UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MIGUEL RODRIGUES FORNARI

# Análise e Desenvolvimento de um Novo Algoritmo de Junção Espacial para SGBD Geográficos

Tese apresentada como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação

Prof. Dr. Cirano Iochpe
Orientador

Porto Alegre, setembro de 2006.

**CIP – CATALOGAÇÃO NA PUBLICAÇÃO**

Fornari, Miguel Rodrigues

Análise e Desenvolvimento de um Novo Algoritmo de Junção Espacial para SGBD Geográficos / Miguel Rodrigues Fornari – Porto Alegre: Programa de Pós-Graduação em Computação, 2006.

132 f.:il.

Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2006. Orientador: Cirano Iochpe.

1.Bancos de Dados Geográficos. 2. Otimização de Consultas 3.Algoritmos de Junção Espacial. I. Iochpe, Cirano. II. Título

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Ferraz Hennemann
Vice-Reitor: Prof. Pedro Cezar Dutra Fonseca
Pró-Reitora de Pós-Graduação: Profa. Valquiria Linck Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

# AGRADECIMENTOS

Agradeço a Deus, por tudo quem ter permitido na minha vida: esposa, filho, trabalho, estudos, alegrias e dificuldades.

Agradeço a Márcia, minha esposa, e Daniel, meu filho, por terem suportado os momentos de ausência e stress.

Agradeço aos meus pais, que me educaram e formaram meu caráter.

Agradeço a minha comunidade na Igreja, pelas orações e momentos de desabafo.

Agradeço a todos que me estimularam a perseguir este objetivo até o final, especialmente nos momentos de crise.

Agradeço ao meu orientador, prof. Cirano, por ter auxiliado muitíssimas vezes a encontrar soluções adequadas.

Agradeço a todo Instituto de Informática e UFRGS, seus professores e funcionários, pela estrutura que tornou capaz a realização deste doutorado, além do mestrado e graduação.

Uma referência especial a Profa. Lia, que foi minha orientadora no Mestrado, e agora esta junto de Deus. Certamente contribuiu de maneira decisiva na minha formação.

Agradeço aos colegas e amigos de pós-graduação, que compartilharam bons e maus momentos.

# SUMÁRIO

# LISTA DE FIGURAS

# LISTA DE TABELAS

# RESUMO

Um Sistema de Informação Geográfica armazena e mantém dados geográficos, combinando-os, para obter novas representações do espaço geográfico. A junção espacial combina duas relações de geometrias geo-referenciadas de acordo com algum predicado espacial, como intersecção e distância entre objetos. Trata-se de uma operação essencial, pois é constantemente utilizada e possui um alto custo de realização devido a realização de grande número de operações de Entrada/Saída e a complexidade do algoritmo.

Este trabalho estuda o desempenho de algoritmos de junção espacial. Inicialmente, apresenta a análise dos algoritmos já publicados na literatura, obtendo expressões de custo para número de operações de disco e processamento. Após, descreve-se a implementação de alguns algoritmos em um ambiente de testes. Este ambiente permite ao usuário variar diversos parâmetros de entrada: cardinalidade dos conjuntos, memória disponível e predicado de junção, envolvendo dados reais e sintéticos.

O ambiente de testes inclui os algoritmos de Laços Aninhados, *Partition Based Spatial Join Method* (PBSM), *Synchronized Tree Transversal* (STT) para árvores R* e *Iterative Spatial Stripped Join* (ISSJ). Os testes demonstraram que o STT é adequado para conjuntos pequenos de dados; o ISSJ se houver memória suficiente para ordenar os conjuntos internamente; e o PBSM se houver pouca memória disponível para buffer de dados.

A partir da análise um novo algoritmo, chamado *Histogram-based Hash Stripped Join* (HHSJ) é apresentado. O HSSJ utiliza histogramas da distribuição dos objetos no espaço para definir o particionamento, armazena os objetos em arquivos organizados em *hash* e subdivide o espaço em faixas (*strips*) para reduzir o processamento. Os testes indicam que o HHSJ é mais rápido na maioria dos cenários, sendo ainda mais vantajoso quanto maior o número de objetos envolvidos na junção.

Um módulo de otimização de consultas baseado em custos, capaz de escolher o melhor algoritmo para realizar a etapa de filtragem é descrito. O módulo utiliza informações estatísticas mantidas no dicionário de dados para estimar o tempo de resposta de cada algoritmo, e indicar o mais rápido para realizar uma operação específica. Este otimizador de consultas acertou a indicação em 88,9% dos casos, errando apenas na junção de conjuntos pequenos, quando o impacto é menor.

**Palavras-Chave:** Bancos de Dados Geográficos, processamento de consulta, junção espacial

**Analysis and Design of a New Algorithm to Perform Spatial Join in Geographic DBMS**

# ABSTRACT

A Geographic Information System (GIS) stores geographic data, combining them to obtain new representations of the geographic space. The spatial join operation combines two sets of spatial features, *A* and *B*, based on a spatial predicate. It is a fundamental as well as one of the most expensive operations in GIS. Combining pairs of spatial, georreferenced data objects of two different, and probably large data sets implies the execution of a significant number of Input/Output (I/O) operations as well as a large number of CPU operations.

This work presents a study about the performance of spatial join algorithms. Firstly, an analysis of the algorithms is realized. As a result, mathematical expressions are identified to predict the number of I/O operations and the algorithm complexity. After this, some of the algorithms (e.g.; *Nested Loops*, *Partition Based Spatial Join Method* (PBSM), *Synchronized Tree Transversal* (STT) to R-Trees and *Iterative Spatial Stripped Join* (ISSJ)) are implemented, allowing the execution of a series of tests in different spatial join scenarios. The tests were performed using both synthetic and real data sets.

Based on the results, a new algorithm, called *Histogram-based Hash Stripped Join* (HHSJ), is proposed. The partitioning of the space is carried out according to the spatial distribution of the objects, maintained in histograms. In addition, a hash file is created for each input data set and used to enhance both the storage of and the access to the minimum bounding rectangles (MBR) of the respective set elements. Furthermore, the space is divided in strips, to reduce the processing time. The results showed that the new algorithm is faster in almost all scenarios, specially when bigger data sets are processed.

Finally, a query optimizer based on costs, capable to choose the best algorithm to perform the filter step of a spatial join operation, is presented. The query optimizer uses statistical information stored in the data dictionary to estimate the response time for each algorithm and chooses the faster to realize the operation. This query optimizer choose the right one on 88.9% of cases, mistaken just in spatial join envolving small data sets, when the impact is small.

**Keywords:** Geographic Database Management Systems, query processing, spatial join.

# 1 INTRODUÇÃO

Um Sistema de Informação Geográfica armazena e mantém dados geográficos, combinando-os, para obter novas representações do espaço geográfico. A junção espacial combina dois conjuntos de geometrias geo-referenciadas, por exemplo, cidades e rios, de acordo com algum predicado espacial, como intersecção e distância entre objetos.

Na literatura já existem diversos algoritmos para realizar junções espaciais. Este trabalho analisa-os, obtendo expressões de custo relativas ao número de operações de Entrada e Saída (E/S), e custo de processamento. A partir das expressões obtidas, foram selecionados os algoritmos de menor custo e implementados em um ambiente de testes.

O ambiente de testes inclui os algoritmos de Laços Aninhados, *Partition Based Spatial Join Method* (PBSM), *Synchronized Tree Transversal* (STT) para árvores R* e *Iterative Spatial Stripped Join* (ISSJ). Os testes realizados envolvem 20 conjuntos de dados reais e vários de dados sintéticos. Os testes demonstraram que o STT é adequado para conjuntos pequenos de dados; o ISSJ se houver memória suficiente para ordenar os conjuntos internamente; e o PBSM se houver pouca memória disponível para buffer de dados.

Com base na análise realizada e nas observações do comportamento de cada algoritmo, características interessantes deles foram combinadas em um novo algoritmo, chamado *Histogram-based Hash Stripped Join* (HHSJ). O HSSJ utiliza histogramas da distribuição dos objetos no espaço para definir o particionamento, armazena os objetos em arquivos organizados em *hash* e subdivide o espaço em faixas (*strips*) para reduzir o processamento. Os testes indicam que o HHSJ é mais rápido na maioria dos cenários, sendo ainda mais vantajoso quanto maior o número de objetos envolvidos na junção.

Com o objetivo de permitir, *a priori*, a seleção de um algoritmo capaz de realizar uma determinada operação de junção espacial no menor tempo de resposta. Um módulo de otimização de consultas baseado em custos, capaz de escolher o melhor algoritmo para realizar a etapa de filtragem é descrito. O módulo utiliza informações estatísticas mantidas no dicionário de dados para estimar o tempo de resposta de cada algoritmo, e indicar o mais rápido para realizar uma operação específica. Este otimizador de consultas acertou a indicação em 88,9% dos casos, errando apenas na junção de conjuntos pequenos, quando o impacto é menor.

## 1.1 Conceitos básicos

Um Sistema de Informação Geográfica (SIG) (RIGAUX, 2000, p.3) armazena e mantém dados geográficos, combinando-os, para obter novas representações do espaço geográfico. Tais sistemas, normalmente, apresentam ferramentas de análise espacial e de simulação, facilitando o trabalho em muitas áreas como, por exemplo, planejamento urbano ou rural, aplicações militares e controle de tráfego.

Um Sistema de Gerência de Bancos de Dados Geográficos (SGBDG) é um caso especial de SGBD, caracterizado por manipular dados cuja geometria está referenciada à superfície planetária (RIGAUX, 2000, p.21), sendo um componente essencial de um SIG. Em um SGBDG, a representação da realidade é materializada, no banco de dados, de forma complementar, através dos chamados *dados descritivos* e *dados espaciais*. Os primeiros são usados para descrever características não geo-referenciadas da realidade (ex.: nome do município). Os dados espaciais representam aspectos ligados à geometria e à localização dos objetos reais. A geometria dos objetos é, usualmente, representada no banco de dados geográficos (BDG) através de pontos, linhas, polígonos ou combinações destes elementos gráficos.

As consultas submetidas pelo usuário são expressas em uma linguagem de alto nível, como SQL. O otimizador de consultas é um módulo do SGBDG capaz de processar estas consultas, com o fim de gerar código executável e eficiente. Para tanto, ele transforma a consulta em operações básicas, que possam ser realizadas pelo SGBDG. Após, o otimizador define a forma mais eficiente de realizar cada uma das operações básicas. Ao contrário dos SGBD Relacionais, em um SGBDG, o tempo de processamento em memória é significativo, podendo ser dominante (SUN, 2003)(BANDI, 2004), e não pode ser desprezado durante a etapa de otimização de consultas. Por exemplo, calcular a área com base em um polígono irregular, definido por mais de 1000 pontos, demanda grande capacidade computacional por se tratar de um algoritmo complexo. Assim, o otimizador de consultas de um SGBDG deve considerar o tempo de operações de E/S em disco e o tempo de operações de processamento realizadas pela UCP (Unidade Central de Processamento) na memória principal do sistema.

Existem diversas operações básicas que podem ser realizadas como seleção por região, distância e centróide (ARONOFF, 1989)(CÂMARA, 2000)(CRISMAN, 1997). Alguns autores organizaram estas operações em classificações diferentes, de acordo com critérios particulares. Este trabalho concentra-se na operação de junção espacial, que combina dois conjuntos de objetos espaciais com base em um predicado espacial. Esta operação está entre as mais importantes e mais difíceis de serem realizadas por um SGBDG. Diversos predicados espaciais podem ser utilizados para definir uma junção espacial:

- Predicados topológicos: são invariantes com relação a transformações topológicas. Pullar e Egenhofer (1988) e Egenhofer e Franzosa (1991) definiu 8 predicados possíveis entre dois polígonos *A* e *B*: *A* disjunto de *B*, *A* toca *B*, *A* intersecciona B, *A* igual à *B*, *A* contém *B*, *A* dentro de *B*, *A* cobre *B* e *A* é coberto por *B*. Hadzilacos (1992) estendeu o trabalho de Egenhofer e identificou um conjunto de predicados entre dois objetos espaciais representados por ponto, linha ou um polígono.
- Predicados métricos ou de distância: testam uma condição a partir do valor calculado por um operador unário com resultado escalar como, por exemplo,

área de um polígono ser maior que determinado valor ou a distância entre dois pontos ser menor que um limite máximo.

- Predicados de direção ou ordem: estabelecem relacionamentos como, por exemplo, *A* ao lado de *B* ou *A* ao Norte de *B*.

Se em um BDG, estiver definida uma relação de objetos geográficos representando municípios, seus prováveis dados descritivos seriam o *nome do município*, a *data de emancipação* e *número de habitantes* do mesmo. Dentre os dados espaciais, pode-se imaginar o atributo *Limites*, representado por um polígono, o qual identifica a forma e as fronteiras de cada município. Portanto, o atributo `Limites` seria do tipo `Polígono`.

Suponha, agora, que o BDG mantém uma outra relação contendo dados sobre os rios da região, como *nome*, *PH*, *grau de poluição* e o *curso*, sendo este último atributo representado por uma polilinha. Na consulta "recuperar Municípios banhados por algum Rio", ocorre uma junção espacial entre duas relações, utilizando o predicado espacial de intersecção. A consulta SQL[1] é

```
SELECT Municípios.Nome, Rios.Nome
FROM   Municípios, Rios,
WHERE  Overlaps(Municípios.Limites, Rios.Curso)=True;
```

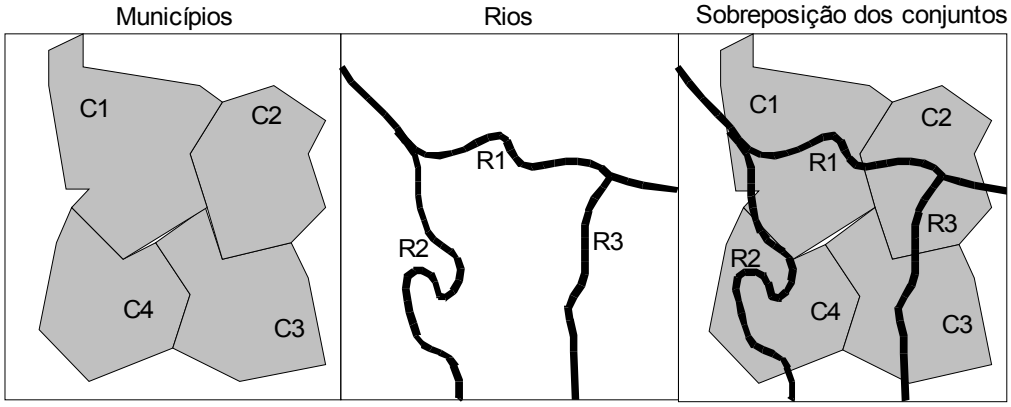



Figura 1.1 : Exemplo de junção espacial

A figura 1.1 ilustra este caso. À esquerda estão polígonos representando a área de cada município. Ao centro, encontram-se polilinhas representando os rios da região. Por fim, à direita da figura, é apresentado o resultado da sobreposição de ambas as camadas de informação espacial. A resposta à consulta acima, seria, portanto, formada pelo conjunto de pares RS = {(C1, R1), (C2, R1), (C1, R2), (C4, R2), (C2, R3), (C3, R3) }.

Devido à complexidade e quantidade de dados manipulados, Orenstein (1986) sugeriu dividir operações espaciais, incluindo a junção espacial, em duas etapas:

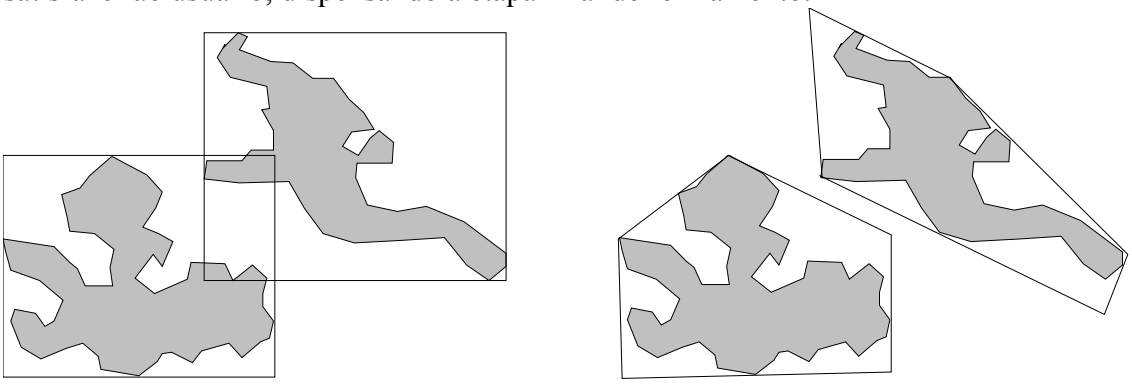
- **Etapa de filtragem**: testa o predicado espacial para as aproximações de objetos, resultando em uma lista de pares candidatos. A forma de aproximação usual são os envelopes, chamados *Minimum Bounding Rectangles* (MBR). Nesta etapa, o predicado espacial deve ser adaptado, para garantir que todos os possíveis pares sejam incluídos na lista.

[1]A notação SQL utilizada não é particular de nenhum SGBDG, pois seu objetivo é apenas dar um exemplo simples de ser compreendido.

- **Etapa de refinamento:** o predicado espacial é verificado para a geometria completa de cada objeto, considerando, apenas, os pares selecionados na etapa anterior. O tempo de processamento, nesta etapa, é maior, devido à complexidade da geometria dos objetos.

Esta abordagem de Orenstein, apesar de largamente utilizada, inclusive em SGBD comerciais, como Oracle, (KOTHURI e RAVADA, 2005), pode gerar um número significativo de falsos candidatos, isto é, pares para os quais os respectivos MBR atendem ao predicado de junção espacial, porém, quando verificados pela sua geometria completa, não atendem ao predicado espacial. Este problema de falsos candidatos ocorre porque MBR é uma aproximação muito grosseira para certos polígonos, abrangendo áreas que não são, de fato, ocupadas pelos objetos. A figura 1.2a ilustra uma situação deste tipo, onde os MBR se interseccionam, mas não os objetos reais.

Para reduzir o número de falsos candidatos, outros autores acrescentam uma etapa intermediária, utilizando aproximações mais detalhadas dos objetos reais, mas mantendo a complexidade controlada. Entre as alternativas encontram-se polígonos convexos de *n*-lados, sendo *n* um número pequeno; ou *Minimum Bounding Ellipse;* (MBE) ou aproximações *raster*. A figura 1.2b ilustra um caso em que a utilização de polígonos convexos de 5-lados é suficiente para identificar que os objetos não se interseccionam. Brinkhoff (1994) apresenta um estudo sobre estas alternativas e Kriegel (2003) uma aplicação para junções espaciais. Azevedo (2004, 2005) argumenta que utilizando uma aproximação *raster*, o grau de correção obtido já é suficiente para satisfazer ao usuário, dispensando a etapa final de refinamento.



(a) MBR (b) Polígono convexo de 5 lados

Figura 1.2 : Dois objetos representados por polígonos, aproximados por MBR (a) ou por polígonos convexos de 5 lados (b).

Vários fatores podem afetar o desempenho de um algoritmo de junção espacial. Alguns exemplos são a cardinalidade dos conjuntos, o tipo de objeto geométrico que é manipulado, a distribuição dos objetos no espaço e o predicado de junção. Também são importantes as características do ambiente computacional, como velocidade de acesso a dados em disco e memória disponível.

## 1.2 Motivação

Diversas organizações manipulam informações geográficas para realizar suas tarefas. Setores como meio-ambiente, planejamento e controle de áreas urbanas e rurais, forças armadas, empresas de *geomarketing* e pesquisa de opinião baseiam suas atividades diárias em dados geográficos. Para analisar e compreender suas informações, a operação de junção espacial é fundamental, pois permite combinar dados isolados. A junção

espacial é a operação algébrica que relaciona objetos de diferentes conjuntos com base em sua localização ou posição relativa. Para o planejamento urbano, por exemplo, mais do que saber a localização de escolas no município, é interessante combinar este dado com a densidade populacional em cada setor censitário e com os meios de transporte disponíveis, pois, dentre outras informações, estes parâmetros podem definir o número de crianças incluídas na área de abrangência da escola.

