entrada *E* para determinar se o retângulo desta entrada *(E.R)* sobrepõe-se ao *S*. Para todas as entradas que se sobrepõem ao retângulo de busca, chame recursivamente o algoritmo de busca, passando o endereço do retângulo filho *(Pesquisa(E.Tid))*.

Passo 2. Se *T* é folha, verifique todas as entradas *E* para determinar se algum filho desta entrada *(E.R)* sobrepõe-se a *S*. Se sim, *E* é um registro do resultado.

É muito comum que o algoritmo de busca percorra vários caminhos da árvore sem que todos eles

realmente levem a objetos que interceptem a área correspondente. Para que isto não ocorra, é necessário fazer a redução de tamanho dos retângulos dos Nós internos para que resulte em menos caminhos percorridos durante a operação de busca. Conseqüentemente, feito isto, o tempo gasto pela operação também será menor. Levando este fato em consideração, os algoritmos envolvidos na construção da árvore *R* (inserção e remoção de elementos) devem utilizar heurísticas que visam reduzir o número de Nós visitados durante a busca.

Algoritmo de Inserção

A heurística de otimização utilizada neste algoritmo é minimizar a área de cobertura dos Nós. Uma primeira situação consiste em escolher em qual dos filhos do Nó interno corrente o objeto deve ser inserido. A subárvore escolhida é sempre aquela cujo retângulo associado necessite de menor expansão de forma a envolver o novo elemento. Uma outra situação consiste em decidir como deve ser feita a divisão (splitting) do Nó em caso de necessidade. Guttman propôs três algoritmos para fazer essa divisão (algoritmos de splitting): o exponencial, o quadrático e o linear, em ordem decrescente de qualidade. Seus nomes indicam qual a sua complexidade. Experimentos mostraram que a diferença de desempenho entre eles é muito pequena, não compensando, assim, o uso de um algoritmo muito complexo. Inserir uma nova entrada *E* na árvore *R* tem os seguintes passos:

Passo 1. Encontre a posição para o novo registro : chamar o algoritmo ***escolhaFolha(E)*** passando o Nó que será inserido, para selecionar o Nó folha *L*, que irá conter a nova entrada *E*

Passo 2. Adicione o registro ao Nó folha: Se o Nó folha (*L*) tem espaço, coloque *E*; senão, chame a função ***divideNo()*** para obter duas novas folhas (*L* e *LL*), que conterão a nova entrada *E* e as entradas de *L* já existentes.

Passo 3. Propagar mudanças para cima: chame o algoritmo ***ajusteTree()*** que irá propagar as mudanças na árvore depois da divisão de Nó *L (split)*. Passar também o novo Nó criado (*LL* se foi realizada uma divisão.

Passo 4. Árvore cresce: Se uma propagação de divisão de Nó causar uma subdivisão da raiz, então crie uma nova raiz cujos filhos são os dois Nós resultantes.

Algoritmo Remoção

Remove um registro *E* de uma árvore *R*.

Passo 1. Encontre o Nó contendo o registro: Chame o algoritmo ***encontreFolha*** passando a entrada *E* para localizá-lo em alguma folha *L*. Pare se o registro não foi encontrado

Passo 2. Remova o registro: remova *E* de *L*

Passo 3. Propague as mudanças: chame o algoritmo ***condensaTree***, passando *L*.

Passo 4. Diminua a árvore: se a raiz tem apenas um filho, depois que a árvore foi ajustada, faça deste filho, a nova raiz.

A árvore *R* é baseada no critério de otimização da diminuição da área de cada retângulo nos Nós. É uma estrutura dinâmica, ou seja, inserções e deleções podem ser mescladas com consultas e nenhuma reorganização periódica global se faz necessária.

Uma série de trabalhos subseqüentes ao de Guttman propõe variantes da árvore *R* com a finalidade de obter desempenhos em consultas de intervalo melhores do que o da árvore *R* original. A principal diferença entre a árvore *R* e suas variantes está nos algoritmos de inserção, sendo que muitas dessas variantes apresentam exatamente a mesma composição que a árvore *R* original. Entre elas estão a *R+−tree*, a R^**-tree* e a *Hilbert R-tree*.

2.4.2 Árvore R*

A árvore R^* [BKSS90] trata não só da otimização da área de retângulos, mas de uma otimização combinada de área, margem e cobertura de cada retângulo em um diretório, para escolher o caminho apropriado para a inserção de uma nova entrada, e minimizar a área coberta pelas regiões.

Assim como a árvore *R*, a árvore R^* é completamente dinâmica: inserções e deleções podem ser intercaladas com consultas e nenhuma reorganização periodicamente global é requerida.

Considerando que a fase de inserção é crítica para um bom desempenho na busca, o desenho de uma árvore R^* introduz uma política chamada reinserção forçada. Se um Nó sofre *overflow* ou seja, Nó cheio, ele não é dividido; ao invés disso, *p* entradas são removidas do Nó e reinseridas na árvore. O parâmetro *p* pode variar. Beckmann *et al.* sugerem que o valor de *p* seja de 30% do número de entradas por página.

Os algoritmos da árvore $R*$ também levam em consideração os seguintes objetivos:

- sobreposições de regiões em um mesmo nível da árvore devem ser minimizadas.
- quanto menos sobreposição, menor a probabilidade que se tenha de seguir vários caminhos de busca.
- o perímetro das regiões deve ser minimizado.
- a utilização do armazenamento deve ser evitadas.

O algoritmo ***EscolheSubÁrvore***, usado com a otimização proposta com a R^*, melhora o resultado

de consultas que buscam objetos em retângulos ou pontos cuja distribuição não é uniforme. Nos outros casos de consultas, os experimentos mostraram que a árvore R^* foi semelhante à árvore R.

A figura 2.4 mostra diversos objetos (p1, p2, m3, m4, etc) distribuídos no espaço e agrupados em retângulos e a figura 2.3 mostra em formato de árvore o mesmo agrupamento destes objetos.

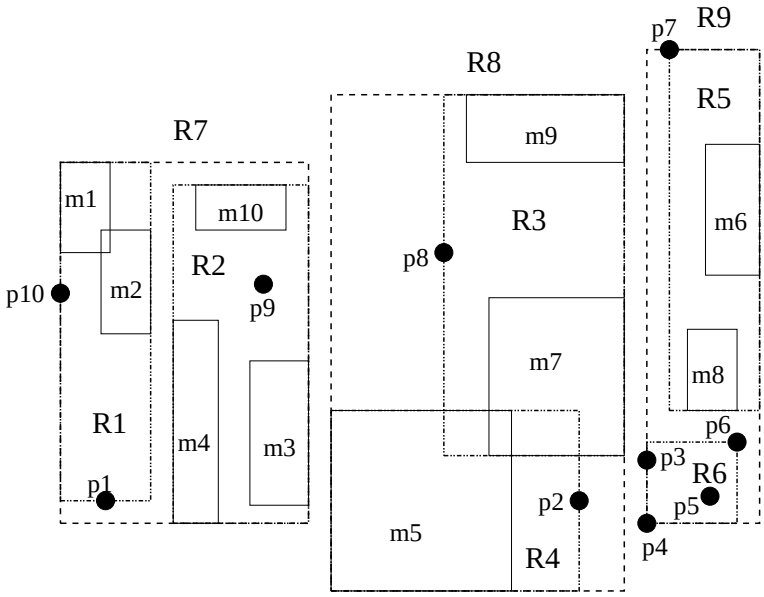


Figura 2.3: Árvore R^* através de retângulos.

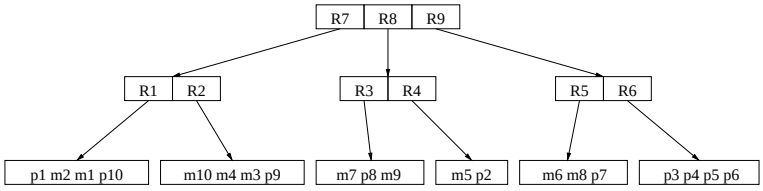


Figura 2.4: Árvore R^* em formato de árvore.

Quando um novo objeto for inserido em uma estrutura de árvore do tipo R ou R^*, pode ocorrer que a folha, indicada para que este novo objeto seja inserido, esteja cheia, isto é, todas as entradas da folha estão preenchidas. Será necessário fazer o que chamamos de *splits*, ou seja, a divisão dos Nós na folha

cheia em duas folhas.

O algoritmo de *split* (divisão de Nó) usado pela R^* usa o seguinte método para encontrar bons *splits*: Em cada eixo, as entradas são sorteadas pelo valor mais baixo e, em seguida, sorteadas pelo maior valor de seus retângulos. O algoritmo de *split* tem os seguintes passos:

Passo 1. invoca um algoritmo chamado ***EscolheEixoDivisor*** para determinar o eixo perpendicular para o qual o *split* será executado.

Passo 2. invoca um algoritmo chamado ***EscolheIndiceDivisor*** para determinar a melhor distribuição em dois grupos no eixo.

Passo3. Distribui a entrada dentro dos dois grupos.

Detalhes destes algoritmos podem ser encontrados no trabalho de Beckmann *et al* em [BKSS90].

Ambas *R* e R^* são não determinísticas em alocação de entradas nos Nós, isto é, diferentes seqüências de inserções deverão construir árvores diferentes.

Para executar reorganizações dinâmicas, as árvores R^* forçam entradas para serem reinseridas durante a rotina de inserções; com isso pode-se ter um decremento da área de cobertura devido à reestruturação e, assim, menos *splits* (divisões) ocorrem e a utilização do armazenamento é melhorada.

Em experimentos realizados [BKSS90], a árvore R^* mostrou-se mais robusta do que as variações da árvore *R*. Devido ao conceito de forçar reentradas, *splits* (divisões) podem ser prevenidos. A estrutura é reorganizada dinamicamente e a utilização do armazenamento é maior do que outras variações da árvore *R*.

2.4.3 Árvore *Kd*

A árvore *kd* [Ben75] é uma estrutura de dados para armazenar um conjunto finito de pontos em um espaço k-dimensional. É uma estrutura projetada para operações na memória principal. Ela permite recuperação da informação por chaves associativas, isto é, pesquisas podem ser feitas com a utilização de várias chaves. Esta estrutura é uma generalização da árvore de pesquisa binária [Knu73] onde cada Nó tem dois ponteiros, e cada um ou é nulo ou é ponteiro para outro Nó na árvore *Kd*.

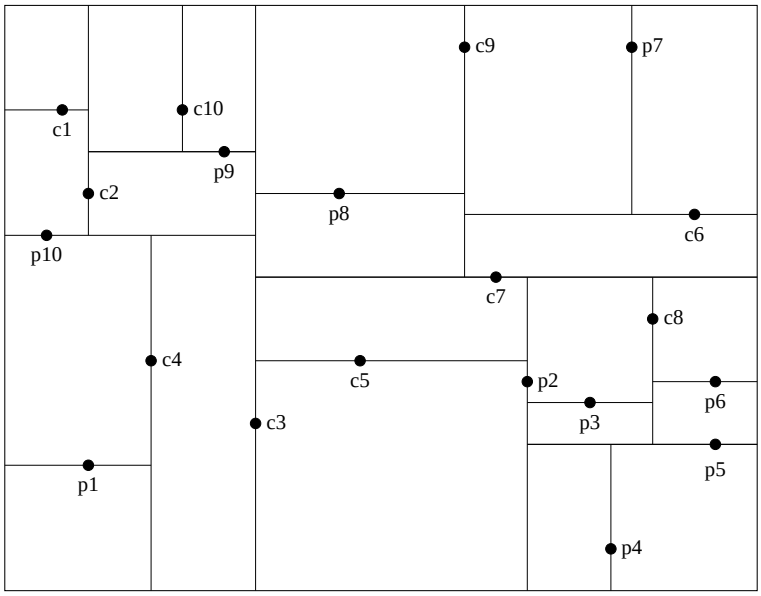


Figura 2.5: Árvore *Kd* representando a divisão do espaço.

A árvore *Kd* é uma árvore binária onde as subdivisões que utilizam hiperplanos de dimensão $(d-1)$ são feitas no espaço de forma recursiva. Esta estrutura lida apenas com pontos; os polígonos são representados pelos seus centróides c_i. É uma estrutura sensível à ordem a que os pontos são incluídos e seus pontos ficam dispersos pela árvore. A figura 2.5 mostra um espaço subdividido segundo as orientações de uma árvore *Kd*, e na figura 2.6, temos a representação de divisão do mesmo espaço em

formato de árvore.

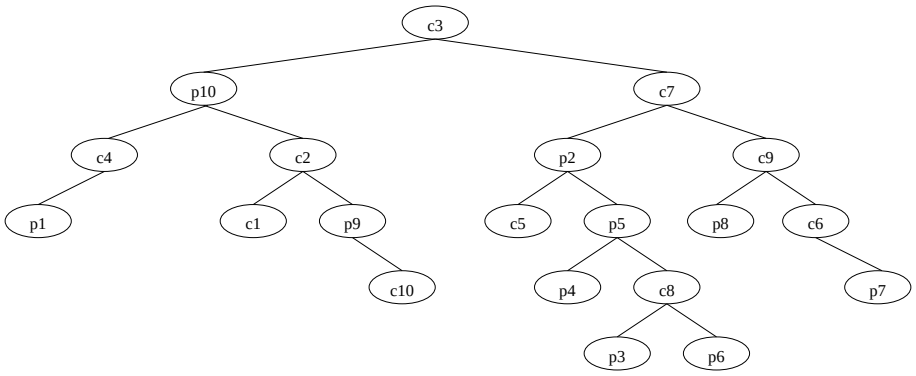


Figura 2.6: Árvore *Kd* em formato de árvore.

Associado a cada Nó, e não necessariamente armazenado como campo, tem-se um discriminador *DISC* que indica a chave utilizada para a divisão do subespaço associado ao Nó. Este discriminador é um inteiro entre 0 e $k-1$ inclusive o que é dado em função da profundidade do Nó. Sendo l a profundidade do Nó P, *k* representa a dimensão do espaço utilizado pela árvore, *DISC*(P) será o resto da divisão de l por k.

As chaves de um Nó P são denominadas $K_0(P), ..., K_{k-1}(P)$. Os ponteiros são $LOSON(P)$, que aponta para o filho da esquerda, e $HISON(P)$, que aponta para o filho da direita. Sendo j o discriminador de P, tem-se: para qualquer Nó Q em $LOSON(P)$, $K_j(Q) < K_j(P)$ e, para qualquer Nó *R* em $HISON(P)$, $K_j(R) \geq K_j(P)$.

2.4.3.1 Algoritmos da árvore *Kd*

Nesta seção, apresentaremos o funcionamento dos algoritmos básicos da árvore *Kd*: inserção, deleção e pesquisa.

Algoritmo de Inserção

O algoritmo usado para inserir um Nó em uma árvore *Kd* é também usado para pesquisar um Nó específico na árvore.

Neste algoritmo, um Nó *P* é passado para ser inserido na árvore *Kd*. Se há um Nó na árvore, igual ao Nó *P*, o endereço deste Nó é retornado, senão, o Nó é inserido na árvore.

Passo 1. Checar se a árvore está vazia: se a árvore estiver vazia, o Nó *P* torna-se-á raiz, o valor de seu discriminante será inicializado, e seus filhos (direito e esquerdo) serão nulos.

Passo 2. Comparar os Nós: Se a árvore não for vazia, comparar as chaves *K* dos Nós com *P*. Se K_i*(P)* = K_i*(Q)* para $0 \leq i \leq k-1$, isto é, os Nós são iguais; então, retorne *Q*. Se K_i*(P)* $> K_i$*(Q)* continuar na subárvore à direita de *Q* e, caso contrário, continuar na subárvore à esquerda de *Q*. Se um Nó *X* deste caminho não tiver filhos, execute o *passo 3*.

Passo 3. Inserir um Nó na árvore: O Nó será inserido como filho do Nó *X* encontrado no *passo 2* e seus filhos serão nulos.

Algoritmo de Busca

Buscas em uma árvore *Kd* consistem em retornar para uma consulta, um conjunto de registros válidos que satisfaçam algum critério.

Consulta a pontos exatos

Tipo de consulta simples que responderá se um registro específico se encontra na estrutura de dados. O algoritmo de inserção, apresentado acima, pode ser usado para buscas deste tipo, através da utilização

do *passo 2* . Uma consideração a ser feita é que, se as consultas a serem propostas são todas deste tipo, uma árvore *Kd* não deverá ser usada para armazenamento de pontos.

Consulta a regiões

Neste tipo de consulta, o conjunto dos registros a serem recuperados pela consulta deve interceptar uma região especificada. O algoritmo para realizar uma pesquisa em uma região não necessita especificamente saber a definição da região na qual ele irá pesquisar. Isto fica a cargo de dois outros algoritmos: no primeiro, ***pesquisa região*** é passado um Nó na árvore e é retornado verdade se e somente se, o Nó estiver contido na região. No segundo, ***intercepta*** são passados os limites de um retângulo e é verificado se uma região qualquer intercepta os limites do retângulo especificado.

O algoritmo que consulta regiões, ***pesquisa região***, é iniciado passando-se a raiz e os limites da região a ser pesquisada. Ele usa o limite armazenado em cada Nó da árvore para determinar se é possível que algum de seus descendentes possa estar situado na região inicialmente pesquisada. A subárvore é visitada pelo algoritmo somente se existir esta possibilidade. Veja abaixo os passos do algoritmo ***Consulta***:

Passo 1. Dado um retângulo *R*, visita-se a raiz *T*

Passo 2. Se a região correspondente a *T* está contida em *R*, reportar *T* e terminar.

Passo 3. Se *T* é um Nó folha, reportar os pontos de *T* que satisfazem a consulta e terminar.

Passo 4. Se a partição da esquerda de *T* intersepta *R*, execute o algoritmo ***Consulta*** para o filho esquerdo de *T*.

Passo 5. Se a partição da direita de *T* intercepta *R*, execute o algoritmo ***Consulta*** para o filho direito de *T*.

Um exemplo de busca em árvore *kd* é mostrado: dado um intervalo de idade de 35 a 55 anos e um intervalo de valor de salário de 100 a 200, pesquisar os registros que atendem a esta condição de pesquisa. A pesquisa na árvore 2.7 é feita como segue: como o salário de 150 da raiz está dentro do intervalo, devemos explorar ambos os filhos. No Nó idade = 60, os filhos da esquerda estão totalmente no intervalo; então, movemo-nos para o Nó salário = 80; agora o intervalo está dentro indo para a direita onde encontramos os registro (50,100) e (50,120). Retornando para o filho à direita, na raiz, o Nó idade = 47 diz que devemos procurar em ambas as subárvores. No Nó salário = 300, devemos ir somente para a esquerda e encontramos o ponto (30,260) que está fora do intervalo. No filho da direita do Nó idade = 47, encontramos dois outros pontos e ambos fora do intervalo.

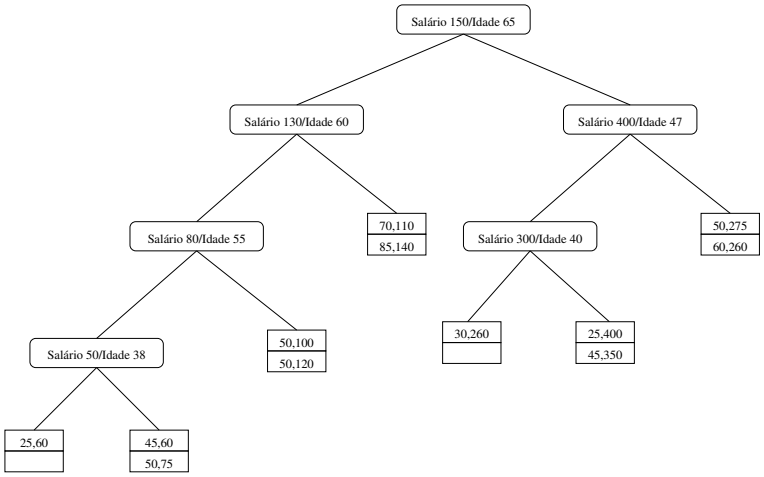


Figura 2.7: Exemplo de consulta em árvore *Kd*.