Para que o SGBDG realize sua tarefa, seus algoritmos devem ser eficientes, especialmente os de junção espacial, pois esta operação apresenta maior complexidade, exigindo um número de operações de E/S e de processamento significativo, maior que outras operações espaciais ou não-espaciais. Combinar, por exemplo, 2.249.727 ruas do estado da Califórnia com 98.451 cursos de água[2], sendo que cada rua ou curso de água é representado por dezenas ou centenas de pontos diferentes, para identificar cruzamentos entre ruas e cursos de água, demanda um esforço computacional significativo. Um algoritmo força bruta poderia tentar todas as combinações possíveis, ou seja 2.249.727 x 98.451. Um algoritmo mais refinado pode reduzir o número de pares testados para algo como (2.249.727 x 98.451)/(32x32), ou seja, 1024 vezes menor.

De fato, os algoritmos existentes procuram utilizar, de alguma forma, informações geográficas para, no mínimo, evitar testes desnecessários com pares de objetos, para os quais pode-se perceber, *a priori*, que não atendem ao predicado espacial, por estarem, por exemplo, muito distantes um do outro (JACOX, 2003). Também são utilizadas estruturas de armazenamento, em arquivo, e de dados em memória especificamente projetadas para tornarem o processamento mais rápido.

Assim, devido à utilidade de SGBDG para muitas organizações e à complexidade da operação de junção espacial, é importante otimizar o desempenho de algoritmos que possam resolvê-la.

## 1.3 Objetivo

O objetivo principal desta pesquisa é estudar a fase de filtragem dos diversos algoritmos de junção espacial já descritos na literatura. A partir deste estudo, pretende-se elaborar fórmulas que expressem o desempenho dos algoritmos. Para validar as fórmulas, foi implementado um ambiente de testes, que permita executar os algoritmos com dados reais e sintéticos. Desta maneira, será possível identificar pontos fortes e fracos de cada algoritmo, bem como os cenários mais adequados para cada um dos algoritmos estudados. A partir da análise e conjunto de testes realizados, pode-se obter regras de ajuste de desempenho.

Também pretende-se propor novos algoritmos, que permitam realizar a junção espacial em menor tempo ou com menos recursos de memória e disco em cenários de processamento específicos.

### 1.3.1 Objetivos específicos

Este trabalho concentra-se em algoritmos para a etapa de filtragem de junções espaciais. Como produtos deste trabalho, espera-se:

- obter fórmulas de desempenho, tanto de operações de E/S quanto de CPU, para os algoritmos da etapa de filtragem;
- definir cenários de utilização dos algoritmos;

[2]Como rios, riachos ou canais.

- melhorar os algoritmos existentes, ajustando seu desempenho para obter o menor tempo de resposta possível em situações específicas;
- propor novos algoritmos que possam reduzir o tempo de resposta em situações específicas;
- propor um otimizador de consultas, baseado em custos, capaz de estimar o melhor algoritmo a ser executado, dada uma operação submetida pelo usuário e um cenário de execução.

## 1.4 Hipóteses

Para alcançar seus objetivos, esta pesquisa pressupõe que é importante para o otimizador de consultas de um SGBDG, que o mesmo possa contar com diferentes algoritmos para realizar a junção espacial e um conjunto de estratégias de seleção destes algoritmos, dependendo do cenário de processamento da consulta a ser executada. Esta visão do processo de otimização de consultas, baseada em custos, já é amplamente adotada em SGBD Relacionais, principalmente, para operações de junção e de seleção da álgebra relacional.

Assim, a hipótese principal é que a análise de desempenho e os testes de execução poderão estabelecer critérios de escolha do melhor algoritmo de junção espacial, dada uma situação específica. A experiência com SGBD Relacionais indica que esta é uma hipótese aceitável, embora, pelo nosso conhecimento, ainda não existam trabalhos publicados que estabeleçam estes critérios para sistemas de banco de dados geográficos ou simplesmente espaciais.

Uma segunda hipótese é de que, após examinar o desempenho dos algoritmos existentes, seja possível propor aperfeiçoamentos a eles, e, inclusive, novos algoritmos para reduzir o tempo de resposta da junção espacial em cenários de processamento específicos. Esta hipótese é apoiada no fato de que, pelo menos até os dias de hoje, não foi estabelecido qualquer limite mínimo de desempenho para algoritmos de junção espacial, nem foram trazidas à luz evidências de que este limite mínimo exista ou já tenha sido atingido por algum algoritmo existente.

## 1.5 Organização do trabalho

No próximo capítulo, os algoritmos existentes na literatura são apresentados e uma análise do número de operações de E/S e complexidade do algoritmo é realizada. No terceiro capítulo, o ambiente de testes é descrito, bem como os resultados obtidos em junções espaciais com conjuntos de dados sintéticos e reais. Um novo algoritmo é apresentado no quarto capítulo, no qual é feita sua análise de desempenho e comparação de tempos de respostas com outros algoritmos. O quinto capítulo aborda a otimização de consultas e apresenta uma abordagem de seleção do algoritmo ótimo, com base em custos, para realizar a etapa de filtragem da junção espacial. O capítulo seis resume as principais contribuições do trabalho e indica trabalhos futuros que poderão ser realizados.

# 2 ALGORITMOS DE JUNÇÃO ESPACIAL

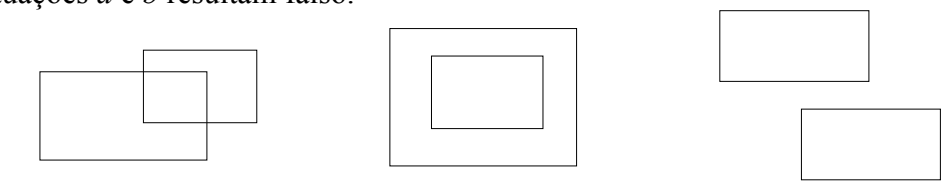
Os algoritmos de junção espacial para a etapa de filtragem tem sido investigados por diversos pesquisadores. Ao longo do capítulo, estes algoritmos são descritos, incluindo uma discussão geral, análise de desempenho e aplicabilidade. Também são apresentadas expressões de custo para os algoritmos, utilizando uma notação uniforme.

Na etapa de filtragem, para cada objeto, há um descritor, composto pelo identificador único (OID), o envelope (MBR) e um ponteiro para a descrição geométrica completa do objeto, armazenada em outro parte da base de dados. O resultado é um conjunto de pares de objetos, que possivelmente atendem ao predicado de junção.

## 2.1 Algoritmo de Intersecção em Memória

Primeiro, apresenta-se o algoritmo para testar predicados espaciais sobre pares de objetos em memória, pois os algoritmos de junção espacial baseiam nele a etapa de processamento em memória.

A figura 2.1 ilustra algumas possíveis situações entre os envelopes de dois objetos. Dependendo do predicado espacial escolhido, cada situação pode resultar em verdadeiro ou falso. Por exemplo, para o predicado de intersecção, as situações *a* e *b* resultam verdadeiro, já para o predicado de disjunção, a situação *c* resulta verdadeiro, enquanto as situações *a* e *b* resultam falso.



a. Intersecção não nula

b. Um retângulo totalmente contido em outro

c. Disjunção

Figura 2.1 : Algumas situações possíveis considerando dois retângulos.

Uma situação mais complexa ocorre quando há um conjunto de vários objetos em memória e deve-se encontrar pares de objetos que se interseccionam. A técnica básica para esta situação chama-se *plane sweep* (DE BERG, 2000), (RIGAUX, 2000). Ela baseia-se em uma linha *L*, chamada *sweep line*, que é deslocada em um dos eixos. É necessário, inicialmente, ordenar os objetos por uma das coordenadas, por exemplo, $x_{min}$. Há, também, a necessidade de uma estrutura auxiliar $\varepsilon$, de objetos ativos, composta pelos objetos que a linha está interseccionando.

A linha percorre a lista ordenada de objetos. Ao encontrar o início de um objeto, é testado se o objeto possui intersecção com cada um dos objetos presentes em ε. Considerando-se que os objetos estejam representados por seus envelopes, isto pode ser feito utilizando-se apenas a projeção dos objetos no eixo das abscissas. Se houver intersecção, o par de objetos é colocado na lista de resposta. O objeto é, então, inserido em $\varepsilon$. Um objeto é retirado de ε quando sua coordenada final, $x_{max}$, for menor que a coordenada inicial do objeto que está sendo inserido na lista.

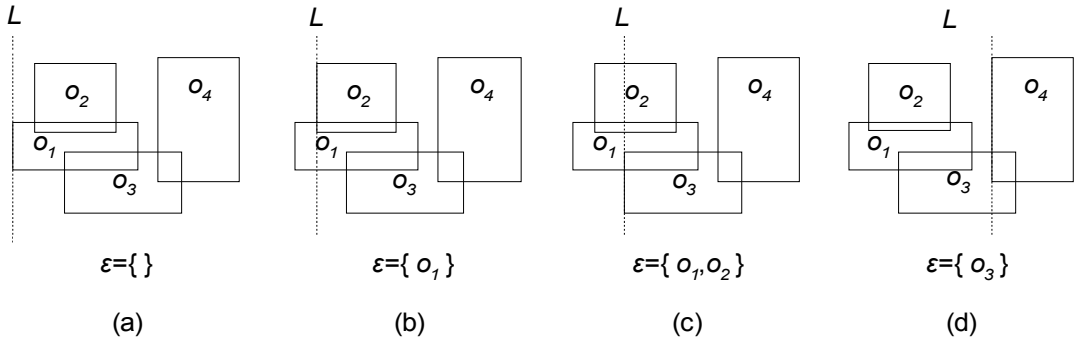



Figura 2.2 : Exemplo de funcionamento do algoritmo de *plane sweep*.

A figura 2.2 ilustra o algoritmo para um conjunto de quatro objetos, representados por seus envelopes. Na posição em 2.2a, a linha encontrou o objeto $o_1$, que é inserido na estrutura $\varepsilon$. O próximo objeto é $o_2$. É testado se há intersecção entre $o_1$, o único elemento mantido na estrutura $\varepsilon$, e o objeto $o_2$. Como a resposta é verdadeira, o par ($o_1$, $o_2$) é colocado na lista de resposta e o objeto $o_2$ é inserido na estrutura $\varepsilon$. Na figura 2.2c, o objeto $o_3$ é encontrado, e a intersecção de $o_3$ com $o_1$ e de $o_3$ com $o_2$ é verificada. Apenas o par $(o_1, o_3)$ é acrescentado ao conjunto de resposta. Após, objeto $o_3$ é inserido na estrutura $\varepsilon$. Em 2.2d, a *sweep line* se posiciona na borda inicial do objeto $o_4$. Neste ponto, os objetos $o_1$ e $o_2$ podem ser retirados da estrutura $\varepsilon$, pois seu $x_{max}$ é maior que $x_{min}$ do objeto $o_4$. Na estrutura de ativos permanece apenas o objeto $o_3$, cuja intersecção com $o_4$ é verificada. O par $(o_3, o_4)$ é incluído na lista de resposta.

A complexidade do *plane sweep* é da ordem de $O(k + n \log n)$, sendo $n$ o número de retângulos e $k$ o número de intersecções (RIGAUX, 2000). O pior caso deste algoritmo ocorre se todos objetos interseccionam todos objetos, resultando em $k = n^2$, e um desempenho da ordem de $O(n^2)$. Esta situação é improvável. Porém, se os objetos forem maiores, $k$ pode se tornar um fator preponderante. Por exemplo, se aplicarmos o algoritmo para localizarmos pares de objetos a uma distância menor que $d$, os envelopes dos objetos são expandidos em $d$ unidades, em todos eixos, como mostra a figura 2.3. Neste tipo de predicado, quanto maior a distância $d$, maior será o número de intersecções ($k$).

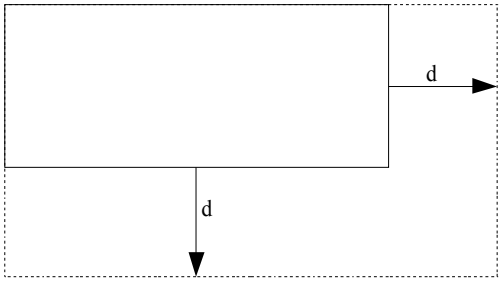



Figura 2.3 : Alteração do envelope para processar predicado de distância.

Outro ponto crítico do algoritmo é a estrutura dinâmica utilizada para os objetos ativos, que pode influenciar de maneira decisiva no tempo de resposta, como demonstrado por Koziara e Bicanic (2005), para 4 diferentes estruturas de dados.

Esta expressão de ordem de complexidade utiliza o número de intersecções, desconsiderando o número de comparações, $c$, realizadas entre objetos, sendo $c \geqslant k$ . A figura 2.4 mostra uma situação onde não há intersecções, ou seja, $k=0$, mas seriam realizadas 6 comparações entre objetos: $(o_2,o_1)$, $(o_3,o_2)$, $(o_3,o_1)$, $(o_4,o_3)$, $(o_4,o_2)$ e $(o_4,o_1)$, pois sempre que o início de um objeto é encontrado, os demais ainda estão na lista de ativos.

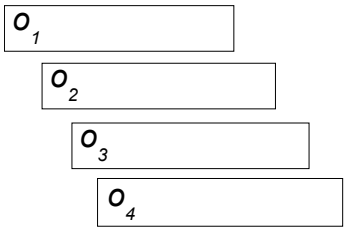



Figura 2.4 : Conjunto de objetos onde $c > k$.

Assim, ao longo deste trabalho será utilizado o número de comparações realizadas, por refletir melhor a complexidade do algoritmo. Considerando que o número de objetos é previamente conhecido, resta definir o valor de $c$. Tomando por base os trabalhos de Belussi (2004) e Das (2004), considerando os objetos uniformemente distribuídos no espaço e os objetos ordenados por suas coordenadas $x_{min}$, $c$ pode ser estimado por

$$c = s_x n \,, \tag{1}$$

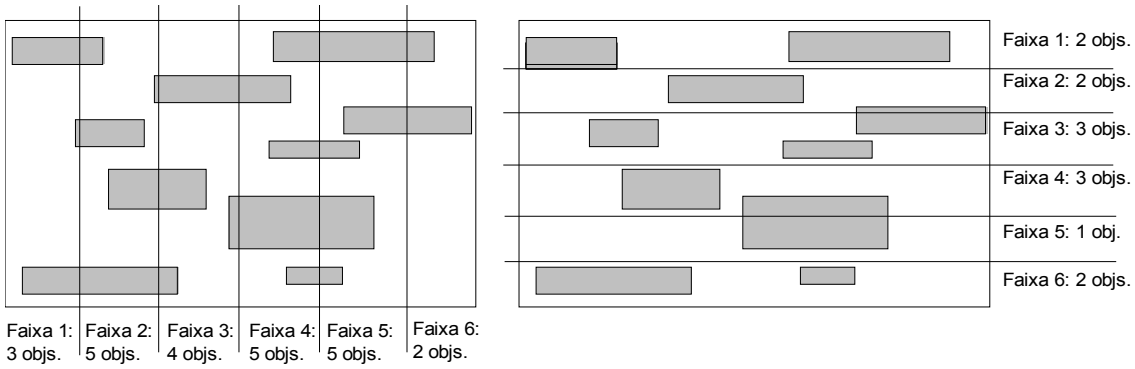
sendo $s_x$ o tamanho médio dos objetos em um espaço normalizado, isto é, entre 0 e 1. Como a técnica de *plane sweep* também funciona se os objetos forem ordenados por uma coordenada do eixo $y$, por exemplo, $y_{min}$, a estimativa deve ser alterada para

$$c = s_y n \,. \tag{2}$$

Para estimar $c$ para conjuntos reais, com objetos distribuídos de maneira não-uniforme, o espaço pode ser dividido em faixas, como mostrado na figura 2.5. Para cada faixa, o número de objetos pode ser contado, definindo um histograma da distribuição de objetos no espaço. A estimativa de $c$ pode ser feita pela expressão (3), onde $\Phi$ representa o número de faixas, $n_i$ o número de objetos em cada faixa.

$$c = \sum_{i=1}^{\Phi} n_i \,. \tag{3}$$

Uma observação rápida da expressão (3) pode conduzir a conclusão que o tamanho dos objetos não influencia na estimativa. Porém, devido a existência de objetos transpassantes, que são contados duas ou mais vezes, de acordo com o número de faixas que interseccionarem, o número de objetos em cada faixa é alterado. A figura 2.5 mostra o exemplo de um conjunto de objetos dividido em 6 faixas. Como pode ser visto, o formato dos objetos modifica o número de objetos por faixa. Dividindo o espaço faixas verticais, o total de objetos é 24, enquanto com faixas horizontais, o total é 13, devido ao formato alongado dos objetos.




(a) Faixas verticais

(b) Faixas horizontais

Figura 2.5 : Exemplo de subdivisão do espaço por faixas verticais e horizontais.

Alguns algoritmos de junção espacial dividem o espaço em faixas para evitar que objetos alinhados, ativos em uma determinada posição da *sweep line*, mas que estão distantes, não possuindo intersecção, sejam verificados. O algoritmo é alterado para manter uma lista de objetos ativos para cada faixa. Objetos transpassantes são replicados em cada faixa que interseccionam, provocando o aumento de objetos manipulados. Considerando $s_y$ o tamanho médio dos objetos, $s_y^{Strip}$ o tamanho das faixas na coordenada *y*, o fator de replicação *r* pode ser aproximado por

$$r = \frac{(s_Y^{Strip} - s_y)}{s_y^{Strip}} + 1$$

Em um espaço dividido em faixas de mesmo tamanho, ou seja, uma divisão regular do espaço, o número de pares verificados é reduzido para

$$c_{Reg} = \frac{(r(n^A + n^B)(s_x^A + s_x^B))}{\Phi} \quad (4)$$

Se a divisão do espaço for irregular, pode-se calcular *c* utilizando o tamanho médio das faixas,

$$c_{Irreg} = \frac{c}{\overline{s}_y^{Strip}} \quad (5)$$

A técnica de *plane-sweep* pode ser adaptada de diversas maneiras, por exemplo, para manipular objetos pertencentes a dois conjuntos diferentes, útil para a junção espacial, ou para resolver a intersecção entre polígonos complexos, necessário na fase de refinamento. Estas adaptações serão vistas a seguir, pois algoritmos de junção espacial diferentes conduzem a adaptações diferentes.

A tabela 2.1 descreve o significado de todos os símbolos utilizados ao longo deste capítulo. Em geral, o superscrito indica o conjunto ao qual a variável se refere, enquanto o subscrito, ao eixo (*x* ou *y*), ou algum aspecto particular da variável.

.

Tabela 2.1 : Significado dos símbolos utilizados em expressões de custo.

| ***Símbolo*** | ***Significado*** |
|---|---|
| | *Referentes aos conjuntos de dados A e B* |
| $n, n^A, n^B$ | Número de objetos em um conjunto indeterminado, ou nos conjuntos *A* e *B,* respectivamente. |
| $s_x^A, s_y^A, s_x^B, s_y^B$ | Tamanho médio dos objetos, em cada eixo, em cada conjunto. |
| $b, b^A, b^B$ | Número de blocos em disco ocupados por um conjunto indeterminado, ou pelos conjuntos *A* e *B,* respectivamente. |
| $f, f^A, f^B$ | Fator de bloco de um conjunto qualquer, ou dos conjuntos *A* e *B*, respectivamente. |
| $n_{Rep}^A, n_{Rep}^B$ | Número de objetos nos conjuntos *A* e *B,* após replicação |
| | *Referentes à árvores-R, que indexam espacialmente os conjuntos A e B* |
| $h, h^A, h^B$ | Altura de uma árvore-R qualquer, ou de uma árvore-R que indexa os conjuntos A ou B, respectivamente |
| $g, g^A, g^B$ | Grau de ocupação médio dos nodos de uma árvore-R |
| $Fanout, Fanout_A, Fanout_B$ | *Fanout* dos nodos de uma árvore-R |
| $n_i^A, n_i^B$ | Número de nodos de uma árvore-R no nível *i*. |
| $n_R, n_R^A, n_R^B$ | Número total de nodos da árvore. |
| $s_{x,i}^A, s_{y,i}^A, s_{x,i}^B, s_{y,i}^B$ | Tamanho médio dos nodos no nível i. |
| | *Referentes a outros aspectos* |
| $M_b$ | Capacidade da memória em número de blocos |
| $M_o$ | Capacidade da memória em número de objetos |
| $k$ | Número de pares de objetos que atendem ao predicado espacial |
| $c, c_{Reg}, c_{Irreg}$ | Número de pares de objetos que são testados pelo algoritmo no caso geral, se houver divisão regular do espaço, ou se a divisão for irregular. |
| $\Phi, \Phi_H, \Phi_V$ | Número de faixas em que o espaço esta dividido. Se o sentido for significativo, pode-se diferenciar entre horizontais ou verticais. |
| $s_x^{Strips}, s_y^{Strips}$ | Tamanho das faixas em cada eixo. |
| $P$ | Número de partições em que os objetos, de ambos conjuntos, são divididos. |
| $\psi$ | Número de partes em que um conjunto é dividido no algoritmo de ordenação externa. |
| $\rho$ | Probabilidade de um certo bloco ser encontrado no *buffer* em memória. |
| $q$ | Número de listas |
| *Rep* | Número de repetições de um algoritmo. |

## 2.2 Classificação dos Algoritmos de Junção Espacial

Os algoritmos para a etapa de filtragem podem ser divididos quanto a estratégia principal de acesso a dados, definindo os seguintes grupos: arquivos sequenciais não-indexados, arquivos ordenados, arquivos indexados e apenas um arquivo indexado. A figura 2.6 lista os principais algoritmos existentes, adequadamente agrupados. Na sequência, cada grupo de algoritmos é detalhado.