Algoritmo de Remoção

Neste algoritmo recursivo, é passado um ponteiro *P* para um Nó na árvore *Kd*. Ele apaga o Nó

apontado por *P* e returna um valor, o qual é um ponteiro para a raiz da subárvore resultante da deleção.

Passo 1. Testar se o Nó *P* é uma folha: se o Nó é uma folha, a deleção acontece de forma direta, com a remoção do Nó.

Passo 2. Decide que lado (esquerdo ou direito) da árvore será usado para encontrar o sucessor de *P*: se o filho à direita do Nó *P* for nulo, execute o *passo 4*.

Passo 3. Encontrar o substituto do Nó *P* na subárvore direita: encontrar um Nó *Q* de valor mínimo na subárvore direita, no mesmo discriminante de *P*. Ir para o *passo 5*.

Passo 4. Encontrar o substituto do Nó *P* na subárvore esquerda: encontrar o Nó *Q* de valor máximo no mesmo discriminante de *P*, na subárvore esquerda do Nó a ser apagado.

Passo 5. Apagar o Nó Q: Este é um passo recursivo que irá liberar o Nó *Q* e assim ele poderá ser a nova raiz.

Passo 6. Fazer *Q* a nova raiz: o discriminante do Nó *P* é repassado para o Nó *Q*, bem como os filhos à direita e esquerda do Nó *P*.

Mais detalhes destes algoritmos podem ser encontrados em Bentley [Ben75].

2.4.3.2 Custos da árvore *Kd*

É mostrado por Bentley em [Ben75] que a inserção randômica de um Nó, pode ser implementada por um algoritmo de classe $O(\log n)$ onde n é o número de registros. Consultas a pontos específicos, com t chaves de pesquisa, e de dimensão do espaço pesquisado chamada de *k*, podem ser respondidas por

uma árvore *kd* em $O(n^{\frac{k-t}{k}})$. Testes empíricos mostram que consultas a vizinhos próximos são respondidas por algoritmos que têm média de execução $O(\log n)$. Remoção da raiz requer $O(n^{\frac{k-1}{k}})$ tempo de execução; a remoção de um Nó pode ser feita em $O(\log n)$.

Nos experimentos com conjuntos de pontos semidinâmicos [Ben90], isto é, com conjuntos que suportam remoções e recuperação de pontos removidos, mas não suportam inserções, a utilização da árvore *kd* para consultas de vizinhos próximos tem desempenho reduzido de $O(\log N)$ para $O(1)$.

A árvore *kd* é sensível a ordem na qual os pontos são inseridos e é uma estrutura que se caracteriza pela sua robustez para uma grande variedade de consultas [BF79], sendo, assim, mais eficiente em situações onde a natureza destas consultas é pouco conhecida.

Consulta dinâmica é uma técnica de consulta para responder a consultas por intervalo de tempo em conjuntos de dados com várias chaves [AWS92]. Este mecanismo é bem aceito para conjunto de dados com várias chaves onde o resultado da pesquisa é ajustado completamente em uma única tela. Experimentos realizados por Jain, em [JS94], mostram que, em consultas dinâmicas, a árvore *kd*, entre outras estruturas de árvores, usa menos memória mas tem um grande custo adicional *(overhead)* para pesquisa. A árvore *kd* é também recomendada para ser usada quando a distribuição de dados não é uniforme e é muito mais fácil de ser construída se comparada à árvore *Quad*. Estes resultados foram obtidos assumindo um conjunto de dados congelados onde novas inserções, deleções ou alterações não eram permitidas [JS94].

2.4.3.3 Extensões da árvore *Kd*

A disposição dos pontos em uma árvore *Kd* pode estar espalhada por toda a árvore. A *kd adaptativa* [BF79] investiga uma maneira de melhorar isto, através da escolha de uma divisão *(split)* tal que seja encontrado o mesmo número de elementos em ambos os lados. Apesar disso, os hiperplanos da divisão, que ainda estarão paralelos aos eixos, não precisarão conter pontos e suas direções não precisam alternar. Como resultado disto, os Nós responsáveis pela divisão não fazem parte dos dados de entrada e todos os pontos estão armazenados nas folhas. Nós interiores contêm uma dimensão (por exemplo x e y) e as coordenadas da correspondente divisão. A divisão é cíclica até cada subespaço conter apenas um certo número de pontos. A *kd adaptativa* é uma estrutura preferencialmente estática pois é difícil assegurar o balanceamento na presença de freqüentes inserções e exclusões. Esta estrutura trabalha melhor se todos os dados forem conhecidos *a priori* e se forem raras as atualizações.

A árvore *Kdb* [Rob81] combina as propriedades da árvore *kd adaptativa* [BF79] e da árvore *B* [CLR89] para manusear pontos multidimensionais. Ela particiona o universo da mesma forma de uma árvore *kd adaptativa* (escolhe o *split* tal que encontre o mesmo número de elementos de ambos os lados) e associa o subespaço resultante com Nós da árvore. Cada Nó interior corresponde a uma região. Regiões correspondentes a Nós no mesmo nível são mutuamente disjuntas. Os Nós folhas armazenam os pontos de dados que estão alocados na partição correspondente.

A estrutura de uma *Kdb* consiste em uma página de região e uma página de ponto. A página de região tem estrutura: <region, page-ID> e a página de ponto: <point, record-ID>. É uma árvore perfeitamente balanceada todavia ela não assegura utilização do armazenamento. A figura 2.8 mostra como uma distribuição de pontos pode ser armazenada na árvore *Kdb*.

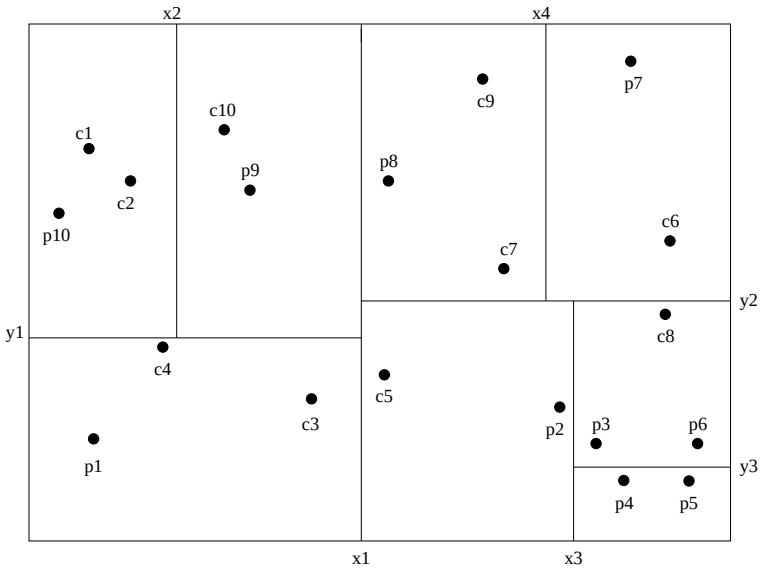


Figura 2.8: Árvore *Kdb* particionando o espaço.

Em 2003, uma estrutura de dados dinâmica, baseada na árvore *Kdb*, em uma extensão da árvore *Kd*,foi proposta por *Procopiuc et al*. [PAAV03]. Esta estrutura chamada árvore *Bkd* mantém seu espaço de utilização de armazenamento alto e um bom desempenho para consultas e alterações.

Os principais ingrediente no projeto da árvore *Bkd* são os algoritmos da árvore *Kdb* e os métodos logaritmos para transformar uma estrutura de dados estática em dinâmica: ao invés de manter uma árvore dinamicamente rebalanceada após cada inserção, é mantido um conjunto de estruturas de árvores *Kdb* estática, e as alterações são feitas pela reconstrução de uma cuidadosa escolha de conjuntos em intervalos regulares e assim são mantidos 100% de utilização do espaço da árvore estática *Kdb*.

A árvore *BKd* consta de um conjunto de árvores *Kd* balanceadas. A construção da árvore é feita de maneira *top-down* onde o primeiro passo é sortear as entradas de ambas as coordenadas e, assim, a árvore é construída de maneira recursiva. A figura 2.10 mostra o conjunto de árvores *kd*.

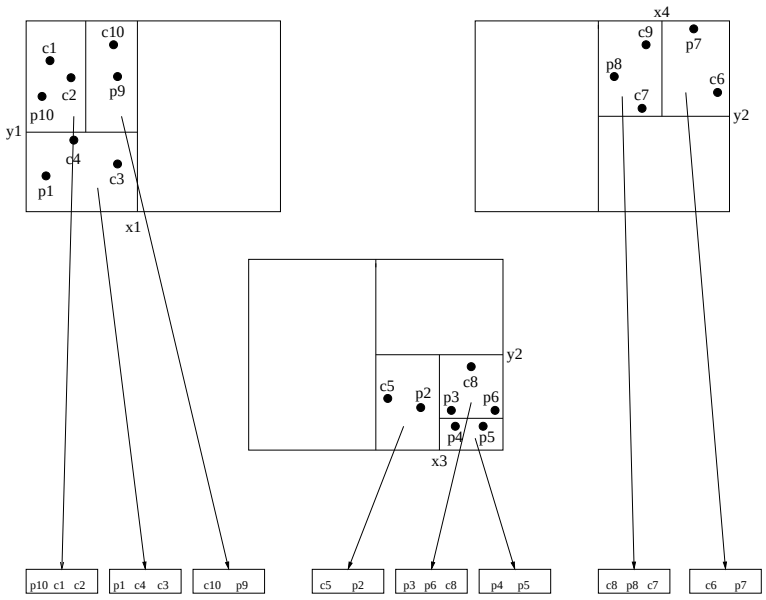


Figura 2.9: Árvore *Kdb*.

Uma árvore *Bkd* com n pontos no plano consta de $\log(n/m)$ árvores *Kd*. No algoritmo de deleção, cada árvore é consultada para encontrar a árvore *T* que contém o ponto a ser deletado.

Muitas inserções são feitas diretamente em estruturas na memória. Se a estrutura estiver cheia, encontra-se a primeira árvore vazia; extraem-se todos os pontos da estrutura de memória T junto com o ponto a ser inserido e carregamo-los na primeira árvore *Kd* vazia. Em outras palavras: pontos são inseridos na estrutura de memória interna e gradualmente "empurrados"para grandes árvores *Kd* por periódicas reorganizações de pequenas árvores *Kd*, dentro de uma árvore *Kd* maior. Na árvore *Kd* maior as reorganizações acontecem com menor freqüência.

Kreveld e Overmars [vKO91] em 1991, propuseram a árvore *dividida Kd*. Esta estrutura é considerada totalmente dinâmica e permite inserções e remoções de *n* pontos em $O(\log n)$ no pior caso. Uma árvore $2-$dimensional, representando um conjunto S, de n pontos, é uma árvore *dividida Kd*

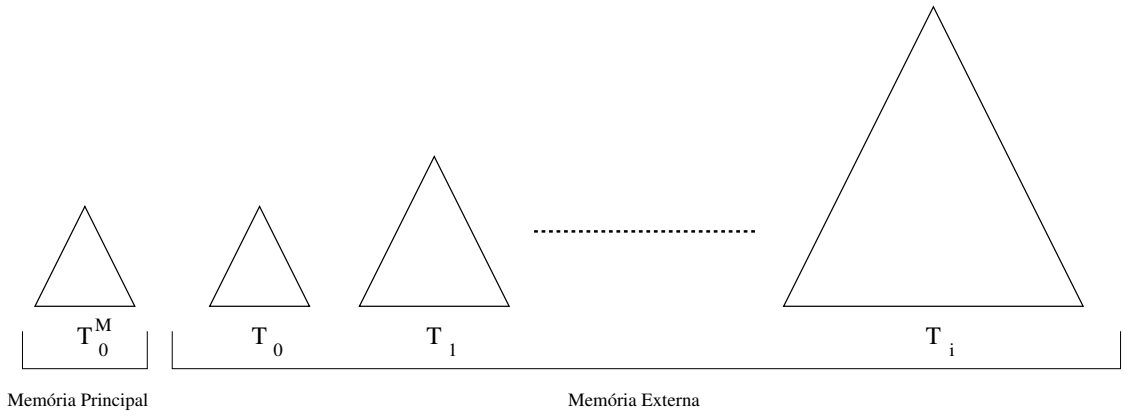


Figura 2.10: Conjunto de árvores Kd.

se, somente se, a árvore constar de uma árvore superior, que divide o conjunto de pontos na segunda coordenada e em cada folha da árvore superior, uma árvore inferior, que divide o conjunto de pontos na primeira coordenada e armazena os pontos do conjunto S nas folhas. Veja a figura 2.11 mostrando um exemplo da representação de pontos no espaço e de sua representação na estrutura de árvore *dividida Kd*.

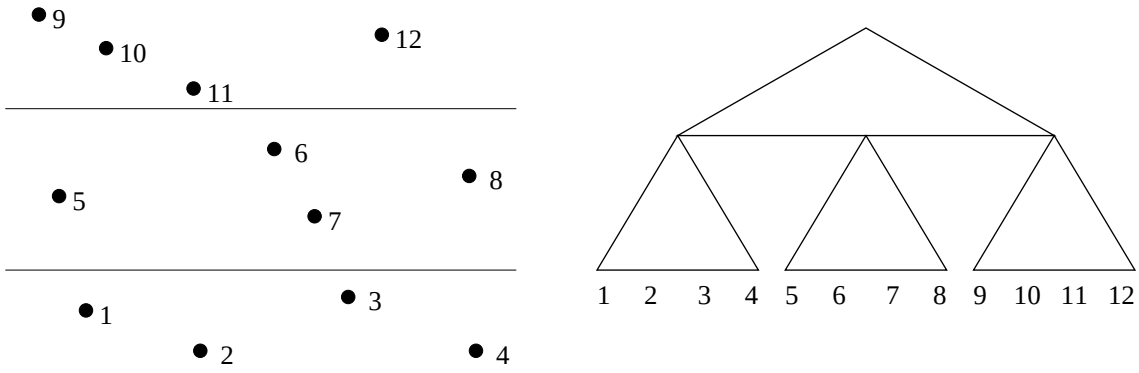


Figura 2.11: Árvore *dividida Kd*.

Segundo os autores, as seguintes condições de balanceamento são esperadas:

a) a árvore superior é uma árvore de pesquisa balanceada e contém pelo menos $2(\sqrt{n/logn})$ folhas;

b) cada árvore inferior é uma árvore de pesquisa balanceada e contém pelo menos $2\sqrt{nlogn}$.

A árvore *dividida Kd* responde a consultas a pontos exatos em $O(\log n)$ e consultas de intervalo em $O(\sqrt{nlogn} + k)$. Este tipo de árvore tem sido construída também dividindo e concatenando em ambas as coordenadas. Isto pode ser útil em pacotes gráficos e sistemas de Banco de Dados se estes querem trabalhar em um subconjunto (intervalo ortogonal) do conjunto completo de pontos na estrutura.

A *dividida Kd* pode ser extendida para altas dimensões, mas o tempo de consultas de intervalo cresce, sendo efetuado em $O(n^{1-1/d} \log^{1/d} n + K)$ tempo.

Outras informações sobre árvore *Kd* e de outras estruturas baseadas em árvore *Kd* podem ser encontradas nos trabalhos de Beckley [BER85] e Moore [Moo90].

2.5 Considerações Finais

A escolha de qual estrutura de indexação utilizar para responder a consultas de pontos exatos e intervalos de buscas leva em consideração diversos conceitos que servem para caracterizar qualquer estrutura de dados de acordo com as suas principais características, destacando as semelhanças e diferenças com outras estruturas. Alguns pontos devem ser considerados tais como: o tipo de dados a ser armazenado, a quantidade de atributos de buscas que estarão envolvidos nas consultas, as operações que serão suportadas pela estrutura, que objetos serão armazenados na estrutura e muitas outras considerações que servirão de suporte para a escolha adequada da estrutura a ser utilizada.

A dificuldade de comparação das estruturas de indexação existe porque há muitos critérios diferentes

Comparativo das Estruturas Apresentadas							
Estruturas	R	R^*	Kd	KdB	$KdAdap$	BkD	$DivKd$
Técnicas p/ Indexar	sobrep.	sobrep.	transf.	transf.	recortes	transf.	transf.
Ordem da estrutura	variável	variável	binária	variável	binária	binária	binária
Altura da árvore	constante	constante	variável	constante	constante	variável	constante

Tabela 2.1: Quadro comparativo

para diferenciar o que seria uma estrutura ótima além disso, existem muitos parâmetros para determinar bom desempenho. Eficiência de Tempo (pesquisas espaciais devem ser rápidas) e eficiência de espaço (um índice deverá ser pequeno em tamanho comparado com o dado a ser endereçado e desta forma garantir uma certa utilização do armazenamento) dependem do dado a ser processado e das consultas a serem requeridas. O número de inserções e remoções também afeta o desempenho da estrutura.

Alguns critérios mostrados na tabela 2.1 foram definidos para resumir o estudo das estruturas descritas neste trabalho. A forma de indexar objetos: transformação, sobreposição e recortes; a ordem da estrutura de indexação ou seja, o número de objetos por nó; a altura da árvore que é determinada pelo tamanho do caminho que vai da raiz até a folha.

O estudo das estruturas de indexação apresentadas foram necessárias para que houvesse um bom entedimento das propostas de indexação apresentadas no capítulo seguinte.

A árvore *Kd* possui algumas características que nos fizeram escolhe-la como a estrutura de indexação base para a nossa estrutura de indexação proposta: uma estrutura de indexação binária onde apesar da árvore ter uma profundidade maior, inclusões, alterações e consultas tem acesso mais rápido pois

caminhos únicos serão seguidos devido as chaves de buscas; ser uma estrutura de memória principal, reduzindo os custos adicionais de acesso a disco e consequentemente maior rapidez nas respostas; uma estrutura de fácil implementação com algoritmos básicos simples.

Capítulo 3

Indexação Espacial de Objetos Móveis

3.1 Introdução

Muitos trabalhos envolvendo a indexação de objetos móveis vêm sendo propostos para melhorar o desempenho de consultas que utilizam o posicionamento de objetos como critério de pesquisa. Expomos aqui alguns destes trabalhos, mostramos suas teorias e fazemos algumas análises sobre eles.

3.2 Indexando Objetos Móveis

No trabalho de **Kollios, Gunopulos e Tsotras** em 1999, [KGT99] técnicas de indexação em uma e em duas dimensões são propostas. Neste modelo os objetos são vistos como pontos, movendo-se com velocidade constante, partindo de uma posição específica em um instante específico. Usando estas informações a localização de um objeto em qualquer tempo futuro poderá ser calculada se suas

características de movimento permaneçam constante. Os objetos são responsáveis por alterarem suas informações de movimento em cada tempo quando suas velocidades ou direção mudam. Duas formas de representação do problema são investigadas. Na primeira representação a trajetória do objeto móvel é desenhada como linhas no plano tempo-localização *(t,y)*. Assim, a posição do objeto é obtida através da equação $y(t) = v(t - t_{ref}) + s(t_{ref})$, onde v é a velocidade, s é a intersecção que pode ser computada pela informação do movimento, t o tempo inicial e t_{ref} o tempo de referência no momento que se deseja obter a posição do objeto. Uma outra representação utilizada neste trabalho foi o mapeamento de uma linha do plano (t, y) em um ponto no plano bi-dimensional; assim, a linha com equação $y(t) = s + vt$ é representada pelo ponto $(s(t_{ref}), v)$ no plano. Os autores consideram a segunda abordagem em seus estudos.