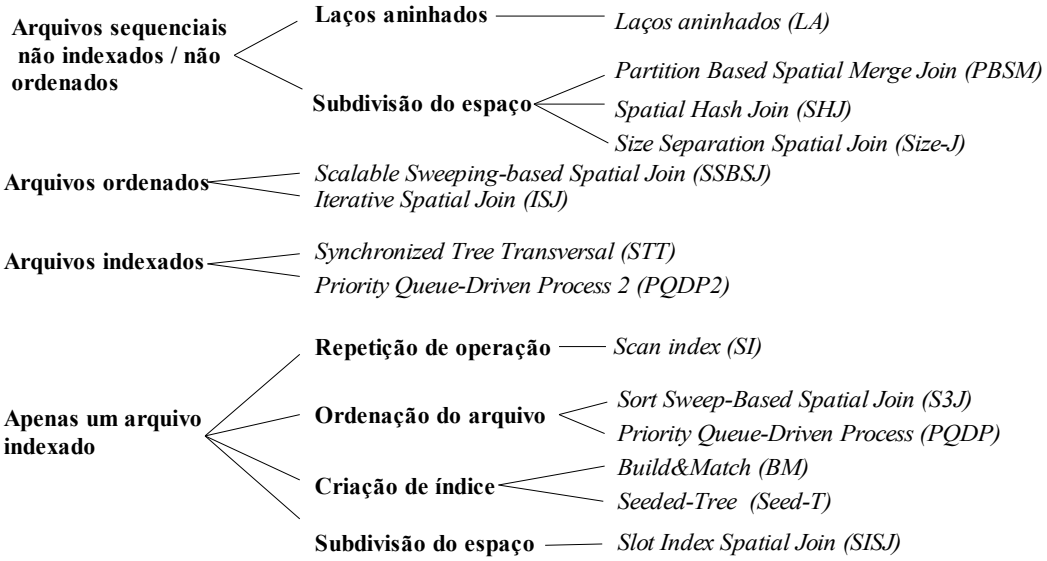



Figura 2.6 : Classificação de algoritmos de junção espacial com base na estrutura de arquivos utilizada.

Tabela 2.2 : Existência prévia de expressões de custo para algoritmos de junção espacial.

| | ***Número de acessos a disco*** | ***Processamento pela CPU*** |
|---|---|---|
| PBSM | Sim | Sim |
| SHJ | Sim | **Não** |
| Size-J | Sim | Sim |
| LA | Sim | Sim |
| SSBSJ | Sim | Sim |
| ISJ | **Não** | **Não** |
| STT | Sim | **Não** |
| PQDP2 | **Não** | **Não** |
| Scan-Index | Sim | **Não** |
| SSBSJ | Sim | Sim |
| PQDP | Sim | **Não** |
| Build&Match | Sim | **Não** |
| Sort&Match | Sim | **Não** |
| Seeded-Tree | Sim | **Não** |
| SISJ | Sim | **Não** |

Uma característica importante da junção espacial é que o processamento de dados, na memória principal do computador pode ser responsável por até 95% do tempo de resposta deste tipo de operação (SUN, 2003). Este fato justifica e exige que o estudo da eficiência de algoritmos de junção espacial não se restrinja apenas ao número de acessos a disco, como é habitual em junções para bancos de dados relacionais, mas também inclua a complexidade do processamento na CPU. A tabela 2.2 indica se há, na literatura, expressões de custo que possam ser utilizadas por um otimizador de consultas para escolher a melhor opção de algoritmo de junção espacial.

Como se pode perceber, ainda não há expressões de desempenho para muitos dos algoritmos de junção espacial publicados. Em geral, ao apresentar um algoritmo, seus autores realizam comparações através da execução de testes em ambientes controlados. Para permitir que o otimizador realize uma escolha ampla, foram obtidas expressões de custo para todos algoritmos ou adaptadas as expressões existentes a uma notação comum, sendo esta umas das contribuições deste trabalho. Para cada grupo de algoritmos, aquele que a análise indicava como sendo mais rápido foi escolhido. Nas descrições dos algoritmos, realizada a seguir, apenas para o algoritmo implementado as expressões de custo são apresentadas. As expressões de custo para os demais algoritmos estão no anexo A. Em todas expressões não estão incluídas as operações de E/S necessárias para gravar o conjunto resposta, já que é uma parcela idêntica para todos algoritmos.

## 2.3 Algoritmos sequenciais não-indexados

Este grupo de algoritmos pode ser subdividido em dois: laços aninhados e subdivisão de espaço. O algoritmo de laços aninhados é uma simples adaptação de tradicional algoritmo para junção relacional, apenas incluindo o *plane-sweep* para verificar pares de objetos em memória. Seu desempenho, tanto em operações de disco quanto número de pares comparados é reconhecidamente ruim (KORTH, 2005).

Já os algoritmos baseados em subdivisão do espaço dividem o universo em subespaços e procuram resolver o problema da junção para os objetos pertencentes a cada um dos subespaços, podendo ser classificado como um método de divisão e conquista. Em princípio, a divisão é realizada durante a junção, mas se já houver uma anterior, esta pode ser aproveitada.

### 2.3.1 *Partition Based Spatial Merge Join*

Patel e DeWitt (1996) propuseram o algoritmo chamado *Partition Based Spatial Merge Join* (PBSM), baseado na subdivisão do espaço em partições. O número de partições é calculado pela função

$$P = \lceil \frac{(n^A + n^B)}{M_o} \rceil$$

Para cada partição é criado um arquivo inicialmente vazio.

O algoritmo utiliza uma grade regular para dividir o espaço em um certo número de células, maior que o número de partições. Cada célula é alocada a uma partição apenas, através de alguma função, como *round-robin* ou *hash*. A figura 2.7 mostra a subdivisão e a definição de partições, quando há uma grade regular (4 x 4), três partições e atribuição de células à partições por *round-robin*.

| | | | |
|---|---|---|---|
| Célula 0<br>Part. 0 | Célula 1<br>Part. 1 | Célula 2<br>Part. 2 | Célula 3<br>Part. 0 |
| Célula 4<br>Part. 1 | Célula 5<br>Part. 2 | Célula 6<br>Part. 0 | Célula 7<br>Part. 1 |
| Célula 8<br>Part. 2 | Célula 9<br>Part. 0 | Célula 10<br>Part. 1 | Célula 11<br>Part. 2 |
| Célula 12<br>Part. 0 | Célula 13<br>Part. 1 | Célula 14<br>Part. 2 | Célula 15<br>Part. 0 |

Figura 2.7 : Alocação de células a partições no PBSM.

Os objetos são alocados a todas células que interseccionarem, sendo inseridos em um ou mais arquivos de partições. Após, cada arquivo de partições é lido para memória, seus objetos são ordenados por uma das coordenadas e é realizado o algoritmo de *plane sweep* para testar o predicado de junção espacial. Uma situação especial pode ocorrer caso o arquivo de uma partição seja maior que a memória disponível. Então, apenas este arquivo é reparticionado, antes de realizar o *plane sweep.*

Com a duplicação de objetos na fase de partição, é necessário eliminar duplicatas no arquivo de resposta. Dittrich e Seeger (DITTRICH, 2000) propuseram uma técnica chamada *Reference Point Method* para evitar a existência de pares duplicados no conjunto resposta. O par de objetos é incluído no conjunto de resposta somente se o ponto superior esquerdo da intersecção estiver dentro da partição que está sendo considerada. Em caso contrário, o par não é incluído no conjunto de resposta, pois será, novamente, identificado durante o processamento dos mesmos objetos em outra partição. O conjunto de respostas é o final, sem replicações de pares de objetos.

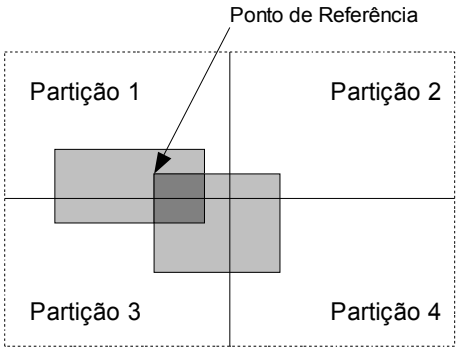



Figura 2.8 : *Reference Point Method*

A figura 2.8 mostra um par de objetos em um espaço dividido em quatro células, cada uma representando uma partição diferente. O par será identificado duas vezes, durante o processamento das partições 1 e 3, mas apenas durante o processamento da partição 1 será incluído no conjunto resposta.

A figura 2.9 mostra o algoritmo completo, já considerando a utilização do *Reference Point Method*. O trabalho original previa a ordenação do conjunto de pares de objeto, para eliminar duplicatas, resultando em tempos de execução mais altos.

```
Procedure PBSM(A,B: conjunto de objetos espaciais)
i, npart: integer
a, b: objeto espacial
c_cel: conjunto de células
c: célula
Begin
npart = calculo do número de partições
For each a in A do
    c_cel = conjunto de células que o objeto a intersecciona
    For each c in c_cel
           insere (c,a)
    End-for
End-for
For each b in B do
    c_cel = conjunto de células que o objeto a intersecciona
    For each c in c_cel
           insere (c,b)
    End-for
End-for
For i=1 to npart do
    Ler objetos de A e B pertencentes a partição i
    Aplicar predicado de junção aos objetos
    For each par de objetos que predicado espacial = true do
       Aplicar Reference Point Method
       If Resultado = true
          then incluir no conjunto de respostas
       End-if
    End-for
End-For
End-proc
```

Figura 2.9 : Algoritmo do *Partition Based Spatial Merge Join* (PBSM).

O número de operações de E/S é influenciado pelo número de objetos replicados. Na etapa de particionamento, todos objetos são lidos para memória e escritos em uma ou mais partições. O fator de replicação, *r*, pode ser calculado como o número de objetos, incluindo as réplicas, dividido pelo número original de objetos, ou seja, $r=(n^{A}_{Rep}+n^{B}_{Rep})/(n_A+n_B)$ resultando um valor igual ou superior a 1. A porcentagem de partições com *overflow* é expressa por *o*, sendo um valor entre 0 e 1. Os objetos pertencentes a estas partições devem ser lidos novamente para memória e gravados em novos arquivos de partições. Novamente, réplicas podem surgir. Para realizar o teste do predicado espacial, as partições são lidas para memória, resultando como número de operações de E/S

$$disco_{PBSM}=(2\,o\,r^2+2r+1)(b_a+b_b) \qquad (6)$$

A expressão demonstra que o tamanho da memória disponível não afeta o número de operações de E/S. Objetos maiores tendem a aumentar o fator de replicação, assim como uma grade regular com maior número de células. O número de partições que sofrem *overflow* é importante, mas difícil de ser previsto, pois é influenciado pela distribuição dos objetos no espaço, número de partições e o critério de alocação.

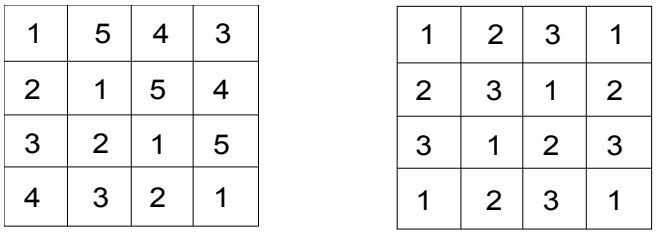



Figura 2.10 : Alocação de células à partições com 5 ou 3 partições.

O desempenho do algoritmo, em termos de processamento, depende do número de objetos. Considerando que todas partições possuem o mesmo número de objetos, incluindo réplicas, o algoritmo *plane-sweep*, em cada partição, manipula $r(n^A+n^B)/P$ objetos. Com o surgimento de réplicas devido à divisão do espaço, o número de pares verificados, *c*, é alterado, uma vez que que um objeto pode ser comparado com as duas cópias. Há duas situações: se o número de partições for igual ou maior que o número de linhas, *c* é dividido pelo número de linhas; se o número de partições for menor, *c* é dividido pelo número médio de células de uma mesma coluna que são alocadas a uma partição. A figura 2.10 ilustra estas duas situações, a esquerda (2.10a) há 4 linhas por coluna e 5 partições, à direita (2.10b), as mesmas 4 linhas por coluna, mas apenas 3 partições. Neste segundo caso, algumas partições têm duas células da mesma coluna alocadas a ela. Portanto, o valor de *c*, quando há partições, sendo $\Phi_H$ o número de linhas horizontais da grade regular, é

$$c_{PBSM} = \frac{c}{\Phi_H}, P > \Phi_H$$

$$c_{PBSM} = \frac{c}{P}, P \leq \Phi_H$$

Assim, em cada partição, o desempenho é da ordem de $O(c_{PBSM}/P + r(n^A+n^B)/P \log(r(n^A+n^B)/P))$. O algoritmo de *plane-sweep* é realizado para todas partições, resultando em um desempenho total na ordem de

$$cpu_{PBSM} = O(c_{PBSM} + r(n^A+n^B) \log \frac{r(n^A+n^B)}{P}) \quad (7)$$

Portanto, o algoritmo é influenciado pelo número de partições, obtido a partir da memória disponível. Não requer nenhuma estrutura prévia dos arquivos, pois organiza os arquivos de partição diretamente a partir dos conjuntos de dados. O esforço de construção das partições não é reaproveitado em outras operações.

### 2.3.2 Demais algoritmos

Lo e Ravishankar (1996) propuseram o algoritmo *Spatial Hash Join* (SHJ) para realizar a operação de junção espacial. A divisão do espaço é realizada de maneira irregular, a partir dos objetos do conjunto *A*. O método de cálculo utilizado é semelhante ao que determina o número de níveis que devem ser copiados para uma *Seeded-Tree*, também proposta pelos autores (LO e RAVISHANKAR, 1995). O número de partições é bastante alto, por exemplo, para dois conjuntos de 100.000 objetos, há mais de 3.000 partições, dependendo da memória disponível.

Inicialmente, os objetos do conjunto *A* são alocados às partições. Cada objeto é alocado à partição em que está contido ou a mais próxima. Caso necessário, os limites

da partição são alterados para incluir totalmente o objeto. Após, procede-se a alocação dos objetos do conjunto *B*. As células/partições já definidas para *A* são mantidas, inclusive as suas geometrias, e cada objeto de *B* é inserido em todas as células que interseccionar.

Na etapa de junção, cada partição, contendo objetos dos dois conjuntos, é trazida para memória. Os elementos de cada partição são verificados de acordo com o predicado da junção. Pode ocorrer do número de objetos exceder a disponibilidade de memória, obrigando a um reparticionamento, mas esta é uma situação improvável, devido ao grande número de partições que é utilizado.

Da mesma forma que no PBSM, o número maior de partições favorece o algoritmo, porém, pode produzir um número de réplicas muito alto, resultando uma questão de otimização difícil, pois está vinculada ao tamanho médio dos objetos, à distribuição dos objetos de *A*, que define inicialmente as partições e ao tamanho final de cada célula de partição. Em geral, o número de operações de disco é menor que o PBSM. Porém, a etapa de alocação dos objetos do conjunto *B* em células distribuídas irregularmente no espaço é cara, necessitando de estruturas de dados auxiliares, como árvores especiais.

Outro algoritmo deste grupo denomina-se *Size Separation Spatial Join* , tendo sido proposto por Koudas e Sevcik (1997). O algoritmo utiliza uma *Filter-Tree* (KOUDAS, 1998) para subdividir o espaço, sem construí-la. A divisão resultante é um conjunto de grades regulares, uma para cada nível da árvore, aumentando o número de linhas e colunas da raiz para as folhas. Cada objeto é inserido no primeiro nível em que uma linha passar sobre o objeto. Desta maneira, objetos grandes são alocados nos níveis mais altos da árvore, e objetos menores, nos níveis mais próximos às folhas, embora objetos pequenos possam ser alocados nos níveis mais altos da árvore por, acidentalmente, cruzarem uma linha de divisão.

O algoritmo inicia calculando dois valores para cada objeto:

- Valor Hilbert do ponto central do envelope.
- Nível da árvore onde o objeto será inserido.

Para cada nível identificado, dois arquivos são criados, um para objetos de *A*, outro para objetos de *B*. Para cada objeto é montado um registro contendo as coordenadas de seu envelope, o valor Hilbert associado a ele é um ponteiro para sua descrição geométrica completa. Este registro é gravado no arquivo correspondente. A seguir, cada arquivo é ordenado pelo valor Hilbert.

A junção é realizada percorrendo, seqüencialmente, todos os arquivos de todos os níveis. Cada bloco de disco de cada arquivo é lido uma única vez e a memória deverá ser suficiente para conter um conjunto destes blocos em um mesmo instante de tempo.

Além da junção espacial, um ponto crítico do algoritmo é o cálculo do valor Hilbert, realizado por algoritmos cuja complexidade varia de acordo com a precisão escolhida, expressa pelo número de bits. O algoritmo realiza, no mínimo, cinco operações de E/S para cada objeto, sendo o número de operações de E/S mais alto deste grupo.

## 2.4 Algoritmos baseados em arquivos ordenados

A primeira etapa do algoritmo de *plane-sweep* requer a ordenação dos objetos por uma das coordenadas. Este grupo de algoritmos ordena cada conjunto de objetos

separadamente, antes de realizar o *plane-sweep,* constituindo-se, assim, em uma adaptação do algoritmo de *sort-merge* utilizado para a junção de relações.

Para a ordenação, se o conjunto for menor que a memória disponível, basta colocá-lo em memória, ordená-lo e gravá-lo. Porém, se o conjunto for maior, será necessário realizar a ordenação externa do arquivo de objetos.

No melhor caso, ordenação interna dos conjuntos *A* e *B*, o algoritmo realiza uma leitura para carregar os objetos para memória e outra para gravá-los, devidamente ordenados, resultando em *2b* operações de E/S. No pior caso, é necessária a ordenação externa, realizando diversas leituras e escritas, e o número de operações de E/S é

$$2b \times \lceil \log_{M_b}(b) \rceil \tag{8}$$

Na segunda etapa do algoritmo, os objetos são lidos para o processamento em memória apenas uma única vez. Assim, o total de operações de E/S, no melhor caso, é

$$disco_{ORD} = 3(b^A + b^B) \tag{9}$$

No pior caso, o algoritmo realiza

$$disco_{ORD} = (b^A + b^B) + 2b^A \times \lceil \log_{M_b}(b^A) \rceil + 2b^B \times \lceil \log_{M_b}(b^B) \rceil \tag{10}$$

Se os arquivos já estiverem ordenados, o número de operações de E/S será mínimo, pois bastará ler os dois conjuntos em uma passagem para processar todos objetos.

A ordem de desempenho do algoritmo, considerando as operações em memória, também depende da forma de ordenação possível. Considerando um algoritmo adequado, como *heapsort*, tem-se *O(n log n)*, sendo *n* o número de objetos ordenados.

Na etapa de verificação do predicado espacial pelo algoritmo de *plane-sweep*, que combina objetos dos dois conjuntos, o desempenho será

$$O(c + (n_A + n_B)\log(n_A + n_B))$$

Portanto, o desempenho do algoritmo completo, no melhor caso, é da ordem de

$$cpu_{ORD} = O(n_A \log n_A + n_B \log n_B + c + (n_A + n_B)\log(n_A + n_B)),$$

que pode ser simplificado para

$$cpu_{ORD} = O(c + (n_A + n_B)\log(n_A + n_B)) \tag{11}$$

Já a análise da ordenação externa requer, também, considerar duas etapas do algoritmo (AZEREDO, 1996)[3]: na primeira, o arquivo é dividido em $\psi = \lceil n/Mo \rceil$ partes, que são ordenadas de maneira independente. Cada parte possui *n/p* objetos, que ordenados em memória, resulta em um desempenho da ordem de $O(M_o \log M_o)$. Ao repetir a ordenação para cada uma das partes, o desempenho final é da ordem de $O(n \log M_o)$. A segunda etapa da ordenação externa, segundo Azeredo (1996), possui uma ordem de desempenho de $O(n \log \psi)$ comparações. Somando as duas etapas da ordenação externa, obtém-se o desempenho da ordem de

$$O(n \log(M_o \psi)).$$

[3]Esta análise considera que a ordenação externa pode ser realizada em apenas duas etapas. Se o tamanho do conjunto for muito maior que a memória disponível, podem ser necessárias mais de duas passagens para ordenar o conjunto em um certo número de partes, que possam ser mantidas em memória na segunda etapa do algoritmo.

Assim, o desempenho do algoritmo de junção completo, no pior caso, é representado pela soma de

$$cpu_{ORD} = O(n^A \log(M_o \psi^A) + n^B \log(M_o \psi^B) + c + (n^A + n^B)\log(n^A + n^B)),$$

que também pode ser simplificado para

$$cpu_{ORD} = O(c + (n^A + n^B)\log(n^A + n^B)) \quad (12)$$

Na literatura, há dois algoritmos para este grupo, que se diferenciam no tratamento dado para o caso do número de objetos ativos exceder a capacidade de memória. Arge, no algoritmo *Scalable Sweeping-based Spatial Join*, propõe dividir o espaço em faixas (*strips*) (ARGE,1998). Se a ordenação dos objetos for realizada pelo eixo *x*, as faixas dividem o espaço pelo eixo *y*. Os objetos ativos, que devem ser mantidos em memória, são alocados cada um a sua respectiva faixa, de acordo com sua localização no espaço. Cada faixa é processada separadamente, através de uma chamada iterativa ao algoritmo, identificando os pares de objetos que atendem ao predicado de junção espacial. Se não houver memória suficiente para manter todas as faixas, uma delas é gravada em disco e processada posteriormente.

Jacox e Samet (JACOX, 2003), no algoritmo *Iterative Spatial Join*, ao identificar a impossibilidade de inserir um objeto na lista de ativos, gravam-no em um arquivo auxiliar, de objetos a serem processados. Quando a *sweep line* atinge o limite final do espaço, o algoritmo de *plane-sweep* é reinicializado, processando apenas os objetos do arquivo auxiliar. Segundo os próprios autores, em testes realizados com conjuntos de objetos reais e um tamanho de memória razoável, em nenhum caso ocorreu esta situação limite. Como os dois algoritmos são idênticos, alterando apenas a forma de tratamento no caso de *overflow* de memória, o número de operações de acesso a disco e de CPU são, igualmente, idênticos.

Este grupo de algoritmos é mais simples de ser implementado, e não requer nenhuma organização de arquivo específica. A ordenação de cada conjunto pode ser em memória ou externa, dependendo da cardinalidade dos conjuntos de dados a serem processados. Se os dois conjuntos já estiverem ordenados, o algoritmo é uma escolha ótima, pois realizará um mínimo de operações de E/S.

## 2.5 Algoritmos baseados em arquivos indexados

O terceiro grupo de algoritmos baseia-se em árvores espaciais para realizar a operação de junção. Uma árvore espacial é uma estrutura de dados utilizada para indexar objetos espaciais, sendo árvores-R as de uso mais amplo, merecendo maior atenção.

### 2.5.1 Árvores-R

A árvore-R, proposta por Guttman (1984), é uma estrutura hierárquica e dinâmica, projetada para manipular objetos *n*-dimensionais. A árvore-R é balanceada e permite a inserção e remoção de objetos, que são representados por seus envelopes.

Os nodos folhas são compostos por um conjunto de entradas do tipo *(oid, env)*, onde *oid* representa o identificador do objeto, que permite a ligação com sua descrição detalhada, e *env*, o envelope do objeto.

Os nodos intermediários e o nodo raiz são compostos por entradas do tipo *(pont, env)*, onde *pont* é um ponteiro para o nodo filho, e *env*, o envelope que abrange todos os

envelopes do respectivo nodo filho. No mínimo, metade das entradas existentes em um nodo, intermediário ou folha, devem estar ocupadas, exceto o nodo raiz.

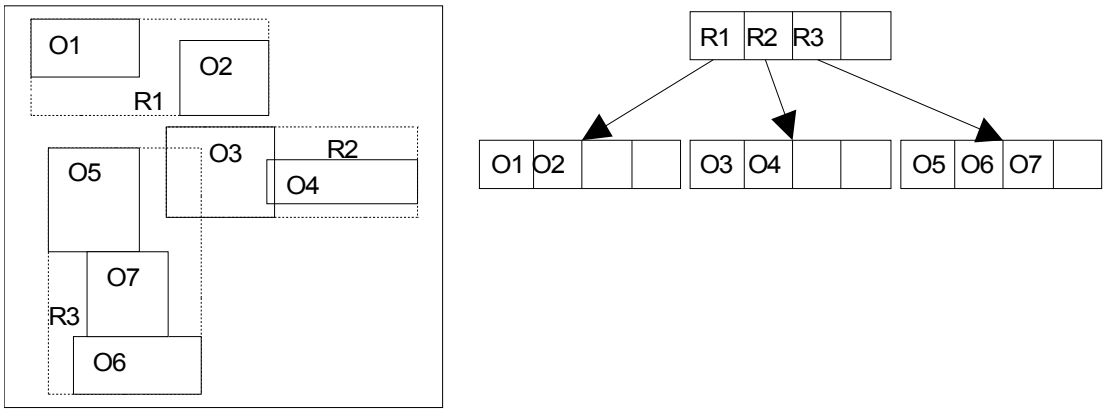



Figura 2.11 : Exemplo de uma árvore-R.

A figura 2.11 mostra um exemplo de árvore para um espaço bidimensional, com 4 entradas em cada nodo, estando ocupadas, no mínimo, metade delas. Envelopes de objetos são representados por linhas contínuas e envelopes dos nodos por linhas pontilhadas.

A inserção do envelope de um objeto é realizada no nodo folha cujo envelope sofre o menor aumento de área. Se o nodo folha já tiver com todas as entradas preenchidas, ocorre uma divisão (*split*) do nodo, sendo criado um novo nodo folha. Neste caso, os objetos existentes no nodo original, mais o novo objeto, são divididos entre o nodo original e novo nodo folha. O critério de divisão dos objetos procura reduzir a extensão dos envelopes dos nodos folha. A divisão deve, ainda, ser propagada para os níveis superiores, criando novas entradas, havendo a possibilidade, inclusive, de criar um novo nodo raiz.

Para remover um objeto, o primeiro passo é localizá-lo em um nodo folha. Neste nodo, sua entrada é removida e o envelope do nodo é recalculado. A alteração do envelope, se houver, é propagada para os nodos superiores. Ao remover um objeto de um nodo folha, pode ocorrer que menos da metade das entradas existentes permaneçam ocupadas. Neste caso, o nodo é retirado da árvore e os objetos restantes são reinseridos. A remoção de um nodo implica na retirada de uma entrada nos níveis superiores.

A altura da árvore-R é calculada pela fórmula

$h = 1 + \lceil \log_{g.Fanout} \frac{n}{(g.Fanout)} \rceil$, onde $g$ representa o grau de ocupação médio dos nodos e *Fanout* o número de entradas de um nodo da árvore-R.

Segundo Theodoridis (1996, 2000), no nível $i$, o número de nodos é

$$n_i = \lceil \frac{n}{(g.Fanout)^i} \rceil$$

O total de nodos da árvore-R é dado pelo somatório

$$n_R = \sum_{i=1}^{h} n_i$$

Há diversas variações de árvore-R descritas na literatura. O tutorial de Gaede (1998) aborda várias delas. Por exemplo, a árvore-R* (BECKMANN,1990) possui as mesmas características estruturais da árvore-R e tem por objetivo reduzir a extensão das intersecções existentes entre os envelopes dos nodos. Como um objeto pode ser inserido em mais de um nodo folha, o critério de escolha favorece aquele que resulta em menor aumento de intersecção com envelopes de outros nodos. Ainda, quando um nodo folha é escolhido para a inserção e todas as suas entradas já estão ocupadas, antes de realizar uma divisão, alguns objetos são removidos e reinseridos em nodos espacialmente próximos, se possuírem entradas disponíveis.

Uma abordagem diferenciada é utilizada por Brakatsoulas (2002), que transforma o problema de encontrar o particionamento ótimo dos objetos de um nodo quando este sofre divisão (*split*), no problema de encontrar grupos (*clusters*) de objetos próximos, utilizando os algoritmos desenvolvidos para definição de *clusters*.

Nesta situação, um nodo pode possuir menos da metade de suas entradas ocupadas, alterando uma regra fundamental na formação de árvores-R. Em testes práticos, os autores determinaram que a divisão de um nodo deve gerar, no máximo, cinco novos nodos. Assim, o grau de ocupação médio não fica muito baixo e reduz o tempo de processamento para definir o agrupamento ótimo. A figura 2.12 mostra diferentes possibilidades de agrupamento, para um pequeno conjunto de objetos.

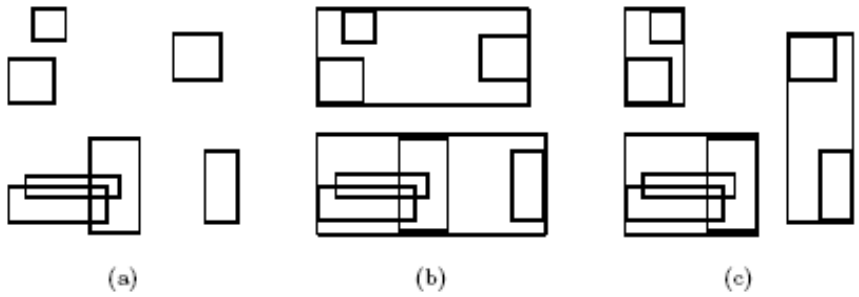



Figura 2.12 : Divisão de um objetos de um nodo (a) em dois (b) ou três (c) grupos.

### 2.5.2 Construção otimizada de árvores-R

A inserção individual de objetos em árvores-R resulta em nodos com grande intersecção e muitas entradas vazias. Aproveitando o fato de que, muitas vezes, já se dispõe de todo conjunto de objetos antes de criar a árvore-R, uma forma de melhorar seu desempenho é realizar a inserção de objetos em massa, de maneira otimizada.

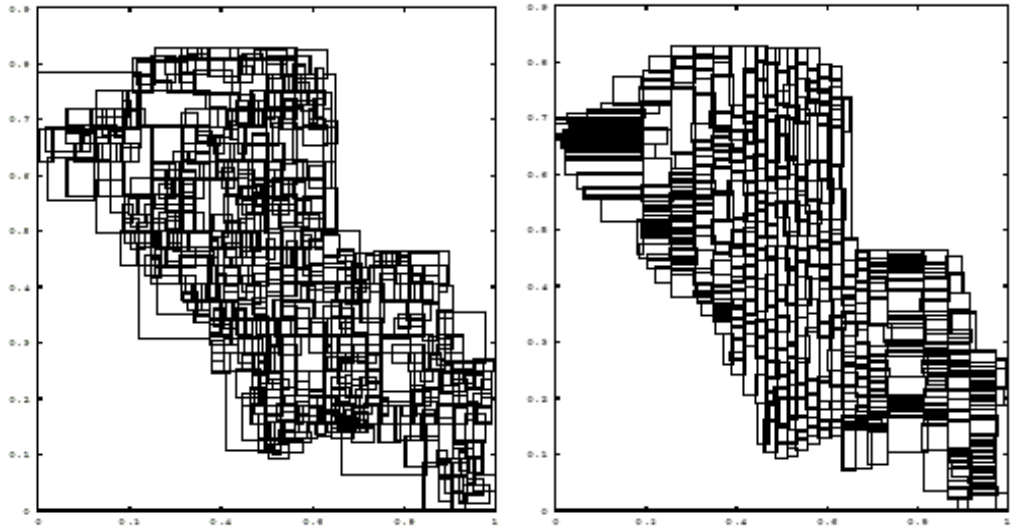
O algoritmo *Sort-Tile-Recursive* (STR) (LEUTENEGGER, 1997)(GARCIA, 1998) inicia pela construção dos nodos folhas, e, progressivamente, os nodos intermediários, até construir o nodo raiz. Considerando que cada nodo mantém um certo número de entradas, os seguintes passos são realizados:

1. Ordenar o conjunto de objetos por uma das coordenadas, e definir $\lceil n/Fanout \rceil$ grupos de objetos contíguos, considerando a proximidade em todas dimensões.
2. Cada grupo de objetos é transformado em um nodo folha da árvore-R e este nodo é gravado no arquivo da árvore-R, que, assim, é construída das folhas para o nodo raiz.
3. A construção de um novo nível é executada. Na primeira vez, a entrada são os nodos folhas, e a saída, nodos intermediários do nível superior às folhas. A construção de novos níveis prossegue, até ser criado o nível do nodo raiz.

A figura 2.13 mostra os envelopes dos nodos folhas obtidos pelo algoritmo para um conjunto de ruas na área de Long Beach/CA. A esquerda, o resultado utilizando o algoritmo de inserção objeto a objeto, à direita, com o algoritmo de inserção em massa. Pode-se observar uma certa regularidade, com pequenas áreas de intersecção entre os nodos folhas.

O número de acessos a disco deste algoritmo é a soma dos acessos realizados na etapa de ordenação, que pode ser externa ou interna, dependendo do número de objetos e da memória disponível, com o número total de nodos da árvore.

A etapa de ordenação já foi examinada na seção 2.4. O número de nodos de uma árvore-R depende do grau de ocupação médio dos nodos, que é otimizado pelo algoritmo STR para valores muito próximos a 100%.



(a) inserção objeto a objeto (b) inserção em massa

Figura 2.13 : Envelopes das folhas de uma árvore-R criada pelos algoritmos de inserção objeto-a-objeto e STR.

Na melhor hipótese, o conjunto de objetos pode ser ordenado em memória, permitindo gravar os nodos folhas. Os nodos intermediários são gravados, e depois, novamente lidos para formar o nível superior. Apenas a raiz não é lida novamente. O número de operações de E/S, para construir uma árvore-R, é

$$disco_{STR} = b + 2n_R - 1 \qquad (13)$$

Este número é muito menor quando comparado ao algoritmo comum de inserção, que requer, no mínimo, percorrer a árvore da raiz até um dos nodos folhas e reescrever este nodo folha, a cada objeto a ser inserido, ou seja, no mínimo, $disco_{INS} = n.h$ . Por exemplo, para uma árvore com *Fanout*=146 (blocos de 4096 bytes), $g \simeq 1$ e um conjunto de 100.000 objetos, obtém-se:

$h = 3,\ n_R = 691$

$disco_{STR} = b + 2 \times 691 - 1 = b + 1381$

$disco_{INS} = b + 100.000 \times 3 = b + 300.000$

Mesmo na hipótese de ordenação externa do arquivo, o número de operações de disco é $disco_{STR} = 2b \times |\log_{M_b}(b)| + 2n_R$, que ainda é menor que o número de operações de E/S para a inserção objeto a objeto.

Em termos de operações de CPU, o algoritmo requer a ordenação interna ou externa dos objetos, também já apresentada na seção 2.4. Os nodos folhas são o produto desta etapa, com a vantagem adicional que os objetos, dentro de um nodo, já estão ordenados por uma das coordenadas, algo útil para o algoritmo de junção. A criação dos nodos de nível superior requer a definição do envelope dos nodos filhos, que é feita verificando os envelopes de cada entrada dos nodos filhos, ou seja, é da ordem de $O(n_R \times Fanout)$, e a ordenação das entradas do nodo que está sendo construído, da ordem de $O(n_R \times Fanout \log Fanout)$.

Na melhor hipótese, de ordenação interna do conjunto de objetos, o desempenho do algoritmo é da ordem de

$cpu_{STR} = O(n \log n + n_R \times Fanout\,(1 + \log Fanout))$,

como $n \gg Fanout$, pode ser simplificado para

$$cpu_{STR} = O(n \log n) \qquad (14)$$

Para a hipótese de ordenação externa, o desempenho é da ordem de

$cpu_{STR} = O(n \log(M_o \psi) + n_R \times Fanout)$,

que, também, pode ser reduzido à

$cpu_{STR} = O(n \log(M_o \psi))$

De qualquer modo, a etapa mais importante do algoritmo é a ordenação dos objetos.

Na inserção objeto a objeto, em cada nodo, o objeto será comparado, na média, com metade das entradas dentro do nodo, resultando uma ordem de desempenho de

$$cpu_{INS} = O(n\, h \frac{Fanout}{2}) \qquad (15)$$

Como $h \frac{Fanout}{2} > \log n$, a partir de (14) e (15), pode-se concluir que $cpu_{STR} < cpu_{INS}$. Portanto, o algoritmo STR apresenta várias vantagens sobre a inserção objeto a objeto: menor número de operações de E/S, menor processamento em memória, os nodos resultantes têm um grau de ocupação média maior e menor intersecção entre os envelopes dos nodos folhas. Ainda, é relativamente simples de implementar, justificando sua utilização sempre que possível.

Outra situação típica é a inserção de um conjunto de novos objetos em um outro já existente, indexado por sua árvore-R. Isto pode ocorrer, por exemplo, quando uma nova área geográfica é mapeada e incluída na base de dados já existente. O algoritmo *Generalized R-Tree Bulk-Insertion* (CHOUBEY, 1999)(CHEN, 2002) primeiro agrupa os objetos a serem inseridos e cria pequenas árvores-R, uma para cada grupo. A inclusão do objeto em um grupo é por proximidade espacial. Não há regras para definir o número de objetos em um grupo. Após, cada subárvore é inserida na árvore-R já existente, em um nodo intermediário adequado. Lee et al (2003) agrupam os objetos a serem inseridos em uma única subárvore-R, que é incluída no nível adequado. Após a

inserção, é útil proceder um reorganização local, para reduzir a intersecção com outras sub-árvores próximas.

## 2.6 Junção espacial baseada em árvores-R

Em geral, os algoritmos de junção espacial baseados em árvores-R podem ser aplicados a quaisquer variação de árvore-R. Quando os dois conjuntos de dados estão indexados, o algoritmo mais conhecido é o *Synchronized Tree Transversal* (STT) (BRINKHOFF, 1993), mostrado na figura 2.14. O STT percorre as duas árvores de índice a partir do nodo raiz de cada uma delas, de modo sincronizado. Para cada nodo filho na árvore do conjunto *A*, são identificados os nodos filhos da árvore do conjunto *B* que atendem ao predicado de junção, e uma lista de possíveis pares é criada.

O algoritmo segue recursivamente, tomando os nodos de cada par como raízes de suas subárvores, até alcançar o nível das folhas. Neste nível, os envelopes de pares de objetos são verificados pelo predicado de junção espacial. Se atenderem ao predicado, o par é incluído no conjunto resposta.

```
Procedure Juncao_STT( N1: Nodo, N2: Nodo)
/*f1 e f2 são entradas em um nodo.
  Cada entrada possui dois campos:
        env, que representa as coordenadas do envelope da subárvore,
        pont, que contém o ponteiro para a subárvore */
Begin
  For each(f1 in N1) do
     For each (f2 in N2) do
           If f1.env∩f2.env≠∅ then
                 If  (está no nível de folhas) then
                       Insere na lista de pares(f1,f2)
                 Else
                       STT(f1.pont, f2.pont)
                 End-if
           End-if
     End-for
  End-for
End
```

Figura 2.14 : Algoritmo STT para junção baseada em árvores-R.