Para indexar em uma dimensão foi usado um método para testar se uma região de consulta e um hiperplano se sobrepõem. A estrutura mais aceita, segundo os autores, dependerá da distância dos pontos mas, as estrutura de dados baseadas em árvores *Kd* são mais aceitas que as baseadas em árvores *R*, pelo motivo que na árvore R, para agrupar pontos em regiões, os *splits* (divisões) usam uma dimensão, a que intercepta. Por outro lado, métodos baseados em árvores *Kd* usam ambas as dimensões para as divisões e assim, é esperado, que se tenha melhor desempenho em consultas. Mas esta assertiva não está demonstrada no trabalho.

Na forma de representação que utiliza o mapeamento de uma linha do plano *(t,y)* em um ponto no plano bi-dimensional há uma variedade de técnicas de transformação com propriedades similares. Os autores citam como podem ser feitas estas transformações [KGT99].

A indexação de objetos móveis foi estudada para uma e duas dimensões. Para uma dimensão são

dados alguns algoritmos de memória externa para o problema da indexação mas estes algoritmos não foram implementados. Para duas dimensões, foi considerado em um primeiro instante o caso onde objetos se movem mas com movimentos restritos a uma coleção de rotas em um terreno finito. Para este caso os autores afirmam que pode ser usado o mesmo algoritmo para uma dimensão. O problema de indexação em duas dimensões onde os objetos são permitidos se moverem para qualquer lugar em um terreno finito foi apresentado com algumas sugestões de soluções [KGT99].

Para resolver o problema de consultas em tempos futuros, os autores sugerem a construção de sua própria estrutura.

3.3 Indexando a posição de objetos em contínuo movimento

Saltenis, Jensen e Leutenegger, em 2000, [SJLL00] propuseram uma estrutura de indexação de objetos móveis baseada na utilização de árvores R^* para prover consultas sobre a posição corrente dos objetos e projetar posições futuras de tais objetos. Foi utilizada a estrutura de dados *Time-Parameterized R-Tree* (árvore *TPR*) que indexa as posições futura e corrente de objetos móveis através de funções lineares de previsão de movimento. A árvore *TPR* estende a árvore *R** [BKSS90] e baseia-se na árvore *R* [Gut84]. A implementação da árvore *TPR* faz uso da biblioteca Gist [HNP95, KMH97] para execução dos procedimentos, tais como inclusão e deleção, que podem ser executados na estrutura de árvore.

A árvore *TPR (time-parameterized R-tree)* [SJLL00] é uma extensão da árvore R^*, que pode responder a predições de consultas de objetos dinâmicos. Este índice tenciona fornecer respostas rápidas a consultas sobre a localização corrente e futura dos objetos móveis. Um objeto dinâmico é representado

com um retângulo envolvente mínimo *(MBR)*, que limita sua extensão no tempo corrente e em um vetor de posição e velocidade.

A árvore *TPR* não emprega replicação (cópia) de dados, pois tais replicações melhoram o desempenho das consultas, mas afetam o desempenho das atualizações. Na árvore *TPR*, cada Nó carrega um *retângulo envolvente mínimo* que contém pontos e os retângulos mínimos dos Nós descendentes. Estes retângulos limites são em função do tempo; com isso, eles são capazes de cobrir continuamente pontos ou outros retângulos que se movam. Um problema surge quando estes retângulos, com o passar do tempo, tornam-se muito grandes, pois as consultas perdem precisão e, com isso, uma nova reestruturação da árvore, com redimensionamento dos retângulos, se faz necessária.

A figura 3.1 mostra um exemplo do comportamento dos retângulos mínimos em um intervalo de tempo. O primeiro diagrama mostra a posição e a velocidade de sete objetos pontuais no tempo 0. O diagrama 2 mostra uma possibilidade de associação dos objetos no retângulo mínimo admitindo o número máximo de objetos por Nó. O diagrama 3 mostra a localização dos objetos e os retângulos mínimos no tempo 3 presumindo-se que eles cresceram e permaneceram válidos. O crescimento dos retângulos adversamente afeta a performance das consultas. Com o passar do tempo, os retângulos continuarão crescendo, chegando inclusive a uma deteriorização. O diagrama 4 mostra uma melhor associação dos objetos no retângulo. Veja [SJLL00] para detalhes de como esta nova associação de objetos no retângulo pode ser feita.

Dois valores de parâmetros afetam a indexação e a qualidade da árvore *TPR*: até que tempo as consultas em janelas permanecerão válidas no futuro $(t_{ref} + w)$? Qual a duração do intervalo de tempo que um índice, ou seja, que a estrutura de árvore poderá ser usada para consultas desde sua última

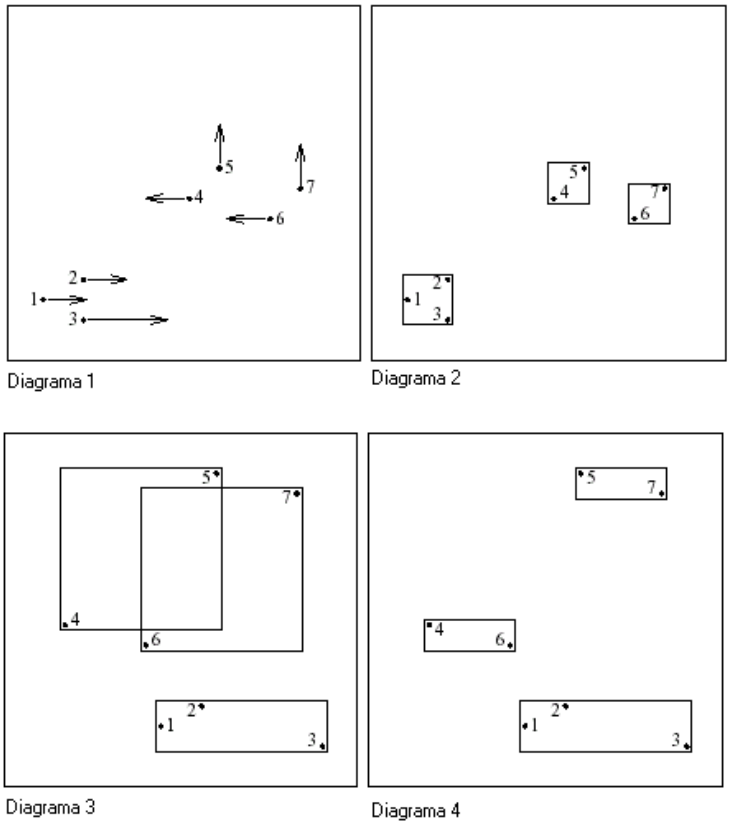


Figura 3.1: Objetos e retângulos minímos em movimento.

reestruturação $(t_{res} + u)$? Estes e outros parâmetros utilizados na indexação são definidos pelo usuário e servem para garantir que a estrutura de indexação forneça informações precisas, quando as consultas são executadas.

A árvore *TPR* indexa dados em seu espaço d-dimensional; isto é possível devido à parametrização da estrutura de índices, usando vetores velocidade e, dessa forma, habilitando o índice para ser visto em tempos futuros. Quando a indexação de objetos é feita no seu espaço nativo, um índice baseado em particionamento de dados parece ser mais aceitável.

A posição de um objeto $x(t) = (x_1(t), ..., x_d(t))$, onde $d \in \{1, 2, 3\}$, em função do tempo t pode

ser estimada por uma função linear $x(t) = x_0 + v_0(t - t_{ref})$, onde é admitido que o tempo t não é inferior ao tempo corrente t_{ref} e, x_0 e v_0 são respectivamente a posição e velocidade no instante t_{ref}. Usando as estimativas de posição dos objetos definidas anteriormente, a freqüência das atualizações de seus atributos é diminuída, e consultas futuras $(t > t_{ref})$ tornam-se possíveis.

Consultas

Define-se R, R_1 e R_2 retângulos d-dimensionais onde $d \in \{1, 2, 3\}$ e t, t_1, t_2 valores de tempo que não são menores do que o tempo corrente. As consultas atendidas neste trabalho são apresentadas a seguir e mostradas graficamente na figura 3.2.

- **Consulta tipo 1:** $Q_1 = (R, t)$ - especifica um retângulo R localizado num tempo t e retorna os pontos localizados neste retângulo no instante t.
- **Consulta tipo 2:** $Q_2 = (R, t_1, t_2)$ especifica um retângulo R que cobre o intervalo $[t_1, t_2]$. Esta consulta tem, como resultado, todos os pontos móveis que cruzam o retângulo $(d + 1)$-dimensional $[a_1, b_1] \times \cdots \times [a_n, b_n] \times [t_1, t_2]$ num instante entre t_1 e t_2 inclusive.
- **Consulta tipo 3:** $Q_3 = (R_1, R_2, t_1, t_2)$ especifica o trapézio $(d + 1)$-dimensional obtido pela conexão de R_1 no tempo t_1 a R_2 no tempo t_2. Retorna a pontos que cruzam o trapézio especificado.

A figura 3.2 mostra os tipos de consultas definidos. Os objetos O_1, O_2 e O_3 são objetos cujas trajetórias de movimento estão representadas por segmentos de reta. Neste exemplo, Q_0 = $([-5, 5], 0.5)$, Q_1 = $([10, 30], 3.75, 5.5)$ e Q_3 = $([-20, -10], [-30, -15], 3.75, 5.5)$. As consultas Q_0 e Q_1 são do tipo 1, Q_2 é uma consulta do tipo 2 e Q_3 uma consulta do tipo 3. Considerando T_Q o tempo quando a consulta Q é emitida, a resposta para Q_0 é o objeto o_1 se $T_Q < 1$. A resposta para Q_1 é o objeto o_1 se $T_{Q1} < 1$ e

se $T_{Q1} \geq 1$ nenhum objeto responderá a consulta. Se $T_{Q2} < 1$ a resposta para Q_2 é vazia; e se $1 < T_{Q2} \leq 2$, a resposta contém o objeto o_1, mas não o objeto o_2 pois sua existência não é conhecida até o tempo 2. Se $2 \leq T_{Q2}$, a resposta contém ambos objetos o_1 e o_2. Finalmente, a resposta para Q_3 é o objeto o_4, considerando os valores do T_{Q3}.

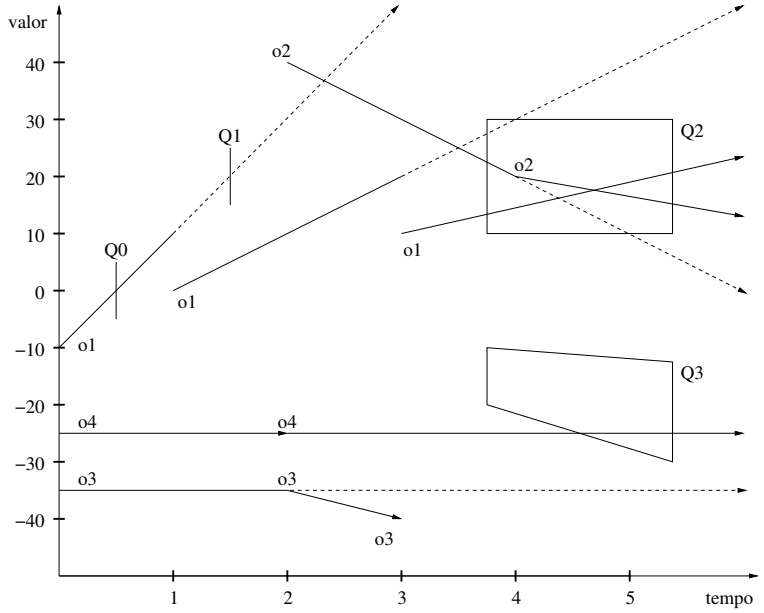


Figura 3.2: Tipos de consultas

O algoritmo de inserção da árvore *R** [BKSS90] usa funções para calcular a área, margem e cobertura dos retângulos. O algoritmo de inserção da árvore *TPR* é similar ao da árvore *R** com uma exceção: ao invés de funções, ele emprega a fórmula integral destas funções obtendo os valores das funções que cruzam o tempo quando a árvore é consultada.

No algoritmo de *split* (divisão) da árvore *R** é selecionada uma distribuição de entradas entre dois nós de um conjunto de distribuições candidatas, que são geradas, baseadas em um sorteio de posições de pontos em cada eixo coordenado. Na árvore *TPR*, no algoritmo de *split*, posições de pontos móveis ou

retângulos em diferentes pontos de tempo são usadas quando sorteadas. No tempo de carga, as posições t_1 dos retângulos limites são usadas, e no tempo de alteração, as posições dos retângulos no tempo corrente são usadas. As remoções na árvore *TPR* são executadas como na árvore R^*. Os retângulos lidos durante as operações de atualização são *"apertados"* com o objetivo de melhorar o desempenho das consultas sem afetar tanto o desempenho destas atualizações.

Esta proposta trata apenas da localização de objetos através de função de previsão de movimento linear, o que não garante um bom desempenho quando a trajetória real dos objetos é não linear. Outros tipos de consultas, tais como consultas a vizinhos próximos, não são considerados neste trabalho.

3.4 Indexando Pontos Móveis

Também em 2000, **Agarwal, Arge e Erickson** [AAE00] propuseram outra forma de indexação onde se presume que cada ponto move-se em uma trajetória linear, com velocidade fixa, dada por $s(t) = s_0 + v_0t$. A trajetória s dos pontos pode mudar a qualquer tempo. Admite-se que os objetos são responsáveis por alterar os valores de s_0 e da velocidade v_0, e que o sistema de Banco de Dados é notificado quando alguns destes valores mudam. Alguns esquemas para a indexação de pontos móveis em um plano são propostos mas o autor neste trabalho não entrou em detalhes. Esta proposta foi feita para responder a consultas utilizando S como o conjunto de pontos móveis no plano $\mathbb{R}^2$:

Q_1 - dado um retângulo R alinhado em um eixo, num plano xy e um tempo t_q, relacionar todos os pontos de S que estão situados dentro de R no tempo t_q.

Q_2 - dado um retângulo R e dois tempos T_1 e T_2, relacionar todos os pontos de S que estão situados

em R em qualquer tempo entre T_1 e T_2.

Q_3 - dado um ponto $\sigma \in \mathbb{R}^2$ e um tempo T_q, relacionar quais os vizinhos mais próximos de σ em S no tempo T_q.

O primeiro esquema de indexação proposto é baseado em partições de árvores que consideram o tempo como um terceiro eixo e resolve o problema no espaço xyt. Um ponto pode ser inserido ou removido e o índice não mudará a menos que a trajetória dos pontos mude e assim, este índice é dito *time-oblivious*. Neste modelo de indexação, consultas do tipo Q_1 podem ser respondidads em qualquer ordem de tempo, mas segundo os autores, têm custo computacional elevado.

Uma segunda abordagem, chamada *árvore cinética de intervalo*, baseada em estruturas de dados cinéticas, considera pontos no plano *xy* e o esquema de indexação envolve um tempo extra. A estrutura é modificada quando certos eventos cinéticos ocorrem, por exemplo, quando as coordenadas x e y de dois pontos tornam-se iguais. Esta estrutura é baseada em uma estrutura de Dados de memória interna com tempo de consulta ótimo verificado por Basch [BGZ97] e uma árvore de intervalo externa desenvolvida por Lars Arge [ASV99]. A idéia principal é armazenar somente um conjunto dos pontos em movimento em qualquer tempo. A qualquer hora, a ordem do sorteio dos pontos muda. Uma atualização cinética na estrutura de dados foi desenvolvida para fazer estas mudanças.

As consultas feitas a estas estruturas devem ser executadas em ordem cronológica. Uma vez que um evento cinético tenha mudado a estrutura de dados, nenhuma consulta pode referenciar pontos em um tempo antes do evento. Consultas não cronológicas foram tratadas usando técnicas de persistência parcial. Foi mostrado também, como combinar árvores de intervalo cinéticas com partições de árvores, para que houvesse uma relação entre o número de eventos cinéticos e o desempenho de consultas.

Um outro esquema de indexação chamado *time-responsive* combina as vantagens de ambos os esquemas definidos. Este esquema poderá responder a consultas em qualquer ordem de tempo e o acesso a disco será pequeno se o tempo da consulta for próximo do tempo corrente. A idéia é que como os objetos se movem continuamente, alterações cinéticas são executados na estrutura de dados quando certos eventos ocorrem e assim não é necessário alterar a estrutura continuamente. Com isso, eventos futuros são agendados para atualizar a estrutura de dados. Não foi relatado que eventos foram usados e como as alterações na estrutura foram executadas.

A indexação de pontos que se movem em trajetória não linear e consultas retornando vizinhos exatos de um ponto são colocados como trabalhos futuros para os autores mas até a presente data não foi encontrado nenhum material destes autores que tratasse sobre estes assuntos.

3.5 Indexando a Posição Corrente de Objetos Móveis utilizando a Árvore *R lazy update*

Kwon *et al.* propuseram uma estrutura de indexação para objetos em contínuo movimento chamada *Lazy Update R-tree (árvore LUR)* [DK02]. A árvore *LUR* é baseada na árvore *R*. Os algoritmos e estruturas de índices são semelhantes aos usados em outras variantes da árvore *R*. Somente o algoritmo de alteração é modificado. Nesta técnica, a estrutura de índice é alterada quando um objeto se move para fora de seu retângulo mínimo. Se, após o movimento do objeto, sua nova posição continuar no mesmo retângulo mínimo, a árvore *LUR* muda somente a posição do objeto no Nó folha. Uma requisição de alteração de posição de um Nó é solicitada através do par *(oid,* P_{new}*)* onde *oid* é a identificação do objeto e P_{new}

a nova posição. Com isso, através de uma estrutura de índice secundária (directLink) é encontrada a folha que contém o objeto a ser alterado. A chave desta estrutura é o identificador do objeto *(oid)*. Cada entrada nesta estrutura secundária tem um ponteiro para uma entrada correspondente no Nó folha da estrutura de árvore.

Assim, no algoritmo de alteração, o primeiro passo é encontrar a folha na qual um objeto se encontra usando o *directLink*. O Nó folha é lido e é encontrada uma entrada cujo identificador do objeto é igual ao identificador dado. Se a nova posição do objeto está no mesmo retângulo mínimo, somente a posição do objeto na entrada é modificada. Se a nova posição não está no retângulo mínimo de origem do objeto, a árvore poderá ser alterada de três formas:

1) Apagando e inserindo o objeto modificado. Este método é o mesmo usado na árvore *R* padrão e pode causar os mesmo problemas de divisões e agrupamentos de Nós.

2) Extensão do retângulo mínimo. Ao invés de uma remoção da posição antiga, o retângulo mínimo é estendido para incluir a nova posição, e os retângulos mínimos, dos Nós acima, deverão ser ajustados. Esse método só será apropriado se a nova posição do objeto não ficar tão longe do retângulo mínimo pois, caso contrário, o retângulo crescerá muito e o desempenho das consultas será degradado.