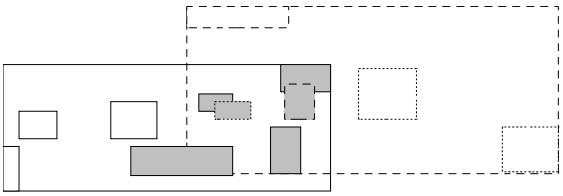


Figura 2.15 : Intersecção entre envelopes de dois nodos de uma árvore-R*.

O algoritmo de Brinkhoff, Kriegel e Seeger foi proposto em 1993 e baseia-se na aplicação do algoritmo STT em árvores-R* (BRINKHOFF,1993). Uma otimização proposta é verificar a intersecção somente entre objetos que pertençam a área de intersecção dos dois envelopes, como mostra a figura 2.15, onde apenas os objetos hachurados serão considerados. Outra otimização é ordenar os objetos por um dos eixos e utilizar o algoritmo de *plane-sweep* para verificar a intersecção entre objetos em dois nodos folhas. A mesma idéia pode ser utilizada para os envelopes que representam sub-árvores no nível intermediário e resultou em redução do número de operações de E/S. Esta redução ocorre porque, ao impor uma certa ordem, aumenta a probabilidade de um

nodo, que deve ser percorrido várias vezes, ser encontrado no *buffer* de memória em acessos futuros.

A estimativa do número de acessos a disco considera, inicialmente, que a altura das duas árvores é idêntica, $h^A = h^B$ e pode se expressa apenas por $h$. Considera também que o espaço é unitário, ou seja, todos objetos estão inseridos em um espaço $d$-dimensional $[0,1]^d$. Inicialmente, o *buffer* será desconsiderado, ou seja, $M=0$. Seja $n_i^A$ o número de nodos no nível $i$ da árvore-R relativa ao conjunto $A$, e $1 \leqslant i \leqslant h$. O tamanho médio, no eixo $x$, dos envelopes de nodos ou objetos no nível $i$ é representado por $s_{x,i}^A$. De acordo com Huang (1997), o número de vezes em que pares de nodos são comparados pode ser estimado por

$$Comp = \sum_{i=1}^{h-1} \sum_{1}^{n_i^A} \sum_{1}^{n_i^B} (min(s_{x,i}^A + s_{x,i}^B), 1) \times (min(s_{y,i}^A + s_{y,i}^B), 1) \quad (16)$$

A partir do número de comparações, pode-se estimar o número de operações de E/S necessárias como sendo:

$$Z = 2 + 2Comp \quad (17)$$

Na primeira parcela, o valor 2 representa a leitura dos nodos raiz. Na segunda parcela, a constante 2 deve-se a necessidade de ler os dois nodos que serão comparados.

Incluindo a existência de um *buffer*, $\rho$ representa a possibilidade de um certo nodo ser encontrado no *buffer*, evitando a realização de uma operação de E/S, reduzindo o número esperado de operações de E/S para

$$disco_{STT} = n_R^A + n_R^B + (Z - n_R^A - n_R^B)\rho \quad (18)$$

De acordo com (18), o desempenho do algoritmo depende das características espaciais dos objetos. Envelopes maiores tendem a degradar o desempenho. Desta maneira, a construção otimizada de árvores-R é positiva, pois os nodos intermediários possuem, em geral, tamanho menor que os nodos resultantes do algoritmo de inserção objeto-a-objeto. *Buffers* de memória maiores tendem a aumentar a probabilidade do nodo já se encontrar em memória, reduzindo o número de operações de E/S.

A ordem de desempenho do algoritmo, em termos de operações de CPU, pode ser obtida a partir da expressão (16), que indica o número de comparações que são realizadas entre diferentes pares de nodos. A cada comparação, um máximo de *2.Fanout* entradas são processadas pelo algoritmo de *plane-sweep*, sendo este o $n$ da sua expressão de desempenho $O(c + n \log n)$, restando estimar $c$.

Para tanto, é importante observar que os nodos de uma árvore-R representam uma parte do universo e apenas pares de objetos alocados a nodos que correspondam a mesma região do espaço serão verificados. A figura 2.16 ilustra este ponto. Pares de objetos alocados aos nodos $A_1$ e $B_1$ serão verificados, enquanto pares de objetos dos nodos $A_1$ e $B_2$ não serão verificados, porque $A_1$ e $B_1$ possuem intersecção, enquanto $A_1$ e $B_2$ não possuem.

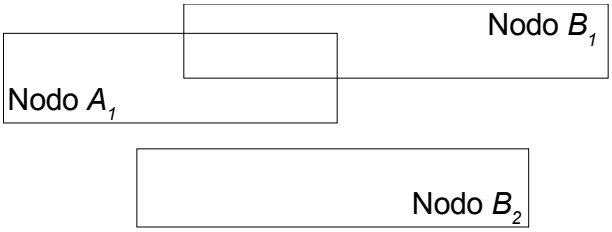



Figura 2.16 : Exemplo de nodos de duas árvores-R, A e B, ocorrendo intersecção entre nodos $A_1$ e $B_1$, mas não entre $A_1$ e $B_2$.

Sem a subdivisão do espaço resultante da árvore-R, como os nodos *A1* e *B2* estão verticalmente alinhados, pares de seus objetos seriam analisados. Com a divisão, apenas objetos em pares de nodos que se interseccionam são verificados, reduzindo o número de pares verificados. No nível das folhas, nível 1, temos

$$c_{STT,1} = \frac{c}{s_1^R}, \tag{19}$$

que é a expressão (5) adaptada para a situação específica.

O número máximo de pares verificados em um nível não-folha depende do tamanho médio dos nodos do nível inferior, e pode ser obtido a partir da expressão (1), resultando

$$c_{STT,i} = min(s_{x,i+1}^A + s_{x,i+1}^B, 1) \times n_{i+1}^A \times n_{i+1}^B$$

Para obter a estimativa do total de pares testados, basta fazer o somatório dos níveis:

$$c_{STT} = \sum_{i=1}^{h} c_{STT,i}$$

A ordem de desempenho do algoritmo é representada pela expressão (20), não sendo dependente do tamanho da memória disponível para *buffer*.

$$cpu_{STT} = O(c_{STT} + Comp \times 2\, Fanout \log 2\, Fanout) \tag{20}$$

Um segundo modelo analítico é proposto por Theodoridis (1996, 2000), que define o número de operações de E/S necessários como sendo

$\sum_{i=1}^{h-1} NumAcc(R_1, R_2, h_i) + NumAcc(R_2, R_1, h_i)$, onde $NumAcc(R_1, R_2, h_1)$ representa o número de acessos para combinar nodos da árvore-R *A* com nodos da árvore-R *B*, no nível *i*. Este pode ser calculado pela fórmula

$$NumAcc(R_1, R_2, h_1) = n_i^A . n_i^B . (s_{x,i}^A + s_{x,i}^B) \times (s_{y,i}^A + s_{y,i}^B)$$

Como $NumAcc(R_1, R_2, h_i) = NumAcc(R_2, R_1, h_i)$, se as árvores tiverem a mesma altura, podemos combinar as duas fórmulas e obter uma expressão muito semelhante a de Huang. A diferença ocorre por que Theodoridis considera que os nodos raiz já estão em memória, não necessitando serem lidos. Quanto a função de mínimo, ela estabelece um limite máximo para a soma dos tamanhos de dois nodos, o valor 1, que representa todo espaço na dimensão considerada. Em uma situação normal, a soma dificilmente ultrapassará o valor, podendo eliminar a função de mínimo, sem perda de correção. Aliás, Theodoridis, em outro artigo (1998), não utiliza a função de valor mínimo.

Os dois modelos possuem uma previsão para o caso da altura das árvores ser diferente. Nesta situação, o algoritmo fixa os nodos folhas da árvore menor e busca

nodos inferiores da árvore maior, até atingir o nível das folhas. O número de pares de nodos processados, considerando $h^A < h^B$ , deve ser alterado para

$$CompH = Comp + \sum_{i=h^A}^{h^B-1} \sum_{1}^{n^A_{h^A}} \sum_{1}^{n^B_i} (s^A_{x,h^A} + s^B_{x,i}) \times (s^A_{x,h^A} + s^B_{x,i})$$

O número máximo de operações de E/S também é alterado para

$$ZJH = 2 + 2\,CompH$$

e o efeito do *buffer*, como conseqüência,

$$disco_{STT} = n^A_R + n^B_R + (Z - n^A_R - n^B_R)\rho \quad (21)$$

Em termos de operações de CPU, a complexidade do algoritmo é da ordem de

$$cpu_{STT} = O(c_{STT} + CompH \times 2\,Fanout \log 2\,Fanout) \quad (22)$$

### 2.6.1 Algoritmo *Priority Queue Driven Process*

Uma alternativa ao STT é o algoritmo *Priority Queue Driven Process* (PQDP), que também pode ser utilizado se os dois arquivos de dados já possuírem árvores-R indexando-os. O PQPD é uma adaptação do *plane-sweep*, sendo as duas árvores percorridas por uma *sweep-line*, que identifica os nodos filhos a serem processados. Há uma lista de nodos ativos para cada árvore. Se dois nodos se interseccionam, o algoritmo verifica a intersecção entre os nodos filhos. Quando um nodo folha é lido para memória, seus objetos podem ser inseridos na lista de objetos ativos, permitindo o teste do predicado espacial de pares de objetos.

Uma vantagem do PQPD é que cada nodo da árvore é lido uma única vez, reduzindo o número de operações de E/S. Já o número de pares de nodos e objetos comparados é idêntico. Porém, como os autores destacam, é necessário manter vários nodos em memória, além dos objetos ativos, requerendo um espaço de memória maior. O algoritmo STT foi implementado neste trabalho por sua popularidade, inclusive estando incorporado a alguns SGBDG importantes, como Oracle 10g (ORACLE, 2003), PostGis (REFRACTIONS RESEARCH,s.d.) e IBM DB2 Spatial Extender (IBM, s.d.). Ainda, é importante notar que as operações de E/S são a parte menos significativa em junções espaciais.

## 2.7 Algoritmos quando apenas um arquivo esta indexado

Um grupo diferente de algoritmos é utilizado quando apenas um dos conjuntos estiver indexado por uma árvore-R. Nesta seção, considera-se o conjunto *A* como indexado por uma árvore-R, e o conjunto *B* como não-indexado, em todas situações, apenas para facilitar o entendimento. De alguma forma, estes algoritmos baseiam-se em abordagens já descritas, como arquivos ordenados ou subdivisão do espaço, cujas características já foram descritas.

O algoritmo *Scan Index* utiliza, para cada objeto do conjunto *B*, seu envelope como "janela" e realiza uma consulta por janela na árvore de índice do conjunto *A*, para localizar objetos que atendam ao predicado de junção. Este algoritmo é dependente da memória disponível para manter nodos da árvore-R em memória, que são repetidamente consultados para verificar a intersecção com diferentes objetos de *B*. Quanto maior a

memória disponível, menor o número de operações de E/S. Porém, o processamento necessário não é afetado pela memória disponível.

Uma alternativa para melhorar o desempenho é classificar os objetos do conjunto *B* por um critério espacial, como valores de uma curva de Hilbert ou apenas por uma das coordenadas do envelope. Espera-se que, deste modo, aumente a probabilidade de dois objetos em seqüência utilizarem os mesmos nodos da árvore de índice, reduzindo o número total de operações em disco, porém, acrescenta o custo da ordenação. A inclusão de uma etapa de ordenação do conjunto *B*, obviamente, aumenta o número de operações de E/S e a complexidade de processamento do algoritmo.

Outra alternativa é o algoritmo *Build&Match (B&M),* que consiste em construir a árvore-R correspondente ao conjunto *B* como uma primeira etapa do processamento, e, depois, aplicar o algoritmo STT. Seu desempenho é a soma da construção da árvore-R do conjunto B e a junção espacial.

Um ganho do *B&M* é que os últimos nodos construídos pelo STR, justamente o nodo raiz e níveis imediatamente inferiores, possuem alta probabilidade de se encontrarem no *buffer* quando do início do algoritmo de junção espacial. Além disso, a nova árvore-R pode ser mantida e aproveitada em outras operações de junção espacial, o que justificaria o consumo de recursos na sua construção.

O algoritmo *Seeded-Join*, proposto por Lo e Ravishankar (1994, 1995), realiza a indexação do conjunto *B* construindo uma estrutura específica, a *Seeded-Tree,* que é uma adaptação de árvore-R. Após, aplica-se o algoritmo STT.

A *Seeded-Tree* possui os nodos superiores copiados da árvore-R existente. Os objetos do conjunto são inseridos como filhos dos nodos copiados. Inicialmente, formam-se arquivos sequenciais de objetos, após, são organizados em sub-árvores-R, uma para cada nodo de nível superior. A figura 2.17 ilustra este processo.

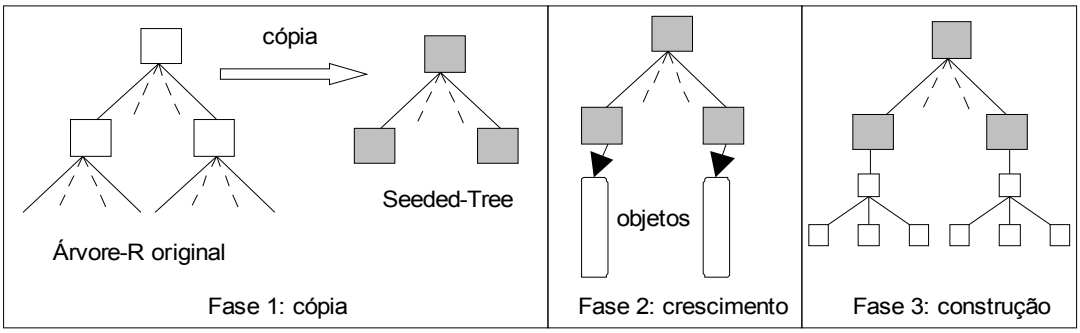



Figura 2.17 : Fases de construção de uma *Seeded-Tree*.

Obtida a nova árvore, é possível aplicar o algoritmo STT aos dois conjuntos. A principal vantagem deste algoritmo baseia-se na igualdade entre os níveis superiores das duas árvores. Deste modo, haverá menos intersecções entre sub-árvores diferentes, reduzindo o número de operações de E/S e CPU. Porém, o algoritmo de construção é mais complexo que o do STR e a árvore resultante não é uma árvore-R, dificultando sua utilização em outras operações espacias.

Já o algoritmo *Sort Sweep-Based Spatial Join (SSBSJ)*, de Gurret e Rigaux (2000), baseia-se em uma abordagem de ordenação. Primeiro, os envelopes dos nodos folhas da árvore-R existente são ordenados por uma coordenada, por exemplo, $y_{min}$. Os nodos folha, a maior parte da árvore-R, não precisam ser lidos, pois seus envelopes encontram-

se no nível imediatamente superior. Além de reduzir o número de operações de E/S, há grande possibilidade de manter todos envelopes em memória. Após, os objetos do conjunto *B* são ordenados pela mesma coordenada.

O terceiro passo consiste em subdividir o eixo em que os objetos não estão ordenados, neste caso, *x*, em faixas. Os limites das faixas são definidos pelos objetos do conjunto *A,* permitindo que cada nodo folha, com seus objetos, seja alocado a uma única faixa, sem replicações. Os objetos do conjunto *B* são alocados posteriormente. Se interseccionarem mais de uma faixa, suas cópias são inseridas em cada faixa. A figura 2.18 ilustra a definição de faixas (*strips*) em um espaço com alguns objetos do conjunto *A*, que são a referência para delimitar as faixas.

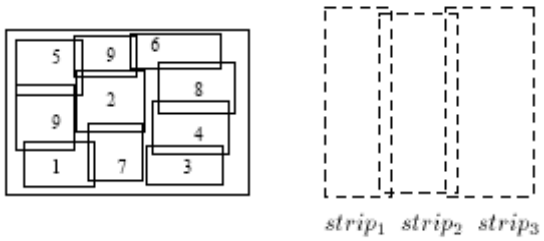



Figura 2.18 : Exemplo de faixas no algoritmo *Sort Sweep-Based Spatial Join*.

Finalmente o algoritmo de *plane-sweep* é realizado para verificar pares de objetos que atendam ao predicado de junção espacial. Para cada faixa há duas listas de objetos ativos, o que permite processar os objetos de cada faixa de maneira independente. Quando é encontrado um envelope de um nodo folha do conjunto *A*, este é substituído pelos objetos contidos no nodo, que serão incluídos na lista de ativos da sua faixa e verificados com objetos do conjunto *B,* também da sua respectiva faixa. Desta forma, a árvore-R existente é lida uma única vez.

O algoritmo é complementado com uma previsão para o caso de não ser possível processar todas as faixas ao mesmo tempo, por limitações de memória. Neste caso, os objetos ativos de uma faixa são colocados em um arquivo auxiliar e o processamento prossegue para as demais faixas. Ao final, será necessário retomar a faixa, recolocando os objetos ativos em memória e relendo a árvore-R e os objetos do conjunto *B*, a partir do ponto de interrupção.

Comparado aos algoritmos baseados em ordenação, para o caso de já existir uma árvore-R em um dos conjuntos, *SSBSJ* é mais eficiente, pois aproveita a existência prévia da árvore-R para não ordenar o conjunto *A*. E, quanto aos algoritmos de junção em árvore, é mais eficiente por não construir uma árvore-R para o conjunto *B*. Porém, não haverá uma nova árvore-B para ser aproveitada em outras operações. Caso o conjunto *B* seja resultado intermediário de uma operação anterior na consulta, talvez realmente não haja interesse em construir a árvore, pois o mesmo conjunto não ocorrerá em outras consultas.

O algoritmo *Priority Queue-Driven Process* (Arge, 2000) pode ser adaptado para o caso de um dos conjuntos possuir uma árvore-R. Para tanto, os objetos do conjunto *B*, não-indexado, são ordenados por um eixo. O caminhamento na árvore é combinado com a sequência de objetos do conjunto B. Quando um nodo intersecciona um objeto de *B*, seus nodos filhos são colocados em memória. Se for um nodo folha, os objetos são carregados em memória e verificados com o objeto do conjunto *B*.

Este algoritmo é uma variação do SBSJJ, alterando a forma de aproveitar a árvore-R existente e eliminando a divisão em faixas. Também não há construção de uma árvore-R para o conjunto *B*.

## 2.8 Sumário

Neste capítulo, os algoritmos apresentados na literatura para resolver a fase de filtragem da junção espacial entre dois conjuntos de objetos foram descritos, utilizando a divisão em classes apresentada no início do capítulo, na figura 2.6.

No primeiro grupo, de arquivos seqüenciais, não indexados, é possível resolver a junção espacial pelo algoritmo de laços aninhados ou utilizando a idéia de subdivisão do espaço. O algoritmo de laços aninhados é simples, com problemas de desempenho já conhecidos (KORTH, 2005). Os algoritmos de subdivisão do espaço criam arquivos temporários de partição, que podem ser descartados ao final do processamento, variando entre eles pelo método de divisão do espaço escolhido.

No segundo grupo, de arquivos ordenados, os dois algoritmos aproveitam-se de que os objetos já estão ordenados por uma das coordenadas e realizam o teste do predicado espacial diretamente. Se um, ou ambos conjuntos, não estiver ordenado, será necessário ordená-los previamente, o que pode se realizar em memória ou disco.

No terceiro grupo, os dois algoritmos podem ser adaptados para as diferentes alternativas de árvore-R descritos na literatura. No entanto, o PQDP2 possui um requisito de espaço em memória maior.

No quarto grupo, inicialmente um dos conjuntos já está indexado por uma árvore-R, há diferentes abordagens: apenas utilizar a árvore e realizar repetidas consultas por janela, ordenar o arquivo não indexado, construir uma árvore de índice ou subdividir o espaço com base nos níveis superiores da árvore.

A construção da árvore de índice pode ser interessante se ela for reaproveitada em outras operações espaciais, justificando o tempo para sua criação. Se não houver esta possibilidade, os demais algoritmos podem se adequar perfeitamente.

No próximo capítulo, os resultados obtidos em um ambiente de testes são apresentados, considerando alguns destes algoritmos, o número de operações de E/S e o tempo de resposta. Esta análise de desempenho foi obtida pela realização de testes com conjuntos de objetos artificiais e reais.