3) Reinserir dentro do Nó pai. Depois de apagar a posição antiga, o objeto com a nova posição é inserido dentro do Nó pai, ao invés de inserir na folha. Se o Nó pai estiver cheio, deve ser tentada a reinserção no Nó pai do pai. Para que esta técnica seja utilizada, cada Nó deverá guardar um ponteiro para o Nó pai, diminuindo o *fanout* da árvore. A proposta diz que faz uso apenas da primeira abordagem no algoritmo de alteração, continuando, assim, com os mesmos problemas de divisão e agrupamento das estruturas baseadas em árvores *R*.

A abordagem de extensões do retângulo mínimo consiste em aumentar o limite do retângulo mínimo, para que ele contenha a nova posição do objeto modificado. Estas extensões são usadas somente nos Nós folhas. Se o aumento do retângulo mínimo for muito grande, o desempenho nas consultas pode ser degradado se a sobreposição entre os Nós folhas for aumentada. O trabalho diz que há uma relação com ganho entre o desempenho de alterações e a perda de desempenho das consultas, mas esta relação não foi mostrada explicitamente, além de ferir um critério básico de construção da árvore *R*: ter sempre retângulos envolventes mínimos. Outro ponto a considerar é que os experimentos foram feitos utilizando poucos pontos indexados, deixando dúvidas no comportamento da estrutura quando aplicada com uma quantidade muito maior de pontos.

3.6 Eficiente Indexação de Objetos Espaço-Temporal

Este trabalho [HKTG02] está interessado na otimização de consultas do tipo "*encontrar os objetos que surgem em uma área* S *em um tempo* t"isto é, o foco está no que acontece em um dado instante t. Um objeto seria representado por um retângulo $3d$ cujo tamanho corresponderia ao intervalo de vida do objeto. Isto introduz a noção de muito espaço vazio pois objetos que permanecem sem mudanças por um longo período deverão ter longo tempo de vida e assim, serão armazenados em grandes retângulos. O retângulo do tempo de vida determina o tamanho do intervalo de tempo associado com o índice no qual ele reside. Isto implicaria em uma perda de espaço e grandes sobreposições de retângulos.

Assim, este trabalho está visa melhorar o desempenho de consultas, através da redução do espaço vazio obtido pela aproximação de objetos espaço-temporal com seus retângulos envolventes mínimos. Uma solução proposta é introduzir divisões *(splits)* artificiais: o tempo de vida de um objeto longo é

dividido em pequenos pedaços consecutivos desta forma, um objeto emitido no tempo t artificialmente é removido e surge no mesmo intervalo de tempo. O objeto original será representado por dois objetos, um com tempo de vida que termina em t e outro com tempo de vida que começa em t.

No caso geral objetos podem mover-se em qualquer direção. A utilização de tuplas com funções polinomiais de grau alto aproxima melhor o movimento dos objetos segundo os autores. Uma tuplo T pode ter o seguinte formato: $(t_i, t_e, d, c_d, ..., c_o)$ onde t_s é o tempo inicial, t_e o tempo final, *d* o grau da função e c_d, ... c_o são os coeficientes. Para o movimento em duas dimensões cada tupla deverá conter duas funções, a primeira representa o movimento no eixo x e a segunda no eixo y. O resultado da combinação das duas funções fornece a trajetória de um objeto.

Considerando que um objeto será permitido mover-se usando funções de combinação polinomial o trabalho apresentou um algoritmo e uma heurística para decidir como aplicar as divisões em um objeto com a finalidade de reduzir os espaços vazios dos retângulos envolventes mínimos e um outro algoritmo para distribuir otimamente um número total de divisões em uma coleção de objetos foi apresentada.

No primeiro algoritmo é dado um objeto e um número de divisões e é desejado encontrar como pode ser feita a divisão no objeto para obter o máximo de ganho em espaços vazios. No segundo, dado uma coleção de objetos e um número de divisões, é tentado distribuir as divisões entre todos os objetos de forma que melhore o desempenho das consultas. Este número de divisões é pesquisado para que possa ser escolhido um bom número.

Os objetos foram indexados usando as estruturas de árvores $3DR^*$ e a *PPR (Partial persistent R-Tree)* [KTF98]. A árvore *PPR* obteve ganho superior em todos os testes de consultas executados. Nos resultados obtidos, foram consideradas pequenas taxas de alterações, deixando o problema da indexação

em um ambiente com altas taxas de alterações como trabalhos futuros.

3.7 Indexação de Objetos Móveis para serviços baseados em localização

O movimento de objetos em contínuo movimento é proposto para ser representado por funções lineares do tempo. Com isso, as alterações de seus posicionamentos são necessárias somente quando os parâmetros da função mudam significativamente. Independente de como as posições dos objetos são representadas, a precisão das posições e a utilidade dos serviços, baseados nesta localização, diminuem com o passar do tempo. Um tempo de expiração deverá ser, então, associado a cada objeto para que estes possam ser descartados. Esta é a idéia básica da árvore R^{EXP} [SJ02], uma árvore baseada na árvore R^* e que faz uso dos métodos de construção idealizados pela árvore TPR [SJLL00].

A árvore R^{EXP} indexa as posições corrente e futura dos objetos em movimento, presumindo-se que suas posições expiram depois de um período de tempo especificado. Encontrar o tempo certo de expiração pode depender de uma série de fatores e tipo de movimento de objetos como por exemplo: objetos podem mover-se de acordo com rotas pré-determinadas e escalonadas como em um sistema de transporte público. A escolha correta destes fatores pode tornar difícil a escolha do tempo de expiração ideal.

A posição do movimento do objeto é representada por uma posição de referência, um vetor velocidade e um tempo de expiração *(x, v, t_{exp})*. Com a remoção das entradas expiradas, o recálculo dos retângulos mínimos fazem-se necessários quando os nós atingem o valor mínimo permitido por folha menos um. Então a árvore R^{EXP} deverá ser reorganizada para que as consultas não percam precisão.

Os experimentos foram realizados e comparados com a árvore *TPR* e com uma árvore *TPR* associada com uma árvore *B* para armazenar escalonamento de remoções. O desempenho da árvore R^{EXP} não foi superior a abordagem que emprega escalonamento de remoções. Esta diferença foi atribuída a uma melhor organização do índice que resulta de muitas alterações quando remoções escalonadas são adiconadas para regular operações de atualização.

A abordagem deste trabalho preocupou-se em eliminar da estrutura de indexação proposta, objetos que tenham tempo de vida expirado para que eles não façam parte do resultado de consultas. Contudo, foi observado pequeno ganho de desempenho com a utlização de um tempo de expiração para objetos.

3.8 Árvore *Star*: Um Índice com Auto-Ajuste para Objetos Móveis

Em 2003, **Procopiuc, Agarwal e Har-Peled** [PAhP03] desenvolveram a *Spatio Temporal Self Adjusting R-Tree (árvore STAR)*. Esta é uma técnica que, segundo os autores, estende naturalmente a árvore *R**[BKSS90]. A forma desta extensão não é citada pelos autores. Eles propuseram que, ao invés de reconstruções ou atualizações periódicas dos índices, sejam mantidas algumas informações auxiliares, como descrito à frente e, assim, a estrutura é capaz de se auto-ajustar para que consultas ao posicionamento dos objetos, em tempo futuro, possam ser respondidas da melhor forma possível.

Nos nós folhas desta estrutura são armazenados os pontos ou mais exatamente, se o movimento de um ponto p_i é definido por uma função linear, $p_i(t) = a_i + b_i t$, os coeficientes a_i e b_i são armazenados nas folhas. Em cada nó interno, é armazenado um ponteiro para cada um de seus filhos e uma representação da parametrização do retângulo mínimo (B) que contém todos os pontos descendentes. Assim, se um

retângulo envolvente mínimo (REM) contém um conjunto de pontos, para cada tempo t maior que o tempo atual, o retângulo mínimo (B) conterá os retângulos envolventes mínimos (REM). A qualidade da aproximação de B para REM é controlada pelo usuário por um fator de aproximação x. Quanto maior esta aproximação, maior o espaço necessitado para armazenar B. Para maiores detalhes veja a seção 3.1 deste artigo.

Eventos internos e externos podem causar alterações da estrutura de índice da árvore *STAR*. São considerados eventos externos, as inserções/remoções do ponto e a mudança na trajetória do ponto. Um evento interno, chamado *eventos box*, dirá quanto tempo o retângulo envolvente mínimo permanecerá válido. Outro evento interno chamado *conflito de evento* ocorre quando, em um nó da árvore, os retângulos mínimos de seus filhos estão muitos sobrepostos. Se *eventos box* e *conflito de evento* ocorrem simultaneamente, o *eventos box* tem prioridade para determinar a alteração.

Para a realização de testes, foram considerados pontos movendo-se no plano e uma função linear para a previsão do movimento destes objetos. Com os experimentos realizados pelos autores na estrutura de árvore *STAR*, comparações com outra estrutura, a árvore *TPR*, foram feitas e segundo os autores, a árvore *STAR* não necessita de estimativa de tempos, utilizados pela árvore *TPR* e mostrado na seção 3.3 de nosso trabalho, para saber até quando os parâmetros (validade de janela e de índice) permanecerão válidos, garantindo o mesmo desempenho das consultas, à árvore *STAR*, para vários ajustes no tempo dos intervalos de atualização.

Assim como a árvore *TPR*, a árvore *STAR* faz previsão de movimentos usando apenas função linear. A árvore *STAR* é mais aceita para cargas de trabalho com poucas alterações [SJ02].

3.9 Considerações Finais

O estudo das propostas apresentadas possibilitou o conhecimento das diversas técnicas de indexação existentes até o momento permitindo que fizéssemos uma análise de suas características com alguns pontos positivos e negativos.

Para redução do número de operações de atualização, todas as abordagens com exceção de [HKTG02] usam uma função linear para expressar o movimento de objetos. Nestas abordagens, o Banco de Dados é alterado somente quando os parâmetros da função mudam. Todavia, se os movimentos dos objetos têm trajetórias complexas, uma simples função linear não é muito aceitável para descrever estes movimentos. Assim, estas técnicas podem ser usadas para reduzir o número de alterações, mas não fornecem precisão na localização dos dados e isto não é apropriado em aplicações que requerem uma boa precisão de localização.

Outras propostas de indexação para objetos móveis também podem ser encontradas nos trabalhos de Agarwal *et al*. [AAV01] e Saltenis [SJ99]. Estratégias de consultas a objetos móveis podem ser encontradas em Prassad Sistla [SWCD98, SWCD97], Vinit [JS94], Pfoser [PJ01] e Mokhtar [MSI02], e técnicas de manipulação de Bancos de Dados para objetos móveis podem ser encontradas em Wolfon [WXCJ98, WCD$^+$98, Wol02] e Forlizzi [FGNS00]. Estas propostas servem para mostrar também que pesquisas em objetos móveis são constantes e que continua-se a busca por soluções que tenham melhor desempenho em se tratando de localização de objetos móveis.

Capítulo 4

Indexação de Objetos Móveis através da árvore *TPKd*

4.1 Introdução

Neste capítulo será descrita a estrutura de árvore TPKd, proposta para indexar objetos móveis e seus algoritmos. Esta estrutura utiliza como base árvores Kd. Uma definição do problema de indexação de objetos móveis é também apresentada.

Aplicações que fazem uso da *localização de objetos espaciais* como atributo de pesquisa têm surgido com muita intensidade com o avanço na área da comunicação sem fio. Manter este atributo sempre atualizado, através de alterações explícitas, pode implicar em uma degradação do Banco de Dados. A garantia de um acesso eficiente aos dados, através deste atributo, pode ser mantida mediante

técnicas de indexação eficientes.

O objetivo deste trabalho é apresentar uma estrutura de indexação que permita consultas sobre posicionamento futuro de objetos serem respondidas de forma rápida e eficiente. Para aumentar a precisão no posicionamento e, conseqüentemente no resultado das consultas, funções quadráticas de previsão de movimento são usadas para estimar a localização, em um tempo futuro, dos objetos.

A posição de um objeto $x(t) = (x_1(t), ..., x_d(t))$, onde $d \in \{1, 2, 3\}$, em função do tempo t, pode ser estimada por uma função linear $x(t) = x_0 + v_0(t - t_{ref})$ ou, ainda, por uma função quadrática $x(t) = x_0 + v_0(t - t_{ref}) + a_0(t - t_{ref})^2/2$, onde se presume que o tempo t não é inferior ao tempo corrente t_{ref} e x_0, v_0 e a_0 são, respectivamente, a posição, velocidade e aceleração no instante t_{ref}. Usando as estimativas de posição dos objetos definidas anteriormente, a freqüência das atualizações de seus atributos é diminuída; e consultas futuras $(t > t_{ref})$ tornam-se possíveis. A escolha da freqüência de atualizações depende do tipo de movimento, da precisão desejada e das limitações técnicas [SWCD98]. Como a estimativa quadrática faz uso da aceleração como informação adicional, ela permite uma melhor aproximação da trajetória original do objeto.

A indexação junto com a função de previsão de posicionamento dos objetos permite a definição de uma variedade de consultas baseadas na localização destes objetos. Algumas destas consultas são baseadas em regiões onde se deseja que sejam encontrados objetos (pontos) situados em determinadas áreas, as quais são fixas ou variam no tempo. Sejam R, R_1 e R_2 retângulos d-dimensionais onde $d \in \{1, 2, 3\}$ e t, t_1, t_2 valores de tempo que não são menores do que o tempo corrente (t_{ref}), tem-se:

- **Consulta tipo 1:** $Q_1 = (R, t)$ - especifica um retângulo *R*, localizado num tempo t, e retorna os objetos (pontos) localizados neste retângulo, no instante t.

- **Consulta tipo 2:** $Q_2 = (R, t_1, t_2)$ - especifica um retângulo R, que cobre o intervalo $[t_1, t_2]$. Esta consulta retorna os objetos (pontos) com trajetória em $(x(t), t)$, cruzando o hiper-retângulo $(d+1)$-dimensional $[a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_n, b_n] \times [t_1, t_2]$.

- **Consulta tipo 3:** $Q_3 = (R_1, R_2, t_1, t_2)$ - especifica o trapézio $(d+1)$-dimensional, obtido pela conexão de R_1 no tempo t_1 a R_2 no tempo t_2. Retorna todos os objetos (pontos) que cruzam o trapézio especificado.

4.2 Critérios de Balanceamento que podem ser usados em Árvores

Andersson [And99] mostrou que novos métodos para a manutenção de dados em árvores de pesquisa tinham sido desenvolvidos e que a atenção principal estava focada nas árvores com altura limitada ou árvores balanceadas. O motivo para estas pesquisas é que o tempo de acesso, no pior caso, seria proporcional à altura da árvore.

Uma grande variedade de critérios de balanceamento já existe. Exemplifiquemos:

1. árvores *AVL*, introduzidas por Adelson-Veslkil e Lanis [AVL62], usam, como critério de balanceamento, a idéia de que a diferença de altura de duas subárvores seja de, no máximo, 1. Árvores *AVL* têm altura máxima de $1.44 \log |P|$ onde $|P|$ define o número de folhas (peso) da árvore *P*.

2. As arestas em uma árvore *SBB* [Bay72] são de dois tipos: horizontal e vertical. Duas arestas adjacentes nunca são, em conjunto, horizontal; e o número de arestas vertical, no caminho da raiz até a folha, é o mesmo para todos os níveis. Este critério garante uma altura máxima de $2 \log |P|$.

Uma questão natural surge do estudo de árvores balanceadas: é realmente necessário fazer um desvio sobre o critério de balanceamento onde o que interessa é a altura logarítmica ?

Andersson mostra, em seu trabalho, um critério de balanceamento que usa uma relação entre o tamanho e a máxima altura da árvore. Isto é feito usando-se uma reconstrução parcial da subárvore desbalanceada; este tipo de reconstrução é atrativo para estruturas de árvores de pesquisas binárias e até para mais complicadas estruturas de dados, tais como árvores de pesquisa multidimensional [Ben75].

A idéia principal é deixar que uma árvore fique com qualquer forma desde que sua altura não exceda $\lceil c \log |P| \rceil$ para alguma constante $c > 1$. Quando este critério é violado, a altura da árvore pode ser diminuída pela reconstrução parcial com um baixo custo de amortização.

Quando uma remoção é feita, a altura da árvore não aumenta; então, remoções são feitas sem que seja necessária uma reconstrução, até que o número de remoções ultrapasse um limite estabelecido. Quando isto acontece, uma reconstrução da árvore se faz necessária.

A forma de balanceamento proposta é feita executando-se rotações e divisões *(splits)* na subárvore desbalanceada até que seja obtida uma nova subárvore balanceada. Os critérios usados para determinar que haja rebalanceamento atendem a estruturas de árvores multidimensionais; mas a forma como foi proposto o balanceamento (rotações de árvores), não se aplica bem a estruturas baseadas em árvores *Kd* devido à utilização de mais de uma chave de busca em diversos níveis da árvore.

Depois de estudos sobre os critérios de balanceamento existentes, definimos o nosso critério de balanceamento, utilizado no algoritmo de inserção pela árvore *TPKd*. Este critério será apresentado na seção 4.3.1.

4.3 A Árvore TPKd

A *árvore Parametrizada pelo Tempo K-dimensional* (árvore TPKd) é uma árvore binária onde os Nós carregam informações de posição, velocidade, aceleração e identidade de objetos móveis. O discriminante associado a cada Nó *(DISC)* serve para indicar a divisão pela qual os Nós são inseridos à direita ou à esquerda. Esta divisão é baseada nos valores de posicionamento. A estrutura proposta, indexa eficientemente as posições atuais e futuras de objetos em contínuo deslocamento. Esta estrutura permite indexação de pontos e regiões, as quais podem ser indexadas através do armazenamento na árvore de seus centróides.

A árvore *TPKd* armazena as informações preliminares dos pontos nos Nós internos e nas folhas, sendo que cada Nó armazena um único ponto. Com isso, buscas podem ser respondidas com maior rapidez, tendo em vista que pode não ser necessário percorrer a árvore até sua profundidade máxima para encontrar um objeto procurado. Esta é uma diferença essencial em relação a árvore *Kd*.

O armazenamento de um único dado por folha em uma estrutura de árvore binária pode resultar em uma estrutura com grande profundidade. Embora isto possa acontecer, as buscas são executadas de maneira eficiente, pois a divisão do espaço, baseada em critérios que utilizam os valores armazenados nos Nós, como chaves para buscas, fazem com que a procura por pontos na árvore tenha um caminho único a ser seguido.

Algumas estruturas de árvore, como a árvore *R*, armazenam seus dados no último nível da árvore, ou seja, nas folhas. Como existe uma quantidade pré-definida de pontos por Nó, se um Nó “encher”, este terá que ser dividido *(split)* e esta divisão pode ser propagada para outros Nós. Na árvore *TPKd*,

estas divisões não acontecem, pois os dados são armazenados também em Nós internos, o que significa um custo computacional a menos para a estrutura.

A árvore *TPKd* é uma estrutura de memória principal, característica que provê um melhor desempenho para consultas, tendo em vista que, quando uma consulta é executada, informações preliminares sobre os pontos que estão sendo pesquisados são procuradas diretamente na árvore, não necessitando acesso adicional a disco, neste momento da busca.

Um dos maiores problemas da árvore *Kd* é manter a árvore balanceada após inserções e remoções. Isto é um procedimento que tem alto custo computacional. Por este motivo, resolvemos utilizar, em nossa estrutura de indexação, critérios para indicar a necessidade de balanceamentos. Estes balanceamentos são ditos parciais por serem aplicados somente nas subárvores que se encontram desbalanceadas. Uma vantagem na escolha de uma estrutura de indexação baseada na árvore *Kd* está também no fato de que balanceamentos, neste tipo de estrutura, são executados em menor tempo de que as estruturas baseadas em árvore *R*, tendo em vista que não são necessárias reconstruções de retângulos mínimos. O critério de balanceamento utilizado neste trabalho será discutido na seção 4.3.1, onde são descritos os procedimentos necessários para a inserção de um Nó na árvore.

Com o passar do tempo, os objetos representados na árvore necessitam ter suas posições atualizadas para que as consultas tenham sempre resultados precisos. Isto é feito na nossa estrutura de indexação através do conceito de *auto ajuste* onde, através de critérios que serão melhor descritos à frente, a árvore é totalmente reconstruída, ou seja, a posição de todos os objetos (pontos) é atualizada, e eles são novamente incluídos na árvore, montando uma nova árvore com a posição atualizada de todos os objetos. Serão descritos, a seguir, os algoritmos utilizados nesta nova estrutura.

4.3.1 Inserção

O algoritmo de inserção da árvore TPKd deste trabalho usa o algoritmo padrão proposto por Bentley [Ben75]. Contudo, propomos uma extensão a este algoritmo, introduzindo um critério de balanceamento [And99] que permitirá a árvore manter-se balanceada após inúmeras inserções. Este balanceamento é dito parcial, pois é executado somente para as subárvores que encontrarem-se desbalanceada após inserções.

Antes de definir o critério de balanceamento, explicaremos alguma variáveis usadas nele: a altura da árvore é dada como sendo a quantidade de níveis que a árvore possui; peso da árvore significa a quantidade de Nós na árvore mais 1. O critério de balanceamento é estabelecido como sendo uma relação entre a altura e o peso da árvore dado por $h(P) \leq \lceil c \log w(P) \rceil$ onde c é uma constante que seja maior que 1, $h(P)$ a altura e $w(P)$ o peso da árvore P.

Quando durante a inserção de um Nó n este critério é violado $(h(P) > \lceil c \log w(P) \rceil)$, um balanceamento parcial da subárvore que contém este Nó *n* é descrito como segue: o caminho da árvore desde o Nó *n* inserido até a raiz é percorrido *bottom-up* ou seja, de baixo para cima, buscando o primeiro Nó que não satisfaz o critério de balanceamento $(h(n) > \lceil c \log w(n) \rceil)$. Encontrado este Nó, a sua subárvore correspondente é rebalanceada com o objetivo de manter a profundidade da árvore uniforme em todos os ramos para que não haja degradação de performace nas consultas.

Algoritmo de Inserção

Insere um novo registro E na árvore TPKd.

Passo 1. Pesquisa Subárvores. Se X, Nó na árvore, não é folha, testar a posição do Nó X no discrimi-

nante da vez:

- Se a posição.E[disc] $<$ posição.X[disc] ou seja, se a posição do registro E, que será inserido, é menor do que a posição de X, no discriminante da vez, uma chamada recursiva da função ***inserir*** é feita, passando-se o registro E, o Nó da esquerda e o próximo discriminante.
- Um procedimento análogo é feito quando a posição do novo registro E é maior ou igual à posição do Nó *X* (posição.E[disc] $\geq$ posição.X[disc]); a chamada recursiva é feita passando-se o registro E, o Nó da direita e o próximo discriminante.

Passo 2. Se X é um Nó folha, os seguintes procedimentos são feitos:

- a altura do Nó inserido é calculado para ser verificado o critério de balaceamento.
- O peso da árvore (desde a raiz) é atualizado.
- O critério de desbalanceamento $(h(E) > c \log w(E))$ é verificado. Se a árvore apresenta-se desbalanceada para o Nó *E* que está sendo inserido, ela se apresenta desbalanceada para a altura da árvore. Se houver desbalanceamento, dois procedimentos são chamados: o ***localizaNoBalanc***, que irá localizar o Nó que causou desbalanceamento, e o ***Balancear***, que irá fazer o balanceamento parcial da árvore. Estes procedimentos serão descritos mais à frente.

Antes de descrevermos os procedimentos ***localizaNoBalanc*** e ***Balancear***, é importante fazermos a seguinte observação: a constante usada para cálculo do desbalanceamento poderá ser qualquer valor maior que 1. Valores próximos a 1 indicarão que a altura da árvore será pequena, mas que muitos balanceamentos serão feitos na árvore. Nos nossos testes, utilizamos valores de constantes entre $(1.1$ e $1.35)$

e as árvores geradas nos diversos experimentos mostraram boa relação entre a altura e a quantidade de balanceamentos. Na avalização dos experimentos, seção 5.6, detalhamos um pouco mais sobre os resultados da utilização deste valor para a constante.

Algoritmo LocalizaNoBalanc

Este algoritmo é utilizado para localizar o primeiro Nó que causou desbalanceamento na árvore. A pesquisa de localização deste Nó, que causou o desbalanceamento, é feita de forma *bottow-up* no caminho que vai desde o último Nó folha, onde foi tentado inserir o Nó *E*, até a raiz.

Passo 1. Inserir em um vetor o endereço dos Nós no caminho que vai da raiz até o ultimo Nó inserido, que causou desbalanceamento.

Passo 2. Percorrer a subárvore onde o Nó *E* causou desbalanceamento de forma *bottom-up* ou seja, de baixo para cima, para encontrar, desde o Nó folha *E* inserido, que causou o desbalanceamento, o primeiro Nó, que não satisfaz à condição de balanceamento ($h(E) < c * logw(E)$) onde *h* é a altura do Nó na árvore e *w* seu peso. Quando um Nó não satisfizer esta condição, ele é retornado como resultado do algoritmo e será passado para o algoritmo ***Balancear*** chamado no algoritmo *Inserir*.

Algoritmo Balancear

Inserir todos os Nós da subárvore desbalanceada, cujo Nó raiz é o Nó retornado pelo algoritmo ***LocalizaNoBalanc***, em um vetor. O objetivo deste algoritmo é criar uma nova subárvore balanceada usando os Nós inseridos neste vetor.

Passo 1. Uma ordenação do vetor que contém todos os Nós da subárvore desbalanceada é feita, passando o começo e o fim das posições a serem ordenados e o discriminante pelo qual a ordenação será feita. Na primeira vez que esta ordenação é executada, o discriminante passado será o discriminante do Nó encontrado no algoritmo ***LocalizaNoBalanc***.

Passo 2. Escolher o elemento mediano do vetor para colocar na subárvore balanceada (o primeiro elemento escolhido será a nova raiz da subárvore rebalanceada).

Passo 3. Uma nova chamada dos passos 1 e 2 para cada metade do vetor, a partir da posição mediana, é feita, passando na chamada recursiva, o discriminante, de forma alternada, ou seja, se, na primeira chamada, foi passado *disc* igual 0, na próxima, será passado *disc* igual a 1 e assim sucessivamente. Este passo é executado até que uma quantidade mínima (dois) em cada metade do vetor seja encontrada.

4.3.2 Remoção

Remover Nós de uma árvore *TPKd* pode causar desbalanceamento da árvore, pois não há rebalanceamentos parciais na remoção. Há, certamente, uma compensação deste problema no momento em que um outro Nó qualquer for inserido em uma subárvore já desbalanceada, pois o algoritmo de inserção poderá fazer rebalanceamentos parciais na subárvore na qual o Nó está sendo inserido (se o critério de balanceamento for violado). Os Nós, quando removidos, são apagados da memória principal. A cada remoção, um critério é testado para verificar se há a necessidade de ser feita uma reconstrução total da árvore. O critério utilizado $(Ndel > R * (Qtdnos + 1))$ testa se o número de deleções feitas (Ndel) é maior do que uma constante *R*, multiplicada pela quantidade de Nós mais um. Quando o número de

remoções ultrapassar o valor de $R*(Qtdnos+1)$ indicará que muitas subárvores estarão desbalanceadas (devido às remoções feitas) e uma reconstrução total da árvore será executada. Nesta reconstrução, o posicionamento dos objetos é atualizado para que as consultas não percam precisão em suas respostas.

Algoritmo de Remoção

Remove um registro *E* na árvore *TPKd*. A pesquisa que indicará o Nó que irá substituir o registro *E* na árvore é iniciada pela subárvore da direita para que haja uma compensação da quantidade de Nós em ambos os lados da árvore pois, quando o algoritmo de inserção é executado, o registro a ser inserido, que tem valor de chave igual ao valor do Nó na árvore, é colocado na subárvore direita.

Passo 1. Se o Nó *E* a ser apagado é uma folha, a sua remoção é direta. A área reservada e ocupada pelo Nó *E* é liberada.

Passo 2. Se existir subárvore à direita do Nó E, ela é percorrida. Um algoritmo ***encontraNoMin*** é chamado para encontrar nesta subárvore o Nó *X* de chave mínima, para o discriminante de *E*.

Passo 3. Encontrado este Nó *X*, um algoritmo ***atualizaNoDel*** será chamado.

Na primeira chamada deste algoritmo, o Nó *X* é retirado de seu lugar e ocupará o lugar do Nó *E*, que será removido. Uma nova chamada a este algoritmo irá encontrar o novo Nó *Y*, que irá ficar no lugar do Nó *X* e este procedimento será chamado recursivamente até que seja encontrado um Nó folha. Depois de todas estas atualizações feitas, o Nó *E* é apagado da memória principal.

Passo 4. Caso a subárvore à direita do Nó *E* seja nula, um procedimento análogo é executado na subárvore esquerda onde será procurado um Nó de chave máxima para o discriminante de *E*.

Como, no momento de inserir, os Nós com valores de chaves iguais foram inseridos à direita, quando a procura por um valor de chave máxima à esquerda for feita, pode acontecer que mais de um Nó tenha o mesmo valor representando a chave máxima na subárvore à esquerda. Então, um deles será escolhido para ser a nova raiz da subárvore de *E* e os outros, serão inseridos na subárvore do Nó filho direito de *E*. Isto é feito para que a árvore permaneça com as características de uma árvore kd.

Passo 5. Encontrado este Nó *X*, o algoritmo ***atualizaNoDel*** será chamado. Na primeira chamada deste algoritmo, o Nó *X* é retirado de seu lugar e ocupará o lugar do Nó *E*, que será removido. Uma nova chamada a este algoritmo irá encontrar o novo Nó *Y*, que irá ficar no lugar do Nó *X* e este procedimento será chamado recursivamente até que seja encontrado um Nó folha. Depois de todas estas atualizações feitas, o Nó *E* é apagado da memória principal.

A figura 4.1 mostra um exemplo de como um Nó é apagado da árvore *TPKd*. Na figura *a*, o Nó *X* $(115, 80)$ é marcado para remoção. Executando o ***passo 2***, um Nó de chave mínima deverá ser encontrado, na mesma posição do discriminante do Nó *X*. Neste exemplo, esta posição é representada pelo eixo y. O Nó *Y* $(90, 95)$ é escolhido, substituirá o Nó *X*, que foi apagado, e as atualizações necessárias serão feitas como está descrito no ***passo 3***. Todos os procedimentos de movimentação dos Nós na árvore são feitos com a utilização de ponteiros. A figura *b* mostra como ficará a nova árvore após a remoção do Nó.

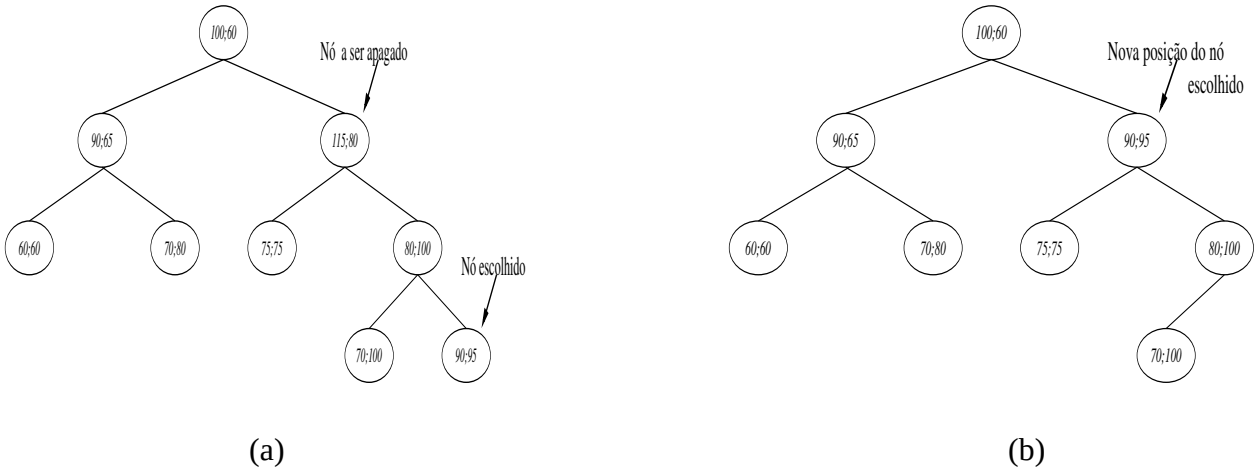


Figura 4.1: Exemplo de deleção de um Nó em uma árvore *TPKd*.

4.3.3 Consultas

Os Nós da estrutura de árvore TPKd carregam informações de posição, velocidade, aceleração, tempo e identificação de cada objeto que está no espaço mapeado pela árvore. O tempo de cada objeto indica o tempo da última inserção dele na árvore. Estas informações servirão para o cálculo da função quadrática de previsão de movimento do objeto, no momento em que uma consulta for realizada.

As consultas são feitas para procurar objetos em um espaço retangular definido. No primeiro momento, são encontrados pontos candidatos em uma região especificada por um aumento, estabelecido pelo usuário, como sendo um percentual da área do retângulo. O objetivo deste aumento é para que sejam incluídos, como candidatos, os pontos que estejam próximos da área do retângulo e, desta forma, aumentar o número de candidatos que possam satisfazer às condições da consulta. Em seguida, a função de previsão de movimento é aplicada e é verificado se os pontos candidatos realmente encontram-se dentro de retângulo original.

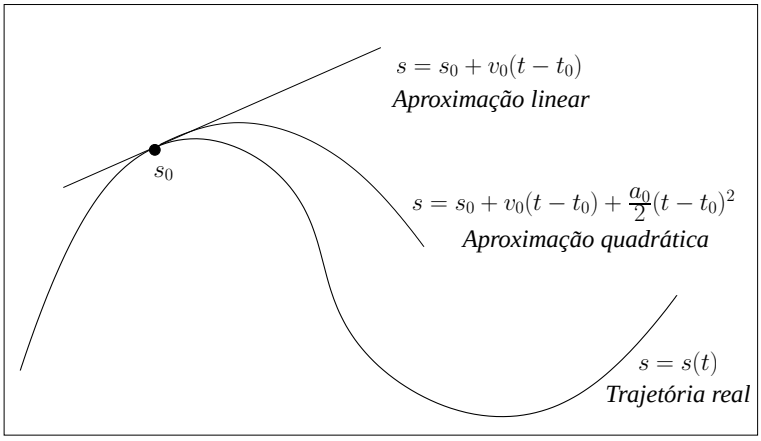


Figura 4.2: Funções linear e quadrática de previsão do movimento.

Nossa função de previsão apresenta uma melhor aproximação da localização do objeto, se for comparada com uma função linear de previsão proposta por Saltenis *et al.* em [SJLL00]. Para ilustrar esta assertiva, observe a figura 4.2. Nesta figura, pode-se observar que, dada uma trajetória não linear, $S(t)$, de um objeto móvel, a função de previsão quadrática $s(t) = s_0 + v_0(t - t_0) + a_0(t - t_0)^2/2$ aproxima-se melhor da trajetória original de que a função linear $s(t) = s_0 + v_0(t - t_0)$.

No início deste capítulo, três tipos de consultas foram definidas para que seja executada a busca por objetos móveis. Aqui apresentaremos os algoritmos para cada tipo de consulta.

- **Algoritmo de consultas do Tipo 1 *(Busca1)***

  Para responder a uma consulta *Q* do tipo (R, t), são executados os seguintes passos:

  Passo 1. A árvore é percorrida, a partir da raiz, para verificar se o Nó será ponto candidado. Ponto candidato é o ponto que se encontra dentro de uma região especificada pelo retângulo de busca *R*, aumentado de uma região vizinha definida pelo usuário. Esta vizinhança é usada para que pontos que estejam próximos do retângulo *R* de consulta possam ser incluídos também como

prováveis resultados da consulta.

Passo 2. Em seguida, as posições destes pontos candidatos são estimadas utilizando a função quadrática de previsão do movimento $s(t) = s_0 + v_0(t - t_0) + a_0(t - t_0)^2/2$ e verificamos se ele está dentro do retângulo *R* no tempo *t* $(s(t) \in R)$.

Passo 3. Chamadas recursivas, se necessárias ao algoritmo de ***Busca1***, são feitas para verificar se os filhos à direita e à esquerda de cada ponto candidato estão dentro do retângulo de busca (R, t). A necessidade da recursão é verificada através do seguinte teste:

a) Se a coordenada inferior do retângulo de busca for menor ou igual à coordenada do Nó pesquisado, no discriminante da vez, isto indicará a possibilidade de haver filhos à esquerda do Nó, dentro de retângulo de busca, e a chamada recursiva é feita passando-se o ponteiro para o filho esquerdo do Nó.

b) Se a coordenada superior do retângulo de busca for maior ou igual à coordenada do Nó pesquisado, no discriminante da vez, isto indicará a possibilidade de haver filhos à direita do Nó, dentro de retângulo de busca, e a chamada recursiva é feita passando-se o ponteiro para o filho direito do Nó.

A figura 4.3 representa consultas deste tipo onde, dado um determinado retângulo de busca, um ponto $\alpha(t_0)$ é considerado ponto candidato por encontrar-se no primeiro instante dentro do retângulo de busca. Quando for aplicada a função quadrática de previsão do movimento em um instante t, o objeto deslocou-se mas continuou dentro do retângulo de busca no instante t. Desta forma, ele fará parte do resultado da consulta.

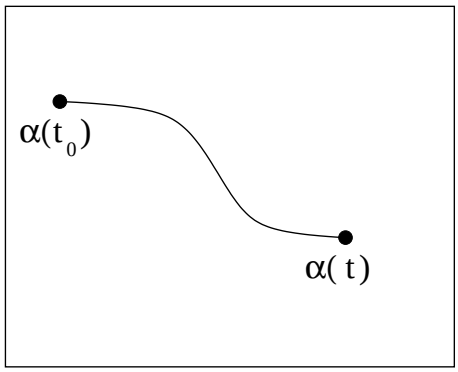


Figura 4.3: Exemplo de consulta tipo 1.

- **Algoritmos de consultas do Tipo 2** ***(Busca2)***

Para responder a estas consultas, um retângulo *R* e um intervalo de tempo $[T_1, T_2]$ são dados.

Passo 1. Atribuímos um Nó *X* igual ao Nó raiz da árvore.

Passo 2. Percorrendo a árvore, é verificado se o Nó está, inicialmente, dentro de uma região especificada pelo retângulo de busca *R*, aumentado da área em sua vizinhança definida pelo usuário. Este ponto encontrado será um ponto candidato.

Passo 3. O ponto correspondente ao Nó *X* responderá positivamente à consulta de três formas:

a) Se, aplicada a função de previsão para o instante T_1, o ponto candidato estiver dentro do retângulo *R*.

b) Se, aplicada a função de previsão para o instante T_2, o ponto candidato estiver dentro do retângulo *R*.

c) Se, em um instante *t* do intervalo $[T_1, T_2]$, a trajetória do ponto candidato interceptar o retângulo *R*. Este procedimento é verificado chamando o algoritmo ***Intercepta***.