# 3 AMBIENTE DE TESTES E RESULTADOS OBTIDOS

Neste terceiro capítulo é apresentado o ambiente de testes para algoritmos da etapa de filtragem de junções espaciais, bem como os principais resultados obtidos a partir da execução destes algoritmos. A construção deste ambiente tem por objetivo verificar a adequação das fórmulas de desempenho obtidas anteriormente e sua utilidade para prever o desempenho dos algoritmos frente a conjuntos reais de dados, pois as expressões são obtidas a partir de algumas simplificações, tanto dos algoritmos quanto dos dados. Por exemplo, quase todas expressões consideram os objetos distribuídos uniformente no universo, o que não corresponde à realidade.

Em geral, ao apresentar um algoritmo, seus autores realizam comparações através da execução de testes em ambientes controlados. A tabela 3.1 procura resumir os principais tipos de testes realizados, indicando com quais algoritmos cada autor comparou o seu, que dados utilizou, quais variáveis de configuração do ambiente modificou e quais variáveis mediu. A tabela está em ordem cronológica, considerando a data de publicação do primeiro artigo que apresenta o algoritmo, tornando evidente quais algoritmos já eram conhecidos anteriormente à cada publicação. No caso de dados reais, os conjuntos utilizados são de polígonos, representados pelos seus envelopes.

Quanto à tabela 3.1, pode-se observar que a maioria dos autores realiza a comparação do seu algoritmo com outros dois algoritmos e, em geral, pertencentes ao mesmo grupo de algoritmos, conforme a classificação da figura 2.6.

O número de conjuntos de dados também é relativamente restrito. A obtenção de dados reais não é simples, porém, cada vez mais conjuntos de dados têm sido disponibilizados na Internet, o que permite realizar testes mais abrangentes. Os parâmetros de configuração do ambiente são, em quase todos casos, o tamanho de buffer em memória. Para alterar a cardinalidade dos conjuntos de teste, é necessário gerar vários conjuntos de dados artificiais, alterando o número de objetos em cada conjunto.

A variável medida é, em geral, o tempo de resposta, que pode ser dividido em tempo de E/S e tempo de processamento.

Este trabalho analisa separadamente algoritmos de diferentes classes, tornando a comparação mais ampla. Da mesma forma, o número de conjuntos de dados reais também garante a verificação de diversas situações reais diferentes, pois as características de cada conjunto são variadas. Para verificar a influência da cardinalidade dos conjuntos e a distribuição espacial dos objetos dentro do conjunto, diferentes conjuntos de dados artificiais são utilizados. Os algoritmos verificam dois

predicados espaciais diferentes: a intersecção e a distância entre objetos. Para cada algoritmo, são medidos o número de acessos a disco, tempo de resposta e o número de pares de objetos comparados.

Tabela 3.1 : Resumo das análises de desempenho realizadas para algoritmos de junção espacial

| | ***Algoritmos com os quais foi comparado*** | ***Conjuntos de Dados utilizados*** | ***Variáveis de configuração*** | ***Variáveis medidas*** |
|---|---|---|---|---|
| STT (1993) | Alternativas do próprio algoritmo | Dois conjuntos de dados reais. | Tamanho de bloco<br>Tamanho de *buffer* | Número de acessos a disco<br>Tempo de resposta<br>Número de pares comparados |
| PBSM (1996) | Scan-Index e STT | Dados reais de duas regiões | Tamanho do *buffer* | Tempo total de execução, dividido entre CPU e E/S |
| Seed-T (1996) | Scan-Index e STT | Dados artificiais, apenas um caso com dados reais | Cardinalidade<br>Distribuição espacial<br>Tamanho do *buffer* | Número de acessos a disco |
| SSBSJ (1998) | PBSM | Três conjuntos de dados reais e dados artificiais | Cardinalidade | Tempo de execução |
| SHJ (1998) | Seeded-Tree e STT | Somente dados artificiais | Cardinalidade<br>Distribuição espacial | Número de acessos a disco |
| Size-J (1998) | PBSM, SHJ | Quatro conjuntos de dados artificiais, três conjuntos de dados reais | Tamanho de *buffer* | Tempo de execução<br>Número de acessos a disco estimado por expressões de custo |
| S3J (2000) | Scan-Index, Seeded-Tree e STT | Três conjuntos de dados artificiais | Cardinalidade<br>Tamanho do *buffer* | Tempo de execução, dividido entre CPU e E/S |
| PQDP (2000) | PBSM, SSBSJ e STT | Dados reais | Computadores diferentes[1] | Tempo de execução, dividido entre CPU e E/S<br>Número de acessos a disco |
| ISJ (2003) | PBSM, SSSJ, SSBSJ | Quatro conjuntos de dados reais, dados artificiais | Tamanho de *buffer*<br>Cardinalidade | Tempo de execução |
| SISJ (2003) | Scan-Index, Seeded-Tree, Sort&Match e Build&Match | Seis conjuntos de dados reais, quatro conjuntos de dados artificiais | Alternativas do algoritmo<br>Tamanho de *buffer*<br>Cardinalidade | Tempo de execução, dividido entre CPU e E/S |

1: Alterando o computador, vários parâmetros são modificados: velocidade de processamento, memória disponível, tempo de acesso a disco e taxa de transferência de dados entre os discos e a memória.

## 3.1 Descrição do Ambiente de Testes

A figura 3.1 mostra a arquitetura do ambiente de testes. Ele foi totalmente implementado em linguagem C, no sistema operacional UNIX. Sua interface permite escolher o algoritmo a ser utilizado, definindo os valores dos parâmetros de configuração. Até o momento, a interface é realizada através da linha de comando, que permite escrever *scripts* de execução, mas, no futuro, deverá ser aprimorada.

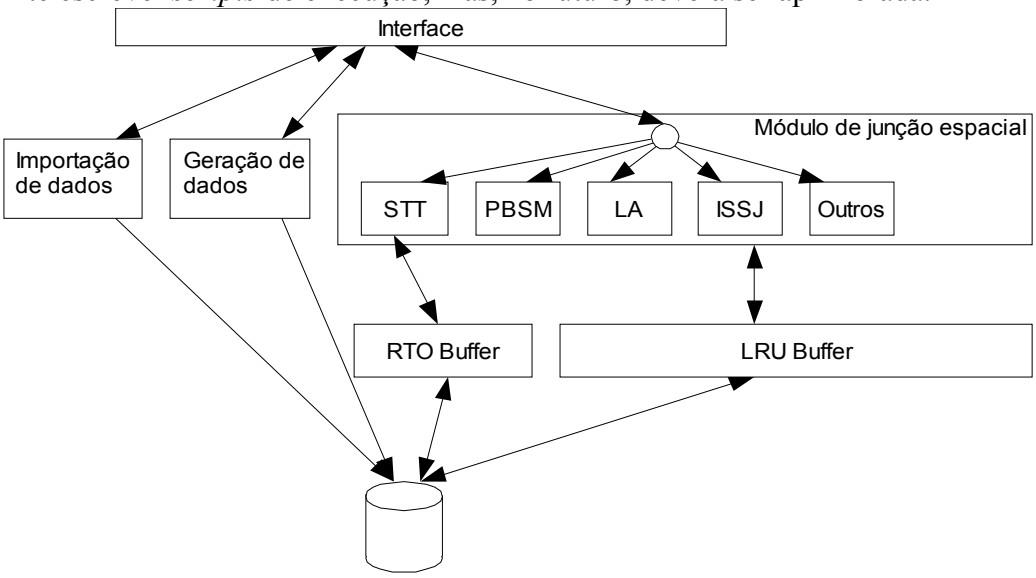



Figura 3.1 : Arquitetura do ambiente de testes de algoritmos de junção espacial.

O módulo de importação de dados permite converter arquivos de dados reais, obtidos na Internet, para o formato utilizado no ambiente. Eventualmente, ele é complementado por uma ferramenta de conversão, chamada FME (SAFE SOFTWARE, 2006) que aceita mais de 40 diferentes formatos de dados de entrada e permite gerar um arquivo texto com as coordenadas dos objetos. Depois, este arquivo texto pode ser processado pelo conversor, gerando um arquivo específico do ambiente. Desta maneira, se atinge um alto grau de flexibilidade, permitindo manipular um maior número de arquivos de dados reais.

Os dados reais utilizados para testes estão disponíveis em três fontes diferentes. A tabela 3.2 lista o nome dos arquivos e as principais características. O tamanho médio baseia-se em um espaço normalizado, com coordenadas entre zero e um.

O módulo de geração de dados sintéticos permite criar arquivos de dados com características específicas e controladas pelo usuário, que pode definir:

- cardinalidade do conjunto;
- tamanho mínimo, médio e máximo dos objetos;
- forma de distribuição dos objetos, escolhendo entre distribuição uniforme ou distribuição concentrada em pontos determinados pelo usuário, ou, ainda, baseada na distribuição de conjuntos de dados reais.

O módulo de geração inclui a construção de árvores-R otimizadas, utilizando o algoritmo STR (2.5.2).

Tabela 3.2: Conjuntos de dados reais utilizados nos testes.

| Nome | Descrição | Cardinalidade | Número de blocos | Tamanho em X | Tamanho em Y |
|---|---|---|---|---|---|
| *Fonte: Rtree portal (www.rtreeportal.org)* | | | | | |
| *ca_streets* | Estradas da Califórnia | 2.249.727 | 13.157 | 0,016 | 0,013 |
| *ca_streams* | Cursos da água da Califórnia | 98.451 | 576 | 0,184 | 0,157 |
| *ge_roads* | Estradas da Alemanha | 30.674 | 180 | 0,146 | 0,108 |
| *ge_utility* | Redes de utilidade pública na Alemanha | 17.791 | 103 | 0,236 | 0,169 |
| *ge_rrlines* | Ferrovias da Alemanha | 36.334 | 213 | 0,117 | 0,084 |
| *ge_hypsogr* | Dados hipsográficos[4] da Alemanha | 76.999 | 451 | 0,067 | 0,048 |
| *gr_roads* | Estradas da Grécia | 23.278 | 137 | 0,188 | 0,187 |
| *gr_rivers* | Cursos da água da Grécia | 24.650 | 145 | 0,187 | 0,200 |
| *la_streets* | Ruas da cidade de Los Angeles | 131.461 | 769 | 0,035 | 0,029 |
| *la_rr* | Cursos da água e ferrovias da cidade de Los Angeles | 128.971 | 755 | 0,086 | 0,071 |
| *Fonte:Bureau of Transportation Statistics (USA) – www.bts.gov* | | | | | |
| *usa_counties* | Distritos | 3.232 | 19 | 0,0049 | 0,01 |
| *usa_rr* | Ferrovias | 166.688 | 981 | 0,00049 | 0,0041 |
| *usa_hydro* | Cursos de água | 517.538 | 2959 | 0,00049 | 0,0009 |
| *Fonte: Secretaria de Ciências e Tecnologia do Estado de Goiás (www.simego.sectec.go.gov.br)* | | | | | |
| *go_hydro* | Cursos de água | 66.327 | 390 | *0,01* | *0,0017* |
| *go_cities* | Áreas urbanas | 484 | 2 | *0,012* | *0,0024* |
| *go_roads* | Estradas | 22.712 | 133 | *0,0025* | *0,0027* |
| *go_geomorfo* | Geomorfologia | 2.183 | 12 | *0,016* | *0,008* |
| *go_hypso* | Dados hipsográficos | 3.142 | 18 | *0,011* | *0,0005* |
| *go_soils* | Tipos de solos | 3.006 | 17 | *0,0103* | *0,0013* |
| *go_soilusage* | Utilização do solo | 6.855 | 40 | *0,010* | *0,007* |

A memória disponível é alocada como área de *buffer* de dados, sendo controlada pelo ambiente. Este *buffer* é uma área contínua de memória, cujo tamanho o usuário determina dinamicamente. Sempre que um bloco é lido do disco, é colocado neste *buffer*. Se o *buffer* estiver completo, o bloco acessado há mais tempo é retirado do *buffer*. Se ele houver sido alterado, é gravado para disco. Deste modo, esta implementada uma política de LRU (*Least Recent Used*) (ELMASRI, 2005, p.574). Para manipular árvores-R, foi utilizada uma política de *buffer* adequada, que privilegia os nodos não-folha (BRINKHOFF, 2002).

O módulo de junção espacial é composto por diferentes algoritmos para realizar esta tarefa. O critério inicial de escolha visou abranger diferentes grupos, permitindo uma variedade maior de situações. Em cada grupo, escolheu-se o algoritmo que as

[4]Curvas de nível do relevo.

expressões de custo indicam ser o mais rápido. Assim, foram implementados os algoritmos de Laços Aninhados (arquivos seqüenciais), PBSM (subdivisão de espaço), STT (arquivos indexados), *Build&Match* (um arquivo indexado) e ISSJ (arquivos ordenados). Os resultados do algoritmo Laços Aninhados, como esperado, apresentam tempo de resposta muito superior aos demais algoritmos, assim, a análise não foi aprofundada. Os valores obtidos estão no anexo B.

Ao realizar uma junção espacial, o usuário deve indicar os conjuntos envolvidos e o tamanho da memória alocada para *buffer,* além do algoritmo a ser utilizado. Durante uma execução, o número de operações de E/S é contado, e o tempo de execução das diferentes etapas de cada algoritmo é registrado.

Todos os testes foram realizados em uma estação de trabalho com processador Pentium 2.4 Ghz, 512 Mb de memória RAM e disco IDE de 80 Gb, sistema operacional Linux, *kernel* 2.4.0. Os programas foram compilados com GNU *compiler* para linguagem C. As medições de tempo utilizam os recursos da linguagem C.

## 3.2 Algoritmo *Partition Based Spatial Join Method* (PBSM)

A previsão de desempenho do algoritmo PBSM é

$$cpu_{PBSM} = O(c + r(n^A + n^B)\log\frac{r(n^A + n^B)}{P}) .$$

Se a memória disponível aumenta, o número de partições reduz, aumentando o tempo de resposta. O fator de replicação é influenciado pelo tamanho das células, definido pela grade regular utilizada e pelo número de repartições, que pode gerar mais réplicas de objetos. Isto é claro no gráfico da figura 3.2, que mostra a curva de desempenho do algoritmo variando a memória disponível. Neste caso, o desempenho é medido normalizando o tempo de resposta tendo por base o menor deles, a junção dos conjuntos de 100K em 50 blocos de memória.

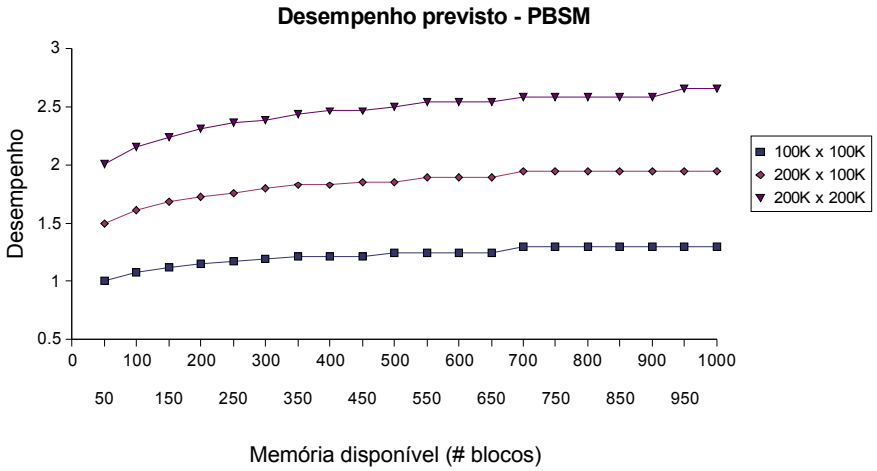



Figura 3.2 : Gráfico do desempenho previsto para o algoritmo PBSM, variando a memória, com base na fórmula de previsão de desempenho.

A versão implementada do algoritmo PBSM inclui uma previsão de aumento do número de objetos igual a 20%, antes de realizar o cálculo do número de partições. Este acréscimo procura incorporar ao cálculo do número de partições uma previsão do fator de replicação, que acrescentará cópias de objeto a serem manipuladas. Também inclui o *Reference Point Method,* para evitar duplicatas no conjunto de resposta. A grade

regular, por *default*, possui 32 x 32 células. Na ocorrência de partições com *overflow,* estas são reparticionadas em quatro novas partições. Esta opção evita que, por uma distribuição muito concentrada dos objetos, uma das novas partições também sofra com *overflow,* como pode ocorrer gerando duas partições, e evita uma replicação desnecessária, se gerássemos mais do que quatro partições.

O gráfico na figura 3.3 mostra o tempo de resposta para três diferentes junções espaciais, variando a memória disponível, onde se pode perceber uma clara tendência ao crescimento do tempo de resposta. As curvas estão interrompidas no ponto em que há memória suficiente para carregar todos objetos de uma única vez, dispensando o PBSM.

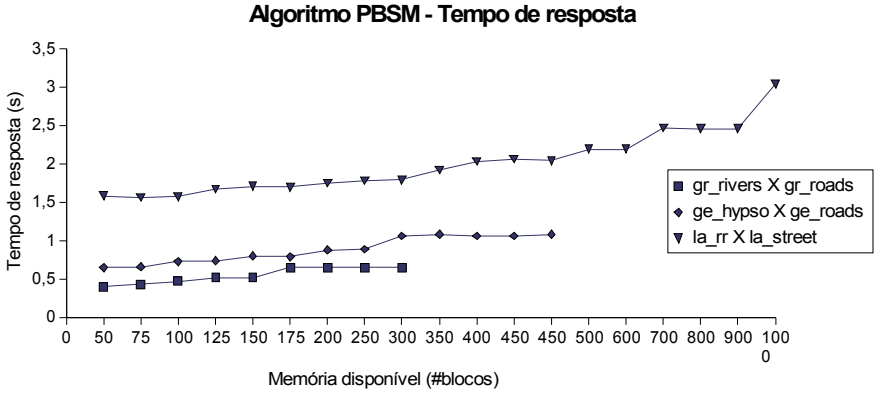



Figura 3.3 : Gráfico do tempo de resposta, algoritmo PBSM, variando a memória disponível.

Comparando os gráficos da figura 3.2, desempenho previsto, e o da figura 3.3, desempenho medido, é fácil constatar que a previsão confirma-se, com o aumento do tempo de resposta à medida que cresce a memória disponível.

O gráfico na figura 3.4 mostra o tempo de resposta para duas junções espaciais diferentes, uma envolvendo conjuntos reais, com distribuição espacial irregular, outra conjuntos sintéticos, com distribuição uniforme. Como são conjuntos com cardinalidade maior, a grade foi alterada para 64x64, totalizando 4096 células. Isto é necessário porque o número inicial de partições, para pouca memória (25 blocos), foi calculado em 1055. Este valor é superior ao número de células de uma grade 32 x 32, não permitindo um mapeamento de várias células para uma partição, que é importante para manter um número de objetos por partição mais constante e evitar a ocorrência de muitos *overflows*.

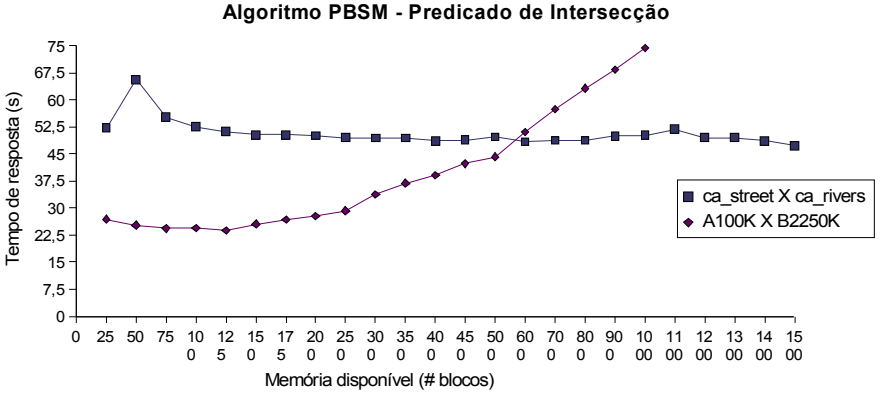



Figura 3.4 : Gráfico do tempo de resposta, algoritmo PBSM, variando a memória disponível, para dados sintéticos e reais.