Passo 4. Chamadas recursivas do ***Busca2***, a partir do ***passo 2***, são feitas, se necessário, para

os filhos (esquerdo ou direito) do Nó. Esta necessidade é verificada por um teste similar ao executado no algoritmo de ***Busca1***, que é executado para verificar esta possibilidade:

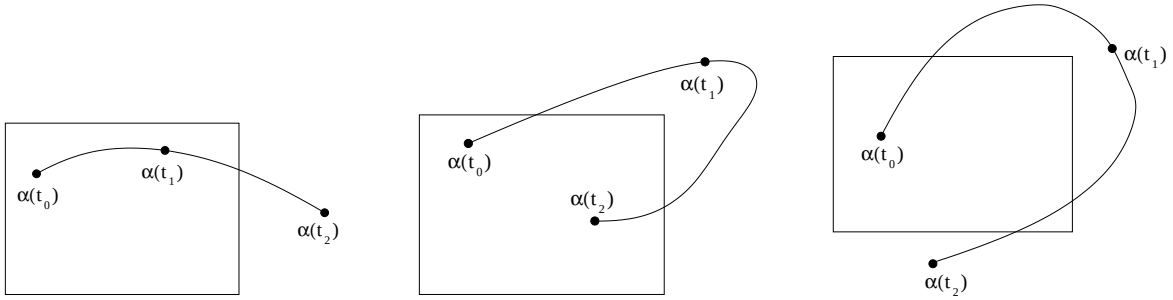
a) Se a coordenada inferior do retângulo de busca for menor ou igual à coordenada do Nó pesquisado, no discriminante da vez, indica a possibilidade de haver filhos à esquerda deste Nó, dentro de retângulo de busca, e a chamada recursiva é feita passando-se o ponteiro para o filho esquerdo do Nó.

b) Se a coordenada superior do retângulo de busca for maior ou igual à coordenada do Nó pesquisado, no discriminante da vez, indica a possibilidade de haver filhos à direita deste Nó, dentro de retângulo de busca, e a chamada recursiva é feita, passando-se o ponteiro para o filho direito do Nó.

Na figura 4.4, estão ilustradas possíveis situações para um ponto ser candidato à consulta do tipo 2. Na primeira situação ***(situação 1)***, o ponto $\alpha(t_0)$ está no retângulo no momento em que a busca é feita para verificação dos pontos candidatos ***(passo 2)***. Em seguida, a função quadrática de previsão é aplicada a este ponto no intervalo $[T_1, T_2]$. O ponto continuará no retângulo apenas para o tempo T_1. No tempo T_2, o ponto candidato $\alpha(t_0)$ estará fora do retângulo de consulta. Como o ponto $\alpha(t_0)$ está dentro do intervalo estabelecido, ou seja, estará dentro do retângulo de busca estabelecido, no instante T_1, (apesar de estar fora do retângulo no intervalo T_2), ele irá fazer parte da resposta à consulta.

Na ***situação 2***, o ponto $\alpha(t_0)$ está no retângulo no momento em que a busca é feita ***(passo 2)***. Quando a função quadrática de previsão do movimento é aplicada, o ponto $\alpha(t_0)$ estará fora para o tempo T_1, mas estará dentro do retângulo no tempo T_2 e assim ele será um ponto na resposta à pesquisa.

Na ***situação 3***, o ponto $\alpha(t_0)$ está no retângulo como ponto candidato ***(passo 2)*** e estará fora do retângulo nos instantes T_1 e T_2 aplicando a função de previsão do movimento. Mas, no intervalo de tempo $[T_1, T_2]$, existe um tempo t qualquer onde a trajetória do ponto $\alpha(t_0)$ intercepta o lado direito e o lado inferior do retângulo. Assim, este ponto também estará na resposta da consulta. Uma parte da trajetória do ponto $\alpha(t_0)$ pode ser representada por uma curva $(x(t), y(t))$. O algoritmo que testa se esta curva $(x(t), y(t))$ intercepta um dos lados do retângulo mostrado na ***situação 3*** é apresentado a seguir.



Situação 1 Situação 2 Situação 3

Figura 4.4: Exemplo de consulta tipo 2.

A figura 4.5 mostra graficamente os pontos *highpoint* e *lowpoint*, que são as coordenadas do retângulo de busca. O índice zero (0) indica valores no eixo *X* e o índice um (1), valores no eixo *Y*. Também mostramos um exemplo de uma curva $(x(t), y(t))$ interceptando o lado superior e o lado esquerdo do retângulo. Esta figura é mostrada para que haja um melhor entendimento do algoritmo ***Intercepta***.

Algoritmo Intercepta

Testa se uma curva $(x(t), y(t))$ intercepta um lado qualquer do retângulo em um tempo (tmp) que esteja dentro do intervalo de tempo $[T_1, T_2]$.

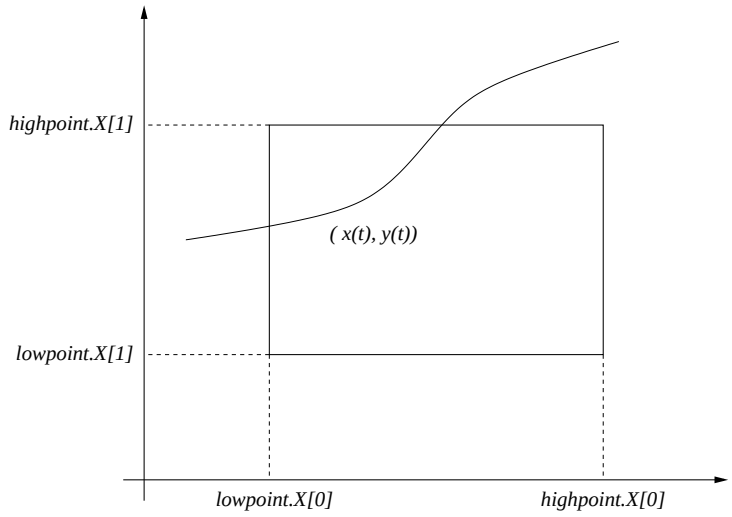


Figura 4.5: Highpoint e Lowpoint.

A localização dos lados do retângulo de busca utilizados nos passos descritos a seguir é mostrada na figura 4.6 para ajudar também na compreensão deste algoritmo.

Passo 1. Para a verificação de interceptação da curva $(x(t), y(t))$, no lado esquerdo do retângulo de busca, em um instante de tempo tmp, os seguintes testes deverão ser feitos:

a) $X(tmp) = \textit{lowpoint}.X[0]$ ou seja, o valor de X, no instante tmp, deverá ser igual ao valor de X na coordenada inferior esquerda do retângulo de busca.

b) $T_1 \leq tmp \leq T_2$ ou seja, o tempo tmp deverá estar no intervalo de tempo especificado na consulta ($[T_1, T_2]$) e

c) $\textit{lowpoint}.X[1] \leq Y(tmp) \leq \textit{highpoint}.X[1]$ ou seja, o valor de Y, no instante tmp, deverá estar entre os valores possíveis para Y, que fazem parte do intervalo entre o limite inferior e superior do lado esquerdo do retângulo de busca.

Passo 2. Para verificar se a curva $(x(t), y(t))$ intercepta o lado direito do retângulo em um instante

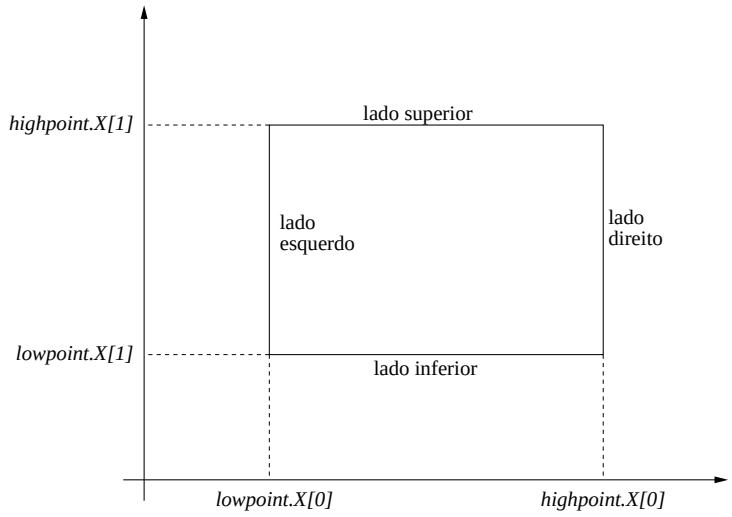


Figura 4.6: Localização dos lados do retângulo utilizada no algoritmo *intercepta*.

de tempo tmp, testar se:

a) $X(tmp) =$ ***highpoint***$.X[0]$ ou seja, o valor de X, no instante tmp, deverá ser igual ao valor X na coordenada inferior direita do retângulo de busca.

b) $T_1 \leq tmp \leq T_2$ isto é, o tempo tmp deverá estar no intervalo de tempo especificado na consulta ($[T_1, T_2]$) e

c) ***lowpoint***$.X[1] \leq Y(tmp) \leq$ ***highpoint***$.X[1]$ ou seja, o valor de Y, no instante tmp, deverá estar entre os valores possíveis para Y do intervalo que vai do limite inferior ao limite superior do lado esquerdo do retângulo de busca.

Passo 3. Os testes executados para verificar se a curva $(x(t), y(t))$ intercepta o lado inferior do retângulo de busca no instante tmp são:

a) $Y(tmp) =$ ***lowpoint***$.X[1]$ ou seja, o valor de Y, no instante tmp, deverá ser igual ao valor de Y na coordenada inferior esquerda do retângulo de busca.

b) $T_1 \leq tmp \leq T_2$ isto é, o tempo tmp deverá estar no intervalo de tempo especificado na

consulta ($[T_1, T_2]$) e

c) *lowpoint*.$X[0] \leq X(tmp) \leq$ *highpoint*.$X[0]$ ou seja, o valor de X no instante tmp, deverá estar entre os valores possíveis para X do intervalo que vai do limite inferior ao limite superior do lado inferior do retângulo de busca.

Passo 4. E, para verificar se a curva $(x(t), y(t))$ intercepta o lado superior do retângulo de busca no instante tmp, os teste são:

a) $Y(tmp) =$ *highpoint*.$X[1]$ ou seja, o valor de Y, no instante tmp, deverá ser igual ao valor Y na coordenada superior esquerda do retângulo de busca

b) $T_1 \leq tmp \leq T_2$ isto é, o tempo tmp deverá estar no intervalo de tempo especificado na consulta ($[T_1, T_2]$) e

c) *lowpoint*.$X[0] \leq X(tmp) \leq$ *highpoint*.$X[0]$ ou seja, o valor de X no instante tmp, deverá estar entre os valores possíveis para X do intervalo que vai do limite inferior ao limite superior do lado superior do retângulo de busca.

- **Algoritmo de consultas do Tipo 3** ***(Busca 3)***

Neste tipo de consulta, dois retângulos R_1 e R_2 e dois tempos T_1 e T_2 são fornecidos como parâmetros da consulta. Isto deve ser interpretado da seguinte forma: o retângulo de busca é R_1 no instante T_1 e desloca-se no plano linearmente para R_2 no instante T_2 segundo a equação: $R(t) = \frac{T_2-t}{T_2-T_1}R_1 + \frac{t-T_1}{T_2-T_1}R_2$. Esta equação servirá para calcular todos os vértices de um retângulo R qualquer, em um tempo t ou seja, um retângulo $R(t)$ situado entre a região trapezóide formada pela união dos retângulos R_1 e R_2. Utilizando a equação, os vértices do retângulo $R(t)$ são: $(A(t), B(t), C(t), D(t))$. Aplicando a equação, quando *t* = T_1, a equação fornecerá o retângulo R_1.

Quando *t* = T_2, o resultado será o retângulo R_2. No retângulo $R(t)$, os vértices serão:

$A(t) = \frac{T_2-t}{T_2-T_1}A_1 + \frac{t-T_1}{T_2-T_1}A_2$

$B(t) = \frac{T_2-t}{T_2-T_1}B_1 + \frac{t-T_1}{T_2-T_1}B_2$

$C(t) = \frac{T_2-t}{T_2-T_1}C_1 + \frac{t-T_1}{T_2-T_1}C_2$

$D(t) = \frac{T_2-t}{T_2-T_1}D_1 + \frac{t-T_1}{T_2-T_1}D_2$

A figura 4.7 mostra o retângulo inicial R_1 com seus vértices, o retângulo final R_2 e seus vértices, a região trapezóide formada pela união dos dois retângulos e um retângulo qualquer $R(t)$ situado nesta região trapezóide. Este retângulo $R(t)$ é determinado calculando-se seus vértices conforme equações mostrada acima.

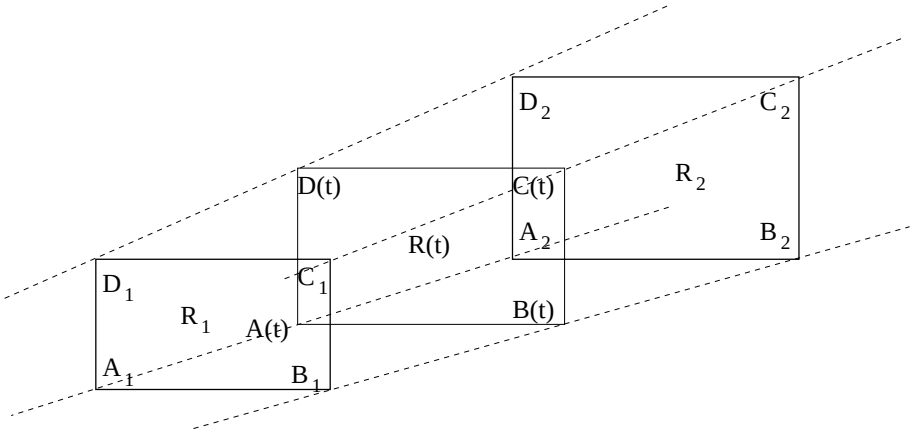


Figura 4.7: Vértices dos retângulos utilizados na consulta tipo 3.

Terão resposta positiva à consulta todos os pontos que se encontrarem nesta região, isto é, $\alpha(t) \in R(t)$ para algum *t*, $t \in [T_1, T_2]$:

Passo 1. A árvore é percorrida a partir da raiz para verifiar se o Nó é um ponto candidato, ou seja, se ele se encontra dentro de uma região especificada pelos retângulos de busca R_1 e R_2

aumentados cada um, de uma região vizinha, definida pelo usuário.

Passo 2. Aplicar a função quadrática de previsão de movimento para o ponto candidato. Um ponto candidato será resposta à consulta se:

(a) aplicada a função de previsão de movimento, no instante T_1, o ponto encontrar-se dentro do retângulo R_1 ou,

(b) aplicada a função de previsão de movimento, no instante T_2, o ponto encontrar-se dentro do retângulo R_1 ou,

(c) aplicada a função de previsão de movimento, no instante T_1, o ponto encontrar-se dentro do retângulo R_2 ou,

(d) aplicada a função de previsão de movimento, no instante T_2, o ponto encontrar-se dentro do retângulo R_2.

(e) a trajetória $(x(t), y(t))$ do ponto intercepta algum retângulo $R(t)$, dentro do trapezóide formado pela união de R_1 e R_2 em algum tempo t no intervalo de tempo $[T_1, T_2]$ como mostra a figura 4.8. O algoritmo ***inteceptaRet***, apresentado à frente, é usado para a execução deste teste.

Passo 3. Se necessário, chamadas recursivas a partir do ***passo 1*** serão feitas de acordo com os testes abaixo:

a) Se a coordenada inferior do retângulo de busca for menor ou igual à coordenada do Nó pesquisado, no discriminante da vez, indica a possibilidade de haver filhos à esquerda deste Nó, dentro de retângulo de busca, e a chamada recursiva é feita, passando-se o ponteiro para o filho esquerdo do Nó.

b) Se a coordenada superior do retângulo de busca for maior ou igual à coordenada do Nó

pesquisado, no discriminante da vez, indica a possibilidade de haver filhos à direita do Nó, dentro de retângulo de busca, e a chamada recursiva é feita, passando-se o ponteiro para o filho direito do Nó.

A figura 4.8 mostra graficamente uma trajetória $(x(t), y(t))$ que intercepta um retângulo $R(t)$, no intervalo do trapezóide especificado pela união de R_1 com R_2.

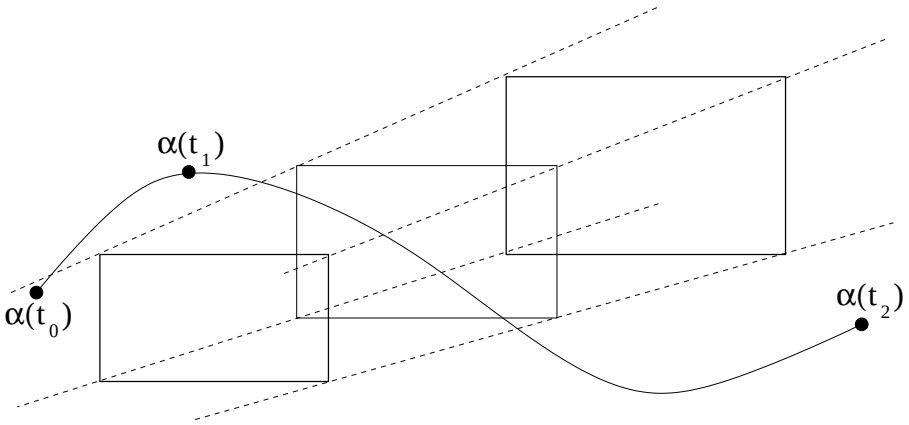


Figura 4.8: Exemplo de consulta tipo 3.

Algoritmo InterceptaRet

Este algoritmo verifica se uma curva $(x(t), y(t))$ intercepta um retângulo $R(t)$ em algum ponto do intervalo $[T_1, T_2]$. As condições necessárias para verificar se alguma curva intercepta a região trapezóide formada pela união de R_1 com R_2:

- existir um t no intervalo $[T_1, T_2]$ tal que:
- $LowpointX(t) \leq X(t) \leq highpointX(t)$ e
- $LowpointY(t) \leq Y(t) \leq highpointY(t)$

Estas expressões são chamadas de inequaçãoes simultâneas, pois são representadas por desigualdades. Para resolvermos este tipo de inequação, utilizamos o estudo do sinal.

Para um melhor entendimento, usaremos a seguinte inequação simultânea como exemplo: $-8 \leq x^2 - 2\text{x} - 8 \leq 0$. Determinam-se assim as soluções $S_1 = (x \in \mathbb{R}^2 \, / x^2 - 2\text{x} - 8 \geq -8)$ e $S_2 = (x \in \mathbb{R}^2 \, / x^2 - 2\text{x} - 8 \leq 0)$ e, em seguida, o resultado desta expressão simultânea será S, formado pela interceção de S_1 com S_2. Detalharemos um pouco mais como encontrar estas soluções; exemplifiquemos:

Passo 1. Separar as inequações, obedecendo o intervalo dado. Assim teremos:

I) $x^2 - 2\text{x} - 8 \geq -8$

II) $x^2 - 2\text{x} - 8 \leq 0$

Passo 2. Determinar as raízes ou zeros de cada uma das funções obtidas pela separação

I) $x^2 - 2\text{x} - 8 > -8$

$x_1 = 0$

$x_2 = 2$

II) $x^2 - 2\text{x} - 8 < 0$

$x_1 = -2$

$x_2 = 4$

Passo 3. Determinar as soluções S_1 e S_2, fazendo o estudo do sinal para cada função, utilizando as raízes x_1 e x_2 encontradas. O sinal de a, coeficiente do termo do segundo grau, nos indica se a parábola terá a concavidade voltada para cima (se $a > 0$) ou voltada para baixo (se $a < 0$). O sinal de b^2-4ac nos fornece indicações sobre a existência de zeros, que devem ser determinados, caso existam.

Passo 4. Calcular a solução *S*, que é dada pela interseção dos intervalos de S_1 e S_2, conforme mostra a figura 4.9. Neste nosso exemplo o resultado de S é o conjunto de todos os pontos de 0 a 2 inclusive.

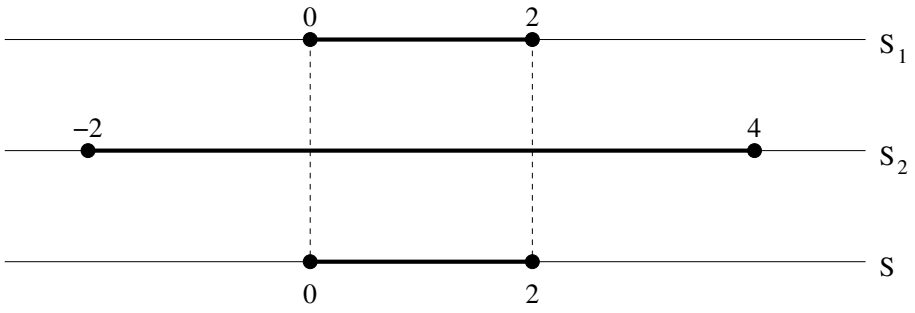


Figura 4.9: Gráfico das soluções S_1, S_2 e S.

Capítulo 5

Os Experimentos

5.1 Introdução

Neste capítulo mostramos como foi projetada a carga de trabalho usada pela nossa estrutura de indexação, o comportamento da estrutura nos diversos testes executados e a comparamos com outra estrutura de indexação para objetos móveis existente.

Outra contribuição apresentada neste trabalho consiste em uma nova técnica para simular o movimento de objetos em um plano. Os objetos desta simulação são indexados pela estrutura de árvore *TPKd*, para que posteriores acesso (consultas) a estes objetos possam ser feitos de forma eficiente, ou seja, que a localização destes objetos seja informada de forma rápida e precisa.

5.2 Geração da carga de trabalho

Necessitamos simular o movimento dos objetos para que possamos fazer a previsão da localização destes objetos utilizando a função quadrática. Esta simulação de movimento foi feita como segue: os pontos móveis deslocam-se ao longo de trajetórias descritas por curvas planas de Bézier por partes [Far02]. O movimento de cada objeto é construído de modo que ele passe por um subconjunto de um conjunto fixo de cidades predeterminadas. O deslocamento de um ponto móvel de uma cidade para outra se dá ao longo de um caminho descrito por uma curva de Bézier, de tal modo que as velocidades de chegada e saída sejam nulas. Isto é feito com o objetivo de que o movimento seja executado com o máximo de realidade possível. A união destes caminhos, desde a cidade origem à cidade destino, constitui a trajetória total do objeto. É possível também que seja calculada a posição, velocidade e aceleração do objeto em cada instante do movimento. A seguir, introduziremos os conceitos matemáticos envolvidos na elaboração das trajetórias que simulam o movimento dos pontos móveis.

5.2.1 Curvas de Bézier

Dados $P_0, P_1, \ldots, P_n \;\; n+1$ pontos no plano $\mathbb{R}^2$, a curva de Bézier determinada por estes pontos é a aplicação $\alpha : [0,1] \longrightarrow \mathbb{R}^2$ dada por

$$\alpha(t) = \sum B_i^n(t) P_i(t)$$

onde $B_i^n(t)$ é, para cada i, o polinômio de Bernstein dado por

$$B_i^n(t) = \binom{n}{i} t^i (1-t) n - i.$$

Note que, $B_0^n(0) = 1, B_0^n(1) = 0, B_n^n(0) = 0, B_n^n(1) = 1 e B_j^n(0) = B_j^n(1) = 0$ para todo $j = 0$ e $j = n$. Sendo assim, conclui-se que $\alpha(0) = P_0$ e $\alpha(1) = P_n$. Observe que a curva de Bézier, determinada por n pontos $P_0, P_1, \ldots, P_n$ não passa necessariamente pelos pontos intermediários $P_1, \ldots, P_n$. Estes pontos apenas determinam a forma da curva que pode ser observada na figura 5.1.

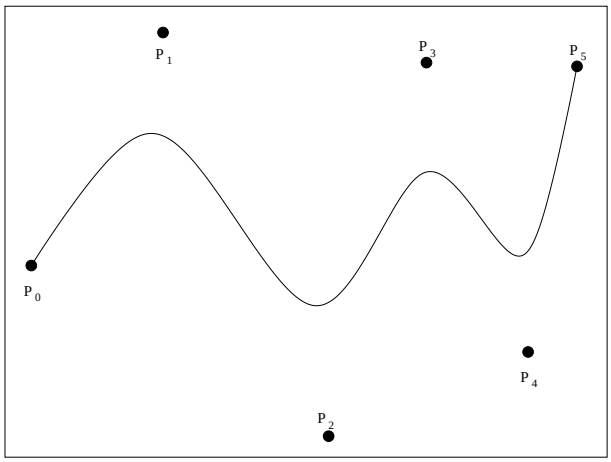


Figura 5.1: Curva de Bézier

Tem-se, ainda, que

$$\alpha'(t) = n \sum (P_{j+1} - P_j) B_j^{n-1}(t);$$

e, sendo assim, conclui-se que:

$$\alpha'(0) = P_1 - P_0 \text{ e } \alpha'(1) = P_n - P_{n-1}.$$

Neste caso, a curva de Bézier determinada pelos pontos

$$P_0 = A, \qquad P_1 = A, \qquad P_2, P_3, P_4 = B, \qquad P5 = B$$

é uma curva ligando os pontos A e B de modo que $\alpha'(0) = 0$ e $\alpha'(1) = 0$. Para este caso, nota-se também que, se os pontos P_2 e P_3 estiverem próximos de A e B respectivamente, então a curva de

Bézier fica geometricamente próxima do segmento ligando os pontos A e B como pode ser observado nas 5.2. Na figura 5.3, os pontos de controle P_2 e P_3, estão distantes de A e B, determinando um outro formato para a curva de Bézier.

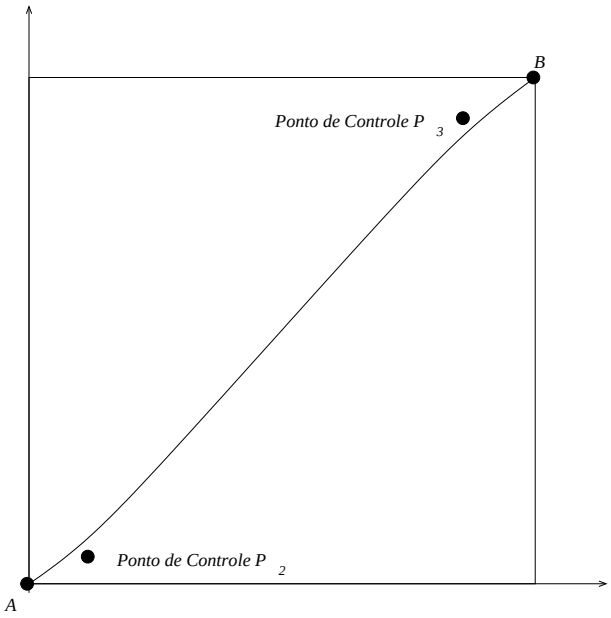


Figura 5.2: Pontos P_2 e P_3 próximos de A e de B.

Sendo $\alpha : [0,1] \longrightarrow \mathbb{R}$ uma curva de Bézier construída com domínio no intervalo $[0,1]$, esta pode ser reparametrizada, para o intervalo de tempo $[a,b]$ qualquer, dada pela curva de Bézier $\beta(t) = \alpha(\frac{t-a}{b-a})$. Desta forma tem-se

$$\beta'(t) = \alpha'(\frac{t-a}{b-a})\frac{1}{b-a} \quad \text{e}$$

$$\beta''(t) = \alpha''(\frac{t-a}{b-a})\frac{1}{(b-a)^2}.$$

Estas fórmulas mostram que, se o tempo para percorrer a curva de Bézier diminui, as velocidades aumentam; e, quando o tempo aumenta, as velocidades diminuem. Neste trabalho, será denominado

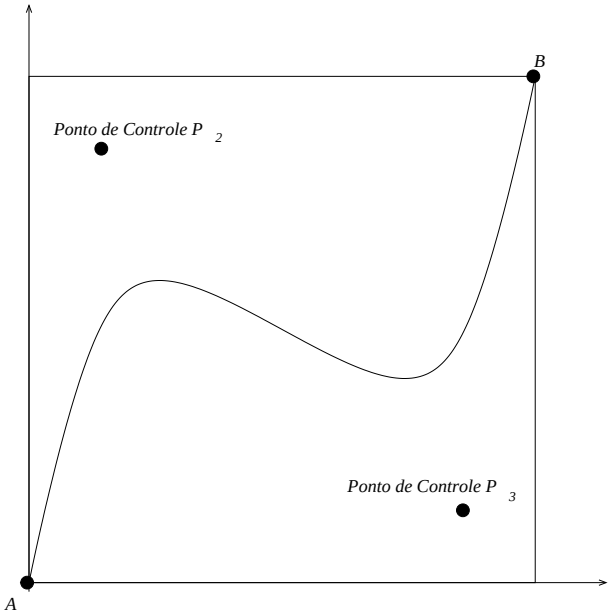


Figura 5.3: Pontos P_2 e P_3 distantes de A e de B.

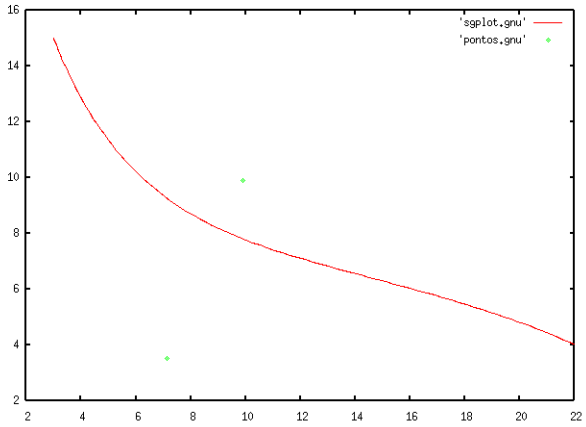


Figura 5.4: Caminho ligando os pontos $(22, 4)$ e $(3, 15)$.

Caminho (figura 5.4) a curva de Bézier ligando dois pontos quaisquer e Trajetória (figura 5.5), a curva formada pela união de caminhos que levarão um ponto móvel da sua origem ao seu destino, passando por pontos intermediários.

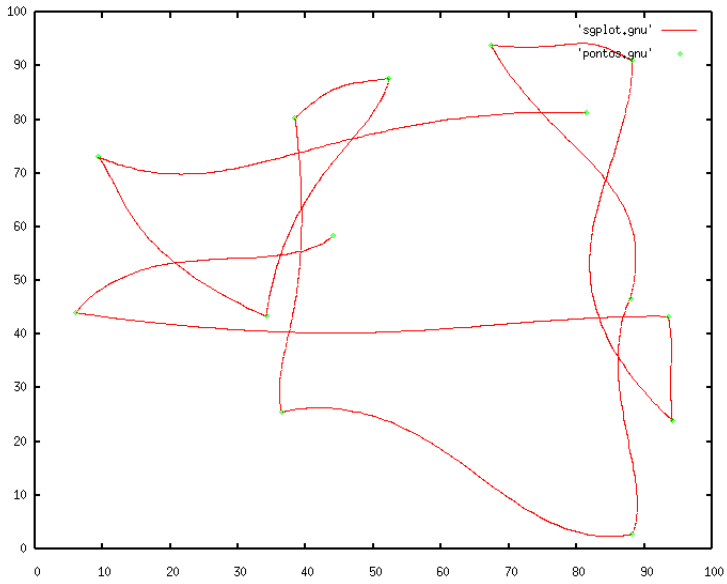


Figura 5.5: Uma trajetória.

5.3 A geração do Movimento

Para simular o movimento dos pontos móveis ao longo de um plano, foi fixado um espaço retangular e distribuído aleatoriamente um número fixo de cidades neste retângulo. Um ponto sempre se movimenta por uma trajetória que o leva de uma cidade origem a uma cidade destino. A criação da trajetória de cada ponto móvel segue três etapas:

- escolha aleatória das cidades por onde o ponto irá passar,
- escolha aleatória do instante em que o ponto começa a mover-se e
- definição do caminho ligando duas cidades consecutivas da trajetória.

O tempo total de movimento do ponto móvel até o seu destino é determinado automaticamente através de critérios estabelecidos na implementação. De fato, o que é determinado automaticamente é

o tempo necessário para percorrer cada caminho intermediário, sendo o tempo total igual à soma destes tempos. Para determinar o tempo gasto em cada caminho, uma velocidade média, para percorrer em linha reta a distância entre duas cidades, é estabelecida . O tempo gasto para percorrer o caminho é calculado pelo quociente entre a distância das duas cidades e esta velocidade média, qualquer que seja o caminho.

5.4 O tempo de inserções, consultas e alterações

Para determinar a seqüência de inserções, consultas e atualizações a serem feitas na simulação, uma seqüência T_j de tempos é gerada da seguinte forma:

- É obtido o menor tempo inicial (T_{min}) e maior tempo final (T_{max}) entre todos os pontos móveis. A diferença entre este dois tempos, resultará no tempo total da simulação.
- T_1 é determinado de modo que $T_1 - T_{min}$ não seja superior a ΔT_{min}.
- Sendo T_j um instante já determinado da seqüência, o instante T_{j+1} é escolhido de forma que a diferença $T_{j+1} - T_j$ não seja inferior a um ΔT_{min} e nem superior a um ΔT_{max} sendo estas constantes determinas como pequenos percentuais do tempo total da simulação.

5.5 Como criar inserções, deleções e consultas

Para cada tempo da seqüência discutida na subsecção anterior, é sorteado aleatoriamente um conjunto de inserções, deleções e consultas. Na simulação, uma atualização se constitui de uma remoção, uma vez

que o objeto já estiver indexado, seguida de uma nova inserção do mesmo ponto com novos atributos para posição, velocidade e aceleração. Um ponto é considerado indexado quando já estiver sido inserido pelo menos uma vez na árvore.

O texto abaixo da tabela 5.1 mostra a parte de um arquivo de carga gerado, a partir do qual as movimentações na árvore *TPKd* são executadas. O arquivo formado é interpretado como segue:

- a letra ***t*** indica os tempos em que as operações são executadas;
- a letra ***i*** indica a inserção de um objeto. Os dados contidos nos parênteses são, respectivamente, posição, velocidade e aceleração. O número no final da linha é a identificação do objeto que será inserido;
- a letra ***d*** indica a remoção de um objeto. Os dados contidos nos parênteses são, respectivamente, posição, velocidade e aceleração. O número no final da linha é a identificação do objeto que será removido e inserido novamente, pois, logo após uma remoção, haverá uma nova inserção do mesmo objeto indicando que será executa uma atualização do posicionamento deste objeto;
- a letra ***s*** indica que será executada uma consulta. O número que segue a letra ***s*** (1,2,3) indica o tipo de consulta. Os parâmetros que seguem são: as coordenadas inferiores e superiores do retângulo, o tempo ou intervalo de tempo em que será executada a consulta e, se ela for do tipo 3, o deslocamento do retângulo de busca.

```
t 0.477868
i ( 91.2785 , 10.4965 ) ( 17.576   , 7.79618  ) ( -3.45628 , 0.0421464 ) 97
i ( 10.374  , 39.4003 ) ( 0.505202 , -1.01553 ) ( 1.55218  , -2.70851  ) 20
t 2.47269
i ( 4.03662 , 10.6214 ) ( 2.15359  , 2.99861  ) ( 3.4503   , 5.36916   ) 90
t 4.39251
i ( 52.0301 , 44.2352 ) ( 4.45666  , 3.1133   ) ( 6.8388   , 4.42974   ) 184
d ( 10.374  , 39.4003 ) ( 0.505202 , -1.01553 ) ( 1.55218  , -2.70851  ) 20
i ( 26.6351 , 14.6377 ) ( 5.84517  , -8.45572 ) ( -0.87421 , 1.40789   ) 20
t 5.80925
d ( 52.0301 , 44.2352 ) ( 4.45666  , 3.1133   ) ( 6.8388   , 4.42974   ) 184
i ( 64.6753 , 52.6484 ) ( 12.3137  , 8.02967  ) ( 2.48661  , 1.45973   ) 184
i ( 74.3139 , 43.7961 ) ( 0.716105 , -2.16567 ) ( 2.60018  , -11.8637  ) 65
i ( 87.9325 , 26.3787 ) ( -4.49542 , -10.022  ) ( 5.92227  , 12.2259   ) 105
s 1 ( 95.9923 , 98.1421 ) ( 97.718 , 99.8677  ) 58.2379 58.2379
.
.
.
t 15.1103
i ( 49.281  , 54.3431 ) ( 10.0606  , 5.16501  ) ( -2.78801  , -1.57883  ) 54
i ( 41.0473 , 18.9297 ) ( -13.1396 , 0.892851 ) ( -0.379047 , 0.0466227 ) 46
i ( 59.2436 , 76.4749 ) ( 6.95566  , -14.0417 ) ( 3.29708   , -6.36275  ) 86
d ( 37.998  , 13.1846 ) ( 2.0158   , 3.9844   ) ( 6.67295   , 15.0792   ) 59
i ( 47.3821 , 63.5448 ) ( -1.6942  , 18.5406  ) ( 1.55993   , -11.3311  ) 59
d ( 64.6753 , 52.6484 ) ( 12.3137  , 8.02967  ) ( 2.48661   , 1.45973   ) 184
i ( 20.6111 , 53.7944 ) ( -12.282  , -2.7105  ) ( 4.56737   , 0.974195  ) 184
i ( 60.2769 , 73.0528 ) ( 3.41867  , 7.08168  ) ( -1.77461  , -4.46901  ) 119
i ( 47.8111 , 21.2269 ) ( 3.35223  , 2.83704  ) ( 4.87375   , 3.29244   ) 62
t 16.2265
i ( 59.1871 , 58.5754 ) ( 9.6117    , 1.5969   ) ( -10.6013  , -2.10965  ) 139
i ( 37.5072 , 27.0599 ) ( -0.431441 , 10.3772  ) ( -0.289567 , 3.374     ) 66
d ( 91.2785 , 10.4965 ) ( 17.576    , 7.79618  ) ( -3.45628  , 0.0421464 ) 97
i ( 81.124  , 34.8998 ) ( -7.2614   , -3.60808 ) ( -9.37393  , -5.75118  ) 97
i ( 12.1258 , 81.0973 ) ( -1.05939  , 1.69261  ) ( 2.00706   , -3.46478  ) 80
i ( 90.9324 , 66.6275 ) ( -0.567236 , -1.65699 ) ( -2.00775  , -4.31637  ) 87
i ( 38.9222 , 34.8566 ) ( 3.78585   , -15.4131 ) ( -0.894169 , 2.60754   ) 32
t 17.7746
s 3 ( 65.601 , 19.1582 ) ( 73.182 , 26.7391 ) 26.187 28.3236 ( -5.9154 , 1.56884 )
.
.
.
t 99.6715
d ( 53.7277 , 68.4811 ) ( -8.31986 , 13.1488 ) ( 1.00856 , -2.04547 ) 82
i ( 40.3552 , 88.8333 ) ( -4.70414 , 6.73936 ) ( 2.38226 , -3.93122 ) 82
t 100.962
d ( 40.3552 , 88.8333 ) ( -4.70414 , 6.73936 ) ( 2.38226 , -3.93122 ) 82
i ( 36.28   , 94.3548 ) ( -1.66704 , 2.00919 ) ( 2.1255 , -3.04089  ) 82
```

Tabela 5.1: Exemplo de um arquivo de carga.