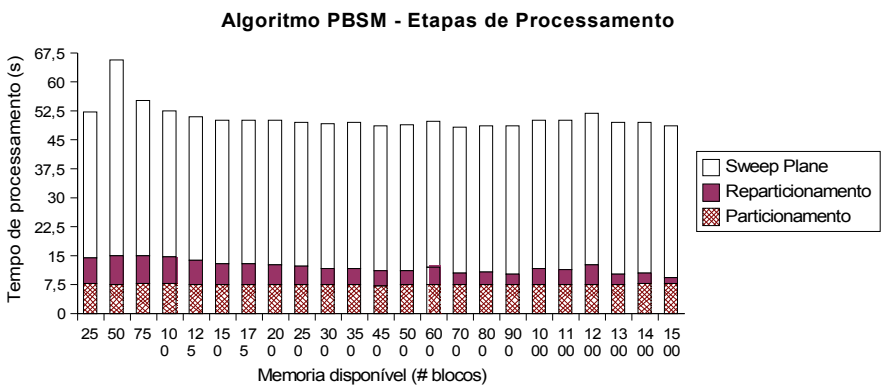



Figura 3.5 : Gráfico da junção espacial *ca_street X ca_stream*, identificando o tempo de processamento de cada etapa do algoritmo.

A junção entre conjuntos sintéticos é coerente com a previsão para o algoritmo. Após uma queda inicial, o tempo de resposta cresce na medida em que a memória disponível também aumenta, indicando a existência de um ponto de inflexão, que é o ponto ótimo para execução do algoritmo. Um forte motivo é que as expressões para prever desempenho do algoritmo supõe uma distribuição uniforme dos objetos, sendo natural que ocorra desvios da previsão quando esta premissa não é atendida pelos conjuntos de dados.

A junção entre conjuntos reais apresenta um comportamento mais complexo. O gráfico na figura 3.5 mostra o tempo de cada uma das etapas do algoritmo. A etapa de particionamento manteve-se estável, em todas configurações. O tempo de processamento da etapa de reparticionamento reduziu, pois com uma capacidade de memória maior, em um número menor de partições ocorreu *overflow*, envolvendo cada vez menos objetos.

A etapa de *plane-sweep* para verificar pares de objetos tem um tempo de execução decrescente. A capacidade máxima da memória impõe um limite superior ao número de objetos de cada partição, mas não há limite inferior. A distribuição irregular dos objetos resulta em partições com número de objetos bem abaixo da capacidade máxima. Estas partições com poucos objetos são processadas muito rapidamente.

Com menor memória disponível, o número de partições é alto. Por conseqüência, cada partição agrega um número pequeno de células, não compensando adequadamente a irregularidade da distribuição de objetos no espaço.

Aumentando a memória disponível, reduz o número de partições, o número de células por partição aumenta e a ocorrência de *overflow* praticamente desaparece. Em um certo ponto, o número de objetos por partição se torna mais equilibrado, aproximando o comportamento do algoritmo ao pressuposto de distribuição uniforme de objetos por partição, base para obter as expressões de desempenho.

A tabela 3.3 mostra o número total de operações de E/S realizada para duas junções espaciais. O número de operações de E/S apresentou uma tendência a reduzir, conforme a disponibilidade de memória aumenta. Isto porque o número de réplicas diminui e, principalmente, a ocorrência de partições com *overflow* reduz, conforme a expressão de custo prevê:

$$disco_{PBSM} = (2 o r^2 + 2r + 1)(b^A + b^B)$$

A queda na porcentagem de partições que sofrem *overflow* não é constante, pois algumas combinações de células/partições podem ser especialmente favorecidas ou prejudicadas por uma determinada distribuição espacial. Por exemplo, para os conjuntos da Califórnia, na situação com 600 e 1000 blocos de memória, o número de objetos manipulados em repartições é superior às situações com menor disponibilidade de memória, enquanto que para conjuntos de Los Angeles a queda é constante.

Tabela 3.3 : Número de operações de E/S para duas junções espaciais.

| ***Memória*** | ***la_rr X la_street*** | | | ***ca_street X ca_stream*** | | |
|---|---|---|---|---|---|---|
| | # de E/S | Partições com *oveflow* | Objetos em partições com *overflow* | # de E/S | Partições com *oveflow* | Objetos em partições com *overflow* |
| 50 | 5464 | 4 | 35201 | 86700 | 45 | 890531 |
| 100 | 5219 | 1 | 17278 | 84424 | 31 | 821208 |
| 150 | 5234 | 0 | 0 | 77730 | 20 | 684865 |
| 200 | 5231 | 0 | 0 | 75461 | 16 | 637999 |
| 250 | 5209 | 0 | 0 | 69636 | 7 | 515234 |
| 300 | 5221 | 0 | 0 | 69179 | 6 | 505488 |
| 350 | 5220 | 0 | 0 | 68023 | 6 | 480979 |
| 400 | 5219 | 0 | 0 | 65072 | 3 | 419467 |
| 500 | 5200 | 0 | 0 | 65766 | 3 | 435042 |
| 600 | 5200 | 0 | 0 | 70475 | 3 | 535482 |
| 700 | 5227 | 0 | 0 | 60863 | 2 | 330796 |
| 800 | 5227 | 0 | 0 | 63610 | 2 | 389859 |
| 900 | 5227 | 0 | 0 | 60999 | 2 | 333975 |
| 1000 | 5202 | 0 | 0 | 68369 | 2 | 490505 |

Tabela 3.4 : Proporção entre tempo de operações de E/S e CPU para o algoritmo PBSM.

| | | ***M= 50*** | ***M=100*** | ***M=150*** | ***M=200*** | ***M=300*** |
|---|---|---|---|---|---|---|
| *gr_rivers x gr_roads* | Tempo de E/S | 0,14 | 0,13 | 0,13 | 0,13 | 0,13 |
| | Tempo de CPU | 0,34 | 0,37 | 0,43 | 0,43 | 0,43 |
| | Tempo total | 0,48 | 0,50 | 0,56 | 0,56 | 0,56 |
| | % de CPU | 69,7 | 73,4 | 76,4 | 76,4 | 76,4 |
| *ge_hypso x ge_roads* | Tempo de E/S | 0,24 | 0,23 | 0,23 | 0,23 | 0,23 |
| | Tempo de CPU | 0,62 | 0,63 | 0,65 | 0,70 | 0,84 |
| | Tempo total | 0,86 | 0,86 | 0,88 | 0,93 | 1,07 |
| | % de CPU | 71,6 | 73,3 | 74,0 | 75,4 | 78,7 |
| *la_rr x la_streets* | Tempo de E/S | 0,83 | 0,72 | 0,67 | 0,60 | 0,55 |
| | Tempo de CPU | 1,64 | 1,61 | 1,64 | 1,67 | 1,90 |
| | Tempo total | 2,47 | 2,33 | 2,31 | 2,27 | 2,45 |
| | % de CPU | 66,4 | 69,1 | 71,1 | 73,5 | 77,4 |

A tabela 3.4 mostra, para algumas situações, o tempo estimado em operações de disco e o tempo de operações em CPU. Como pode-se observar, operações em CPU representam entre 70%, até mais de 80%, do tempo de resposta total.

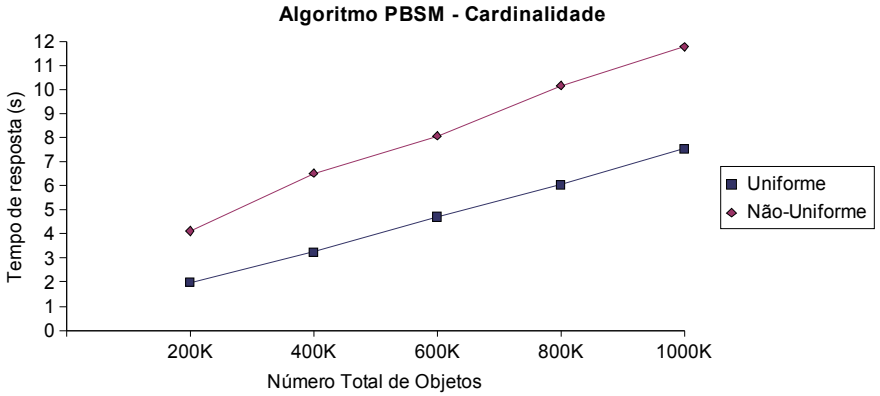



Figura 3.6 : Gráfico do algoritmo PBSM variando a cardinalidade dos conjuntos.

Outro teste realizado procurou verificar o comportamento do algoritmo frente à variação do número de objetos, mantendo o número de pares de objetos testados. Assim, na fórmula básica de desempenho do *plane-sweep*, $O(c+n\log n)$, apenas $n$ é alterado. Para tanto, foram gerados 5 pares de conjuntos de objetos, aumentando a cardinalidade de 100.000 objetos até 500.000 objetos. Portanto, a junção espacial envolvia 200.000 até 1.000.000 de objetos. O tamanho médio dos objetos foi mantido, em números absolutos, sempre idêntico, assim como a memória disponível. Porém, o universo foi sendo aumentado, tornando os objetos mais esparsos, para manter o número de comparações em torno de um mesmo valor. A figura 3.6 mostra o tempo de resposta para duas seqüências de conjuntos: um com distribuição não-uniforme; outra com distribuição uniforme. Como pode-se observar, o tempo de resposta cresce de maneira linear, junto com o número de objetos.

Outro teste verificou o comportamento quando o número de pares de objetos testados modifica-se, ou seja, o outro fator de desempenho do *plane-sweep*. Para tanto, foram gerados conjuntos de objetos com a mesma cardinalidade, sempre 200.000, alterando o tamanho do universo. Assim, o conjunto chamado D10 possui a menor densidade de objetos, pois seus objetos estão espalhados na maior área. O conjunto D20, com a mesma cardinalidade, ocupa metade da área total. Assim, sucessivamente, até o conjunto D50, cujos objetos estão distribuídos em um quinto da área inicial.

A tabela 3.5 contém o número de comparações entre pares de objetos realizadas em cada situação e o número de pares de objetos no conjunto resposta. É interessante notar que o número de pares no conjunto resposta apresenta um crescimento bem mais acentuado que o número de pares testados, sendo, este, o principal componente a influenciar o tempo de resposta. Por exemplo, o número de pares testados para D20 distribuição não-uniforme é, pontualmente, maior que D10 e D30. O tempo de resposta também é maior. Este resultado é importante, pois confirma o acerto de substituir $k$ por $c$ nas expressões de desempenho do *plane-sweep,* e por, conseqüência, dos algoritmos de junção espacial. Ainda, o baixo crescimento no número de pares de objetos testados justifica-se pelo particionamento do espaço, que, nestes casos, é eficiente. A figura 3.7 mostra o tempo de resposta para conjuntos não-uniformes e uniformes. A tendência é de crescimento pouco acentuado. A memória disponível para o algoritmo foi mantida em todos testes.

Tabela 3.5 Número de comparações realizadas pelo algoritmo PBSM, variando a densidade dos objetos no espaço.

| ***Densidade*** | ***Não Uniforme*** | | ***Uniforme*** | |
|---|---|---|---|---|
| | ***k*** | ***c*** | ***k*** | ***c*** |
| D10 | 478.062 | 17.143.251 | 215.612 | 1.998.492 |
| D20 | 688.101 | 21.872.554 | 230.838 | 2.763.637 |
| D30 | 825.675 | 19.437.901 | 247.056 | 3.369.545 |
| D40 | 974.849 | 21.854.051 | 263.032 | 3.865.631 |
| D50 | 1.062.824 | 22.680.960 | 277.678 | 4.314.906 |

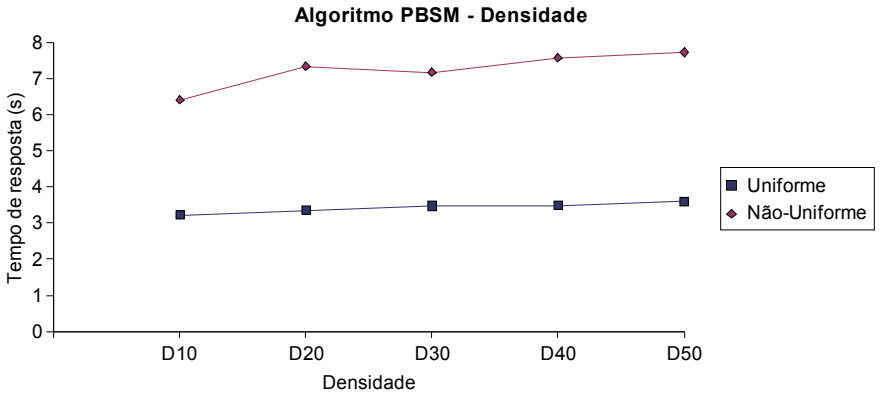



Figura 3.7 : Gráfico do algoritmo PBSM alterando a densidade dos conjuntos de objetos.

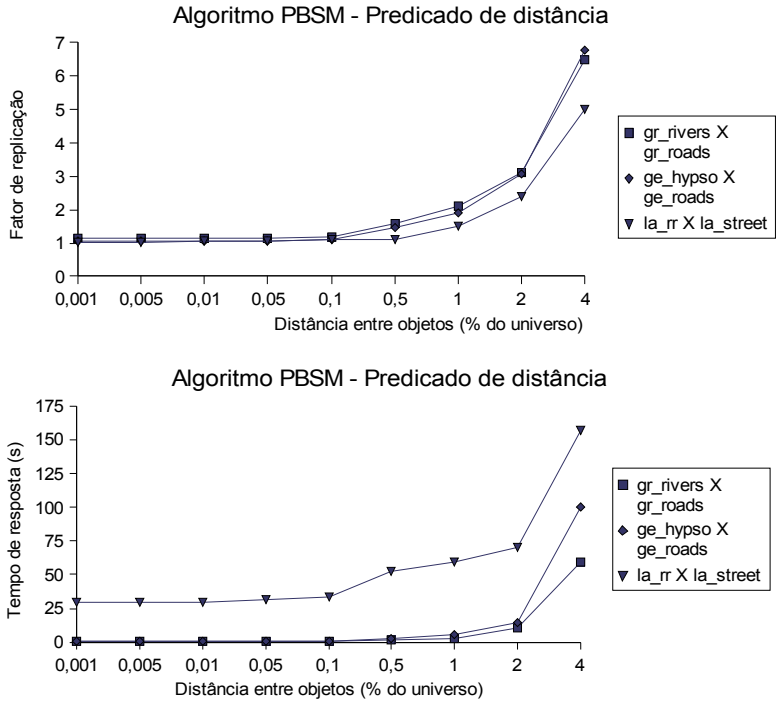



Figura 3.8 : Gráficos do fator de replicação e tempo de resposta do algoritmo PBSM, variando a distância entre objetos no predicado de junção espacial.

Alterando o predicado de junção espacial para a distância entre objetos, o tempo de resposta é proporcional ao fator de replicação, como pode ser visto nos gráficos da figura 3.8. Neste caso, a densidade também aumenta, com o crescimento dos objetos. A cardinalidade dos conjuntos foi mantida. A esquerda, o gráfico mostra o fator de replicação crescendo conforme a aumenta a distância entre objetos, assim como o tempo de resposta.

## 3.3 Algoritmo STT

O algoritmo STT, para junção espacial utilizando duas árvores-R já existentes, foi implementado para prover o suporte a apenas objetos bi-dimensionais. As árvores-R de conjuntos de dados reais, descritos na tabela 3.2, possuem entre 2 e 4 níveis. A construção das árvores-R foi realizada utilizando o método de construção otimizado, chamado STR, que resulta em aproveitamento quase total das entradas.

A figura 3.9 mostra o gráfico de desempenho previsto para três diferentes junções espaciais, obtido a partir da aplicação da expressão de desempenho:

$cpu_{STT} = O(Comp \times 2\, Fanout \log Fanout)$ , sendo

$$Comp = \sum_{i=1}^{h-1} \sum_{1}^{n_i^A} \sum_{1}^{n_i^B} (min(s_{x,i}^A + s_{x,i}^B), 1) \times (min(s_{y,i}^A + s_{y,i}^B), 1) .$$

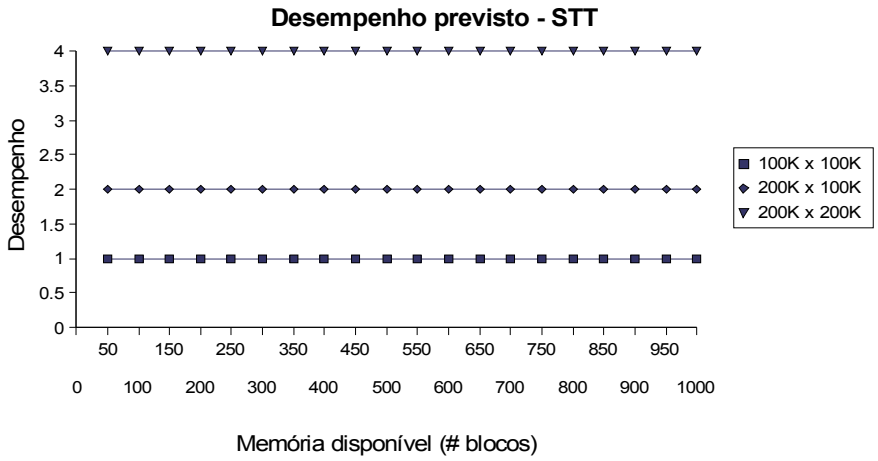



Figura 3.9 : Gráfico do desempenho previsto para o algoritmo STT, variando a memória, com base na fórmula de previsão de desempenho.

Na figura 3.10, o gráfico mostra o tempo de resposta medido de junções espaciais entre três pares diferentes de conjuntos, variando a memória disponível. Confirmando a previsão da expressão de desempenho e os trabalhos de Gatti (2000) e Rocha (2003), o tempo de resposta do algoritmo STT é constante. Esta característica é importante, pois o torna mais previsível, e permite executá-lo sem alocar um espaço de memória significativo.

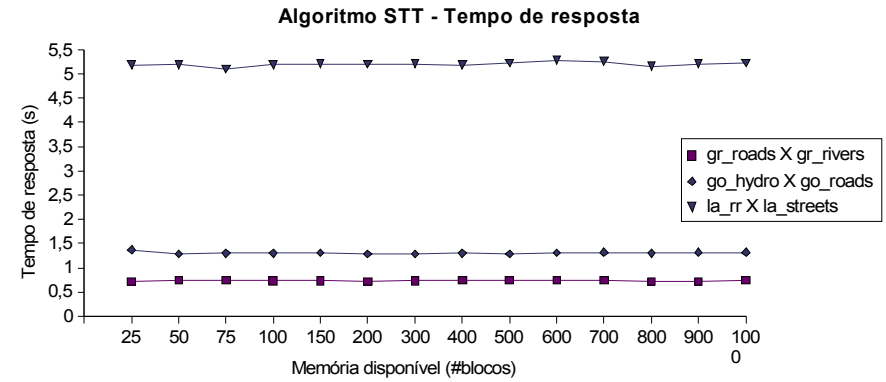



Figura 3.10 : Gráfico do tempo de resposta do algoritmo STT para quatro junções espaciais diferentes, variando a memória disponível.

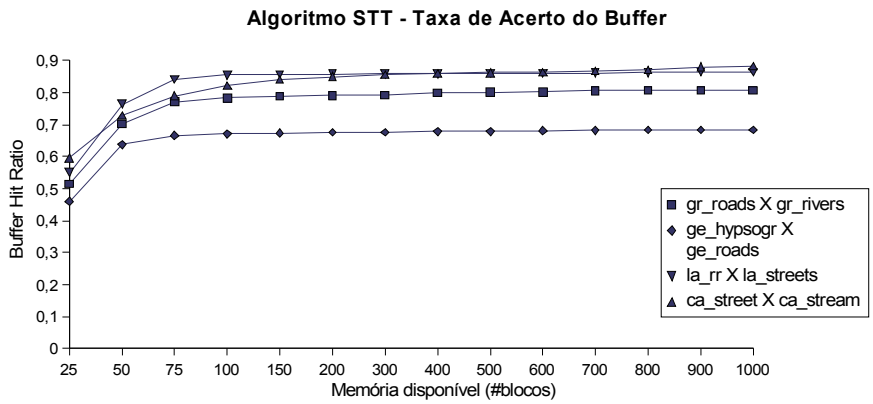



Figura 3.11 : Gráfico da taxa de acerto do *buffer* no algoritmo STT, variando a memória disponível para *buffer*, em quatro junções espaciais.