5.6 O comportamento da estrutura

Os algoritmos propostos foram desenvolvidos na linguagem C++ utilizando o compilador GCC 3.2 no sistema operacional Linux. Os experimentos foram testados com a utilização de um Pentium III 1G Mhz e 512MB de memória principal. Foram realizados testes com cargas, utilizando os parâmetros mostrados na tabela 5.2. Nesta tabela mostramos o número de cidades que foram utilizadas nos testes para simular o movimento de um objeto de uma origem a um destino, a quantidade de pontos móveis utilizada nos experimentos, as velocidades máximas e mínimas com as quais os objetos se moviam e os percentuais dos três tipos de consultas para localizar os objetos móveis que foram utilizadas a cada execução.

Número de cidades	$20, 50, 100$
Número de pontos	10.000, $100.000, 200.000, 300.000,$ $400.000, 500.000$
Velocidades Max. e Min.	$(Max = 5, Min = 3)$, $(Max = 3, Min = 2)$
Percentual Consultas 1	0.3, 0.4
Percentual Consultas 2	0.3, 0.4
Percentual Consultas 3	0.3, 0.4

Tabela 5.2: Parâmetros utilizados nos arquivos de carga.

A tabela 5.3 mostra o comportamento da estrutura *TPKd* quando submetida a diversos arquivos de cargas. O peso da árvore é calculado como sendo a quantidade de Nós mais um. Ele mostra o número

Comportamento da árvore TPKd						
Qtde. Máx. Pontos	10.000	100.000	200.000	300.000	400.000	500.000
Peso da árvore	8.928	86.924	175.368	264.060	363.753	455.501
Profundidade	18	23	25	30	25	25
Qtde. Balanceamento	516	4.057	7.351	10.891	17.070	14.031
Qtde. Remoções	16.620	149.121	306.412	464.726	703.997	868.146
Qtde. Inserções	25.547	236.044	481.779	728.785	1.067.749	1.323.646
Qtde. Consultas 1	79	968	1.946	2.926	3.250	4.067
Qtde. Consultas 2	80	705	1.486	2.191	3.261	3.874
Qtde. Consultas 3	99	711	1.434	2.244	4.274	5.249

Tabela 5.3: Comportamento da árvore *TPKd*.

de folhas nulas da árvore. O programa sorteia a quantidade de pontos que será inserido de acordo com o parâmetro de quantidade máxima de pontos a serem inseridos.

Quando a carga foi executada com 100.000 pontos, a altura da árvore aumentou apenas 27% dela, executada com 10.000. Comparando os resultados da execução com 100.000 e 200.000, o aumento da altura da árvore foi de apenas 8%, mostrando um comportamento satisfatório da estrutura com um acréscimo de pontos considerável.

Os balanceamentos ocorrem segundo o critério estabelecido e discutido na seção de inclusão de dados no capítulo anterior. Os dados da quantidade de balanceamento da tabela 5.3 foram todos executados com a constante de balanceamento *c* igual a 1.35. Como foi dito anteriormente, quanto mais

próximo de 1 for a constante de rebalanceamento, menor será a altura da árvore; porém, inversamente proporcional será a quantidade de rebalanceamento. Depois de vários testes executados, concluímos que a constante com o valor acima mencionado mostra uma boa relação entre a altura e a quantidade de balanceamentos.

O número de remoções indica a quantidade de operações de atualizações que foram feitas na simulação do movimento dos objetos. Quando um objeto se move de um caminho a outro, em diversos pontos, sua nova posição é "anotada" de forma aleatória. Para isso, no arquivo de carga, esta atualização é simbolizada com uma remoção e uma nova inserção deste ponto, caso ele já tenha sido inserido anteriormente. Isto explica o motivo pelo qual a quantidade de inserções apresentadas na tabela 5.3 seja bem maior do que a quantidade de remoções feitas.

O quantitativo dos três tipos de consultas apresentados na tabela 5.3, mostra um bom percentual de buscas em relação à quantidade de pontos inseridos. O resultado das previsões de localização dos pontos, obtidos pela função de previsão quadrática, obteve um acerto médio de 90% dos pontos que, efetivamente, estavam no retângulo de busca.

5.6.1 Análise Comparativa

Diversos testes foram executados para analisar o comportamento da estrutura *TPKd* em diferentes cenários. Adicionalmetne, várias simulações foram realizadas para que fosse feita uma análise comparativa entre a abordagem *TPKd* e a abordagem *TPR*.

Inicialmente, vale destacar que a árvore TPKd é uma estrutura de memória principal, ao contrário

da árvore TPR, portanto, o acesso a disco não é necessário para a pesquisa de dados na árvore, sendo apenas necessário no momento de carregar a árvore na memória.

Outro aspecto importante a ser analisado é o seguinte: a árvore TPR tem informações de seus objetos nas folhas e a TPKd tem informação nos nós internos e nas folhas possibilitando que uma consulta possa ser respondida de forma mais rápida já que pode ocorrer de não ser necessário uma pesquisa em toda a profundidade da árvore.

A árvore TPR necessita de um parâmetro que deve ser definido pelo usuário, o *intervalo de update (UI)* isto é, a média de tempo entre duas sucessivas alterações do mesmo ponto que interfere também no tempo horizontal H, ou seja, o tempo em que a árvore permanece válida para responder precisamente às consultas. Assim, mudando H o desempenho da estrutura também é afetado pois, a média de acessos a disco das consultas aumenta. Em contraste, a árvore TPKd tem o mesmo desempenho para os diferentes ajustes, destes parâmetros, apresentados na árvore TPR, pois ela não leva em consideração o tempo máximo horizontal e, o intervalo entre as alterações é feito aleatoriamente, sem considerar intervalo de tempo nem quantidade de alterações.

A trajetória de um objeto utilizada em nosso trabalho não é pré-definida, pois tanto o número de cidades (destinos) quanto o local das cidades escolhidos em cada trajetória é aleatório. Consideramos trajetória, a união de diversos caminhos aleatórios, que levam um objeto de uma origem a um destino. Na TPR o número de cidades (destinos) é pré-determinado e uma trajetória é estabelecida como sendo o movimento de uma cidade (origem) a outra (destino) e uma nova trajetória só é escolhida quando o objeto chega a um destino.

A implementação da árvore TPR requer que seja estimado o tempo de validade da estrutura de

índices. Após este tempo, a estrutura tem que ser refeita implicando em uma reconstrução de todos os retângulos envolventes mínimos. Na árvore TPKd proposta, a estrutura permanece válida até que o critério utilizado no algoritmo de remoção $(Ndel > R * (Qtdnos + 1))$ seja violado. Neste momento, a árvore é reconstruída e a posição dos objetos é atualizada.

Foi testado o comportamento das estruturas TPR e TPKd utilizando cargas de trabalho nas quais muitas alterações acontecem em um intervalo de tempo e os pontos alterados diversas vezes. A construção da árvore TPKd utilizou 30% do tempo de construção da árvore TPR com simulações envolvendo 10.000, 100.000, 200.000, 300.000, 400.000 e 500.000 objetos móveis

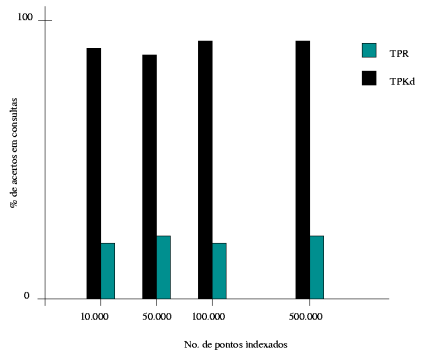


Figura 5.6: Percentual de acertos das consultas.

Utilizando uma mesma carga de movimentos para as duas estruturas, o resultado das buscas aplicando a função de previsão quadrática utilizada pela TPKd obteve uma taxa de acerto muito superior ao resultado obtido pela TPR com a utilização da função linear de previsão. O gráfico 5.6 apresenta uma comparação entre a quantidade de pontos indexados e o percentual de acerto em relação a quantidade de pontos obtidos nos resultados das consultas efetuadas.

Outro fato observado no comportamento das duas estruturas utilizando a mesma carga foi que a

quantidade de candidatos encontrados pela TPR muitas vezes mostrou-se superior aos encontrados pela TPKd, mas com uma quantidade de acertos bem inferior mostrando assim, que a busca na estrutura proposta TPKd, está bem mais eficiente, pois, mesmo encontrando menos pontos, ela obtém uma taxa de acerto bem maior.

Capítulo 6

Conclusão e Trabalhos Futuros

6.1 Introdução

Neste capítulo, além das conclusões obtidas ao longo deste projeto, serão apresentadas uma série de propostas de continuidade para o trabalho aqui apresentado.

6.2 Considerações finais e contribuições

O crescente avanço das tecnologias nas comunicações sem fio vem incentivando o desenvolvimento de aplicações que fazem uso do atributo *localização*, para que respostas sobre a localização de objetos, em um tempo corrente e futuro, possam sem respondidas. Interessados em contribuir para que este nicho de aplicações obtenha êxito, nós fizemos um estudo sobre a indexação de objetos móveis, suas características, seus problemas e possíveis soluções.

A realização deste trabalho de dissertação envolveu uma análise das estruturas de indexação existentes e de formas de indexação de objetos móveis já propostas, com o intuito de desenvolver um método de indexação que respondesse às consultas que utilizam a localização de tais objetos móveis como atributo de pesquisa.

Desta forma, apresentamos uma nova técnica de indexação de objetos móveis, a qual chamamos árvore *TPKd*, baseada na estrutura de dados de árvore *Kd*. Esta estrutura herda da árvore *Kd* as características básicas, mas é acrescentado também, algumas modificações, tais como, rebalanceamento parcial e critério para auto reconstrução, que corrigem algumas deficiências encontradas na árvore *Kd*.

Outro contribuição deste trabalho consiste em considerarmos um objeto se movendo em uma trajetória não linear. Com isso, ao invés de prevermos a trajetória de um objeto através de uma função linear, utilizamos uma função de previsão de movimento quadrática, fazendo que consultas possam ser respondidas, com maior precisão, conforme mostrado anteriormente.

O desenvolvimento de um protótipo para gerenciar a árvore *TPKd* possibilitou que diversas simulações de inserções, remoções e consultas pudessem ser feitas, para que a análise sobre o desempenho da estrutura proposta fosse feita corretamente.

O resultado obtido com a execução de diversos tipos de consultas se mostrou preciso. Os experimentos realizados mostraram que a árvore trabalha bem com grande quantidade de objetos. Os balanceamentos parciais e reconstruções na árvore são feitos automaticamente através do uso de critérios que indicam suas necessidades. Isto faz com que as consultas tenham maior precisão em seus resultados e que nenhuma informação adicional deve ser tratada para garantir estas precisões.

Com isso, pela integração de técnicas de indexação com técnicas de cálculos de equações, nós acreditamos estar contribuindo para o problema da freqüência de alterações das estruturas de indexação, do armazenamento de atributos que mudam freqüentemente e para o melhoramento do desempenho das consultas do tipo 1, tipo 2 e tipo 3 definidas no capítulo 4.

6.3 Trabalhos Futuros

Como trabalhos futuros podemos sugerir o desenvolvimento de outros tipos de consultas tais como consultas a vizinhos próximos, e projetar um *pool* de trajetórias que simulem uma variedade maior de movimentos dos objetos.

Podemos ainda pesquisar eventos externos que possam indicar a necessidade e a possibilidade de alterações da estrutura de índice e a atualização da posição espacial dos objetos.

Novas comparações com outras estruturas de indexação que são utilizadas para indexar o movimento de objetos também poderão ser feitas para análises comportamentais da estrutura. Extensões da árvore TPKd baseadas em outras extensões de árvore *Kd* também representam um bom caminho para pesquisas futuras.

Referências Bibliográficas

[AAE00] Pankaj K. Agarwal, Lars Arge, and Jeff Erickson. Indexing moving points. In *Symposium on Principles of Database Systems*, pages 175–186, 2000.

[AAV01] Pankaj K. Agarwal, Lars Arge, and Jan Vahrenhold. Time responsive external data structures for moving points. In *Workshop on Algorithms and Data Structures*, pages 50–61, 2001.

[And99] Andersson. General balanced trees. *ALGORITHMS: Journal of Algorithms*, 30, 1999.

[ASV99] Lars Arge, Vasilis Samoladas, and Jeffrey Scott Vitter. On two-dimensional indexability and optimal range search indexing. pages 346–357, 1999.

[AVL62] G. M. Adelson-Velski and E. M. Landis. An algorithm for organization of information. *Dokl. Akad. Nauk SSSR*, 146:263–266, 1962.

[AWS92] C. Ahlberg, C. Williamson, and B. Shneideerman. Dynamic queries for information exploration: An implementation and evaluation. In *Proc. of CHI-92*, pages 619–626, Monterey, CA, 1992.

[Bar99] Daniel Barbara. Mobile computing and databases - a survey. *Knowledge and Data Engineering*, 11(1):108–117, 1999.

[Bay72] Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. 1(4):290–306, November 1972.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[Ben90] Jon Louis Bentley. K-d trees for semidynamic point sets. In *Proceedings of the sixth annual symposium on Computational geometry*, pages 187–197. ACM Press, 1990.

[BER85] D. A. Beckley, M. W. Evens, and V. K. Raman. Multikey retrieval from K-d trees and QUAD-trees. pages 291–301, 1985.

[BF79] Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Computing Surveys (CSUR)*, 11(4):397–409, 1979.

[BGZ97] Julien Basch, Leonidas J. Guibas, and Li Zhang. Proximity problems on moving points. In *Symposium on Computational Geometry*, pages 344–351, 1997.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference*, pages 322–331, Atlantic City, NJ, June 1990.

[CLR89] Thomas T. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. MIT Press, 1989.

[DK02] Sukho Lee Dongseop Kwon, Sangjun Lee. Indexing the current positions of moving objects using the lazy update r-tree. In *3rd International Conference on Mobile Data Management (MDM2002)*, pages 113–120, 2002.

[Far02] Gerald Farin. *Curves and surfaces for CAGD: a practical guide*. Morgan Kaufmann Publishers Inc., 2002.

[FGNS00] Luca Forlizzi, Ralf Hartmut Güting, Enrico Nardelli, and Markus Schneider. A data model and data structures for moving objects databases. pages 319–330, 2000.

[GG98] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[Gut84] A. Guttman. R-trees : A Dynamic Index Structure for Spatial Searching. In *Proc. of ACM SIGMOD Int. Symp. on the Management of Data*, pages 45–57, 1984.

[HKTG02] Marios Hadjieleftheriou, George Kollios, Vassilis J. Tsotras, and Dimitrios Gunopulos. Efficient indexing of spatiotemporal objects. In *Extending Database Technology*, pages 251–268, 2002.

[HNP95] J.M. Hellerstein, J.F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *Proc. 21st Int. Conf. on Very Large Data Bases (VLDB)*, pages 562–573, Zürich, 1995.

[JS94] Vinit Jain and Ben Shneiderman. Data structures for dynamic queries: An analytical and experimental evaluation. In *Advanced Visual Interfaces*, pages 1–11, 1994.

[KGT99] George Kollios, Dimitrios Gunopulos, and Vassilis J. Tsotras. On indexing mobile objects. pages 261–272, 1999.

[KMH97] Marcel Kornacker, C. Mohan, and Joseph M. Hellerstein. Concurrency and recovery in generalized search trees. In *Proc. ACM-SIGMOD International Conference on Management of Data, Tucson, May 1997.*, pages 62–72, 1997.

[Knu73] Donald E. Knuth. *Fundamental Algorithms*, volume 3 of *The Art of Computer Programming*, section 1.2, pages 10–119. Addison-Wesley, Reading, Massachusetts, second edition, 10 January 1973. This is a full INBOOK entry.

[KTF98] A. Kumar, V.J. Tsotras, and C. Faloutsos. Designing Access Methods for Bitemporal Databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE'98)*, 10(1):1–2, 1998.

[Moo90] A. W. Moore. *Efficient Memory-based Learning for Robot Control*. PhD thesis, Cambridge, UK, 1990.

[MSI02] Hoda Mokhtar, Jianwen Su, and Oscar Ibarra. On moving object queries: (extended abstract). In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 188–198. ACM Press, 2002.

[NST98] M. Nascimento, J. Silva, and Y. Theodoridis. Access structures for moving points, 1998.

[PAAV03] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. Bkd-tree: A dynamic scalable kd-tree. To appear in proceedings fo SSTD'03 in Santorini Island, Greece, July 2003.

[PAhP03] Cecilia M. Procopiuc, Pankaj K. Agarwal, and Sariel har Peled. Star-tree: An efficient self-adjusting index for moving objects. Lecture Notes in Computer Science, pages LNCS 2409, 178ff. Springer-Verlag, Berlin, July 12 2003.

[PJ01] Dieter Pfoser and Christian S. Jensen. Querying the trajectories of on-line mobile objects. In *Second ACM international workshop on Data engineering for wireless and mobile access*, pages 66–73. ACM Press, 2001.

[Rob81] J. T. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM SIGMOD Conference*, pages 10–18, Ann Arbor, MI, April 1981.

[SJ99] Simonas Saltenis and Christian S. Jensen. R-tree based indexing of general spatio-temporal data, 1999.

[SJ02] Simonas Saltenis and Christian S. Jensen. Indexing of moving objects for location-based services. In *ICDE*, 2002.

[SJLL00] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD Conference*, pages 331–342, 2000.

[SW95] A. Prasad Sistla and Ouri Wolfson. Temporal conditions and integrity constraints in active database systems. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 269–280. ACM Press, 1995.

[SWCD97] A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain, and Son Dao. Modeling and querying moving objects. In *ICDE*, pages 422–432, 1997.

[SWCD98] A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain, and Son Dao. Querying the uncertain position of moving objects. *Lecture Notes in Computer Science*, 1399:310–??, 1998.

[vKO91] Marc J. van Kreveld and Mark H. Overmars. The divided k-d trees. *Algorithmica*, 6(6):840–858, 1991.

[WCD+98] Ouri Wolfson, Sam Chamberlain, Son Dao, Liqin Jiang, and Gisela Mendez. Cost and imprecision in modeling the position of moving objects. In *ICDE*, pages 588–596, 1998.

[Wol02] O. Wolfson. Moving objects information management: The database challenge. In *Proceedings of the 5th Workshop on Next Generation Information Technologies and Systems (NGITS-2002)*, pages 75–89. Springer LNCS 2382, 2002.

[WSX+99] Ouri Wolfson, Prasad Sistla, Bo Xu, Jutai Zhou, and Sam Chamberlain. DOMINO: Databases fOr MovINg Objects tracking. pages 547–549, 1999.

[WXCJ98] Ouri Wolfson, Bo Xu, Sam Chamberlain, and Liqin Jiang. Moving objects databases: Issues and solutions. In *Statistical and Scientific Database Management*, pages 111–122, 1998.