O número de operações de E/S é dependente da memória disponível, pois a taxa de acerto de blocos no *buffer* cresce a medida que a memória aumenta. Pelo gráfico da figura 3.11 pode-se notar, também, que a partir de um certo ponto, a taxa de acerto fica estável, quando o *buffer* é suficiente para acomodar as duas árvores inteiras, ou, pelo menos, praticamente todos os nodos que representam certa região do espaço onde o processamento está concentrado. Desta maneira, também confirma-se a expressão que prevê o número de operações de E/S:

$disco_{STT} = n_R^A + n_R^B + ((2 + 2Comp) - n_R^A - n_R^B)\rho$ , sendo $\rho$ a probabilidade de uma página não estar presente no *buffer*.

A tabela 3.6 mostra o percentual de tempo entre operações de E/S e CPU. Como é possível observar, a CPU é responsável por até 97% do tempo total de resposta. Isto se deve, principalmente, a alta taxa de acerto do *buffer* de memória, que a partir de 3 ou 4 vezes a altura das árvores envolvidas, assume valores acima de 90%.

Tabela 3.6 : Proporção entre operações de E/S e CPU para o algoritmo STT.

| | | *M= 25* | *M=50* | *M=100* | *M=200* | *M=300* |
|---|---|---|---|---|---|---|
| *gr_rivers x gr_roads* | Tempo de E/S | 0,022 | 0,019 | 0,018 | 0,017 | 0,016 |
| | Tempo de CPU | 0,358 | 0,341 | 0,342 | 0,343 | 0,354 |
| | Tempo total | 0,38 | 0,36 | 0,36 | 0,36 | 0,37 |
| | % de CPU | 94,2 | 94,8 | 95 | 95,4 | 95,6 |
| *ge_hypso x ge_roads* | Tempo de E/S | 0,031 | 0,030 | 0,028 | 0,026 | 0,026 |
| | Tempo de CPU | 0,419 | 0,44 | 0,442 | 0,454 | 0,464 |
| | Tempo total | 0,45 | 0,47 | 0,47 | 0,48 | 0,49 |
| | % de CPU | 93,1 | 93,7 | 94,0 | 94,6 | 94,7 |
| *la_rr x la_streets* | Tempo de E/S | 0,297 | 0,113 | 0,089 | 0,073 | 0,066 |
| | Tempo de CPU | 5,738 | 5,860 | 5,981 | 5,927 | 5,874 |
| | Tempo total | 6,08 | 5,98 | 6,07 | 6,00 | 5,94 |
| | % de CPU | 95,1 | 98,1 | 98,5 | 98,8 | 98,9 |

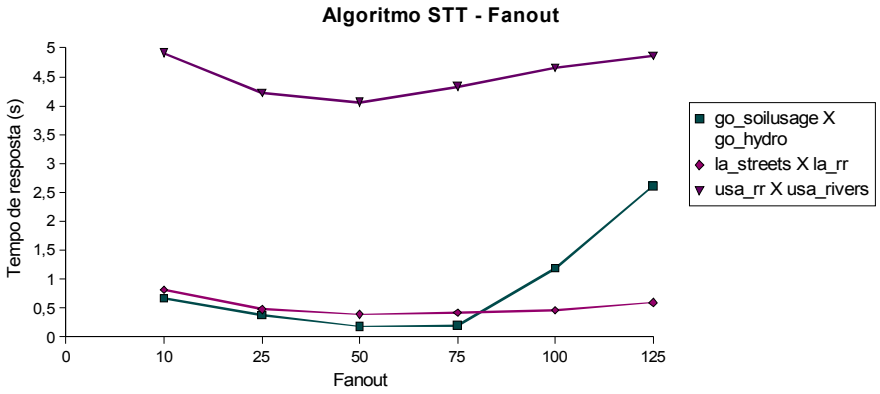



Figura 3.12 : Gráfico do desempenho do algoritmo STT, para diferentes *fanout*.

Um fator que altera o desempenho do algoritmo é o tamanho dos nodos. Nodos com muitas entradas tendem a representar áreas maiores do universo, aumentando a possibilidade de intersecção com nodos da outra árvore, aumentando o número de pares de nodos comparados. Ainda, a cada comparação de um par de nodos, o número de objetos envolvidos é maior. Por outro lado, nodos pequenos resultam em um número de comparações entre pares de nodos muito alto. O tempo de resposta para diferentes número de entradas (*fanout*) em cada nodo é mostrado na figura 3.12, confirmando os dados de (PFOSER e JANSEN,2005), inclusive quanto ao ponto ótimo, em torno de nodos com 73 entradas, ou 2Kb. Este valor, como ponto ótimo para a maioria dos casos, é utilizado em todos resultados deste trabalho, exceto se dito explicitamente.

Para verificar o desempenho do algoritmo quanto à cardinalidade dos conjuntos, mantendo o número de pares comparados, foram utilizados os mesmos conjuntos da figura 3.6. O tempo de resposta obtido para cada operação é mostrado pela figura 3.13. Para conjuntos com distribuição uniforme dos objetos, o comportamento está de acordo com o esperado, aumentando o tempo de resposta conforme aumenta o número de objetos. A divisão do espaço proporcionada pela árvore-R resulta em um crescimento suave. Porém, para o caso de objetos distribuídos não-uniformente, há uma queda no tempo de resposta. Esta redução ocorre também no número de objetos comparados,

conforme a tabela 3.7, e é ocasionada por que o *fanout* não está otimizado para conjuntos menores.

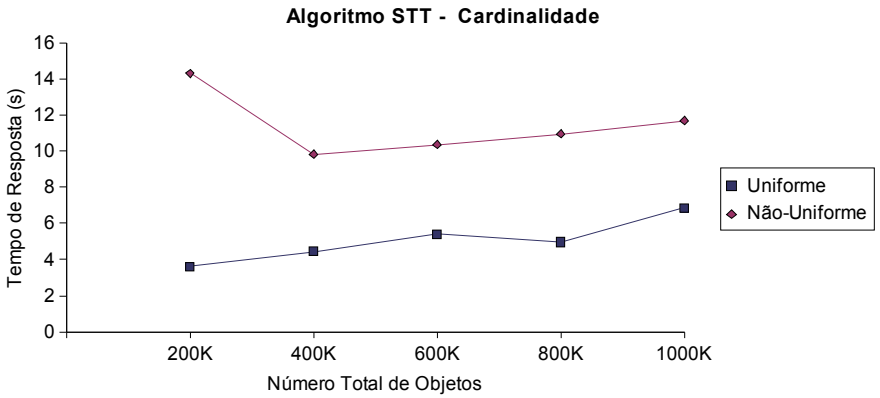



Figura 3.13 : Tempo de resposta do algoritmo STT, variando a cardinalidade dos conjuntos.

Tabela 3.7: Número de pares de objetos comparados pelo algoritmo STT, variando a cardinalidade dos conjuntos de objeto.

| ***Cardinalidade total*** | ***Número de comparações*** |
|---|---|
| 100 K | 25.098.993 |
| 200 K | 25.189.738 |
| 400 K | 22.569.049 |
| 600 K | 22.168.074 |
| 800 K | 22.400.243 |
| 1000 K | 23.446.352 |

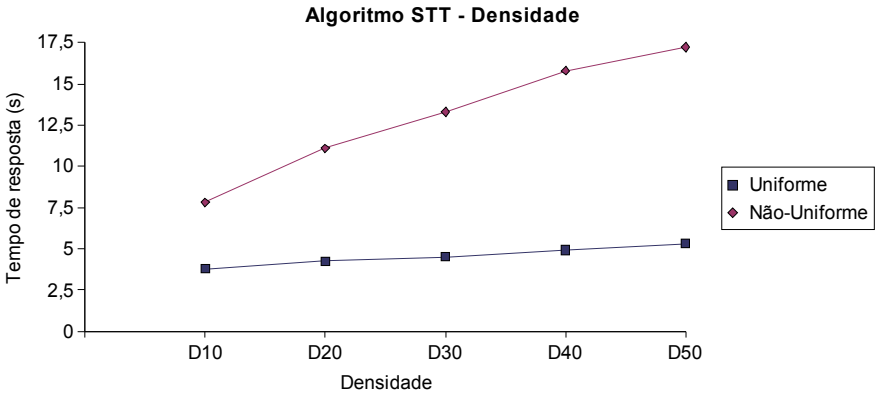



Figura 3.14 : Gráfico do tempo de resposta do algoritmo STT variando a densidade dos conjuntos.

Aumentar a densidade de um conjunto afeta uma árvore-R porque a extensão coberta por um nodo tende a ser menor, porém o número de comparações entre pares de objetos aumenta com a proximidade entre eles. A figura 3.14 mostra o tempo de resposta para uma sequência de conjuntos com objetos distribuídos de maneira não-uniforme e outra de maneira uniforme. No caso de conjuntos não-uniformes, o tempo de resposta é sempre mais alto, pois a divisão do espaço é realizada de maneira menos eficiente pelas

árvores-R. Para conjuntos uniformes, o número de comparações é mantido em cerca de 13% do número de comparações em conjuntos não-uniformes.

A figura 3.15 mostra o desempenho do algoritmo para o predicado de distância, que aumenta o tamanho médio dos objetos, resultando em tempos de resposta cada vez mais altos. A avaliação do predicado de distância é realizada aumentando, em memória, o envelope dos objetos, sem afetar as árvores-R mantidas em disco.

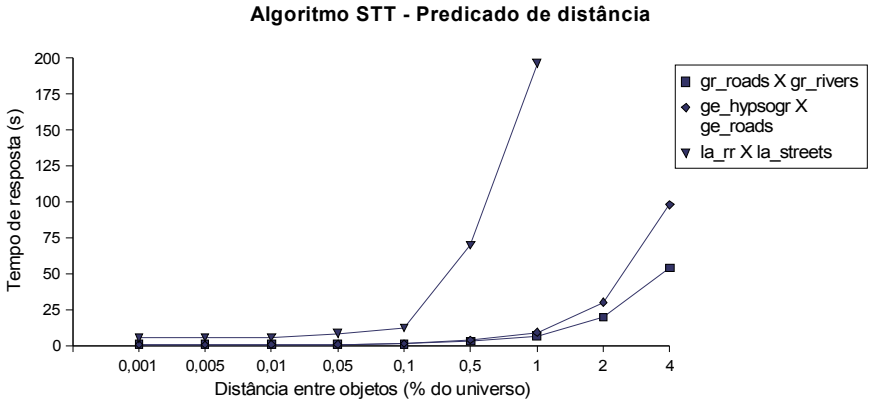



Figura 3.15 : Gráfico do tempo de resposta do algoritmo STT para o predicado de distância, variando a distância mínima entre objetos.

## 3.4 Algoritmo *Build&Match*

O algoritmo *Build&Match* é uma das alternativas para a situação em que apenas um dos conjuntos de objetos possui a árvore-R construída. Na sua implementação, ele constrói uma árvore-R pelo algoritmo STR, e, depois, realiza a junção espacial pelo algoritmo STT. Sendo assim, apenas as questões relativas à fase de construção são analisadas nesta seção.

A figura 3.16 mostra o gráfico com o tempo de construção de vários dos conjuntos de dados reais. Todos os testes foram realizados utilizando um *buffer* de memória com 500 blocos. A etapa de ordenação dos objetos é a principal, sendo responsável por 73% a 88% dos tempo. A etapa de construção das folhas inclui a ordenação dos envelopes de objetos colocados em cada folha, o que sempre pode ser feito em memória, porque em uma folha há, no máximo, apenas 146 objetos. A terceira etapa, de construção dos nodos não-folha, representa cerca de 1% ou menos do tempo, sendo, inclusive, difícil de identificar no gráfico.

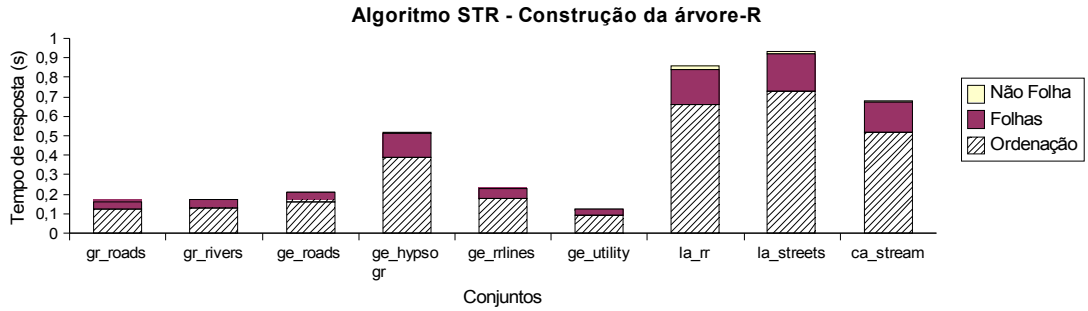



Figura 3.16 : Gráfico do tempo de construção de árvore-R, utilizando o algoritmo STR.

O conjunto *ca_street* não foi colocado no gráfico acima, porque seu tempo de construção é bastante alto. O tempo total é de 27,76 s, sendo 23,07s (88,29%) na etapa de ordenação; 3,07s (11,1%) na gravação de nodos folhas; e apenas 0,18s (0,01%) na criação dos nodos não-folha.

Como este é o conjunto com mais objetos, a memória disponível demonstrou ser fundamental para elevar o tempo de construção, pois a ordenação dos objetos é externa. Aumentar a memória é uma solução desejável, mas o tempo de construção só é reduzido quando torna-se possível empregar um algoritmo de ordenação interna.

Por exemplo, para o conjunto *la_street*, isto ocorre quando a memória é capaz de manter 770 blocos, reduzindo o tempo de construção de 1,46s para 0,84s. O número de operações de disco também é reduzido de 4.933 para 4.165. Para o conjunto *ca_street*, são necessários 13.300 blocos para realizar a ordenação em memória. O tempo de construção diminui 40%, para 15,77s.

A tabela 3.8 mostra o tempo de construção das árvores-R, o tempo de junção espacial pelo algoritmo STT, o tempo total do algoritmo *Build&Match* e porcentagem de tempo que representaria a construção da árvore-R no total. Pelos dados da tabela, uma situação difícil pode ocorrer quando o conjunto não indexado possuir muitos objetos, pois a sua construção demanda grande esforço computacional, tornando o tempo de resposta do algoritmo muito alto.

Tabela 3.8 : Tempo de construção e junção para alguns pares de conjuntos.

| | ***Tempo de construção (s)*** | ***Tempo de junção (s)*** | ***Tempo total (s)*** | ***% da construção*** |
|---|---|---|---|---|
| *gr_roads* | 0,16 | 0,56 | 0,72 | 22,2% |
| *gr_rivers* | 0,17 | | 0,73 | 23,3% |
| *ge_hypso* | 0,52 | 0,50 | 1,02 | 51,0% |
| *ge_roads* | 0,23 | | 0,73 | 31,5% |
| *la_rr* | 0,86 | 6,08 | 6,94 | 12,4% |
| *la_streets* | 0,93 | | 7,01 | 13,3% |
| *ca_stream* | 0,68 | 52,19 | 52,87 | 1,1% |
| *ca_street* | 27,97 | | 80,16 | 34,9% |

## 3.5 Algoritmo ISSJ

O algoritmo ISJ, proposto por Jacox e Samet (JACOX, 2003), baseia-se fundamentalmente em duas etapas: ordenação dos conjuntos por uma mesma coordenada e a realização do *plane-sweep* sobre estes conjuntos. Se um dos conjuntos já estiver ordenado, bastaria ordenar o segundo conjunto. Se ambos já estiverem ordenados, realiza-se apenas o *plane-sweep*.

Na sua implementação, o algoritmo foi modificado para incluir a divisão por faixas do algoritmo SISJ, resultando em um algoritmo diferente, chamado ISSJ (*Iterative Stripped Spatial Join*).

Para este novo algoritmo, o número esperado de operações de E/S é idêntico ao ISJ, sendo a etapa de ordenação responsável pela maioria das operações:

$$disco_{ISSJ} = 2(b^A + b^B) + 2b^A \times \lceil \log_M(b^A) \rceil + 2b^B \times \lceil \log_M(b^B) \rceil$$

O processamento é reduzido, pois a divisão em faixas resulta na realização de vários *plane-sweep*, um para cada faixa, e cada um processando um número menor de objetos. A ordem de desempenho do algoritmo é

$cpu_{ISSJ} = O(c_{Reg} + r(n^A + n^B)\log\frac{r(n^A + n^B)}{\phi})$ , onde $l$ representa o número de faixas e $c_{Reg}$ é o número de comparações em um espaço dividido regularmente, obtido através da expressão (4). Na figura 3.17 pode-se ver gráficos com o desempenho previsto para três pares de conjuntos, quanto as operações em memória e os acessos a disco. Em memória, mantendo o número de partições, por consequência, o fator de replicação, o tempo de processamento é constante. Quanto ao número de acessos a disco, este sofre redução sempre que se há memória disponível para ordenar o conjunto.

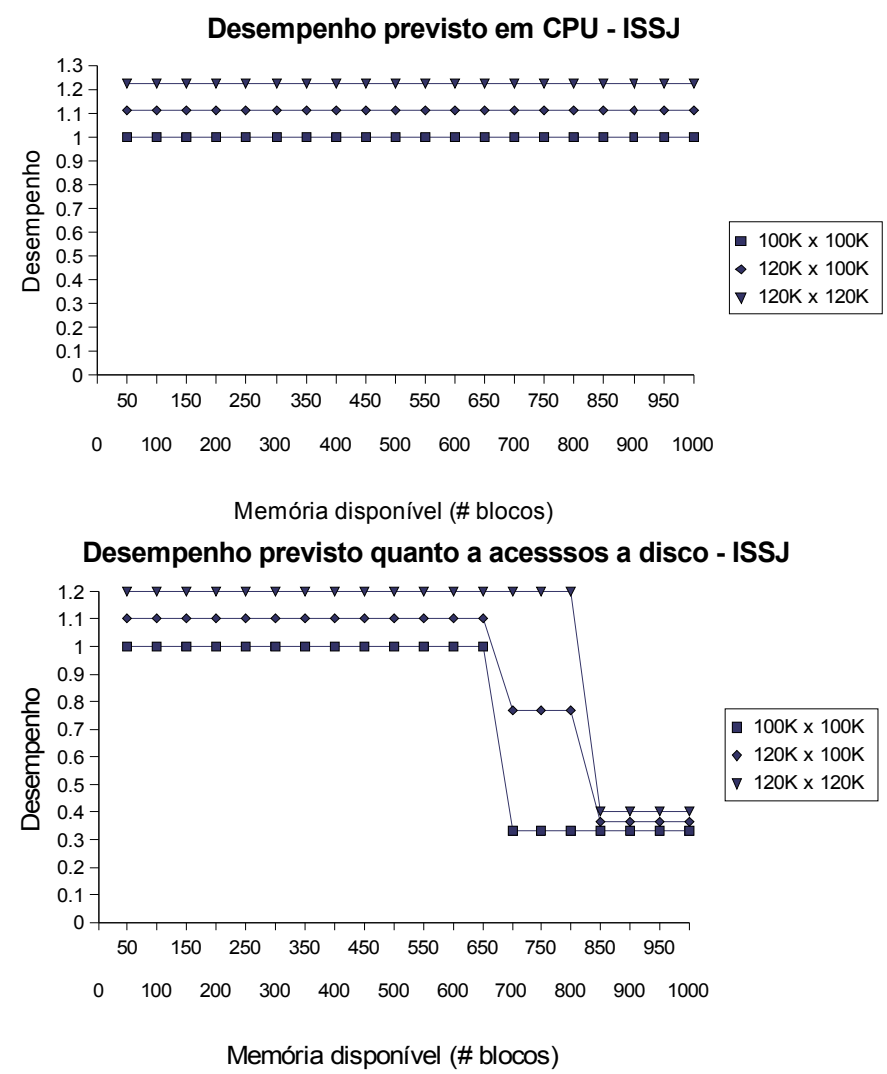



Figura 3.17 : Gráficos de desempenho previsto, quanto a processamento em memória e acessos a disco para o algoritmo ISSJ.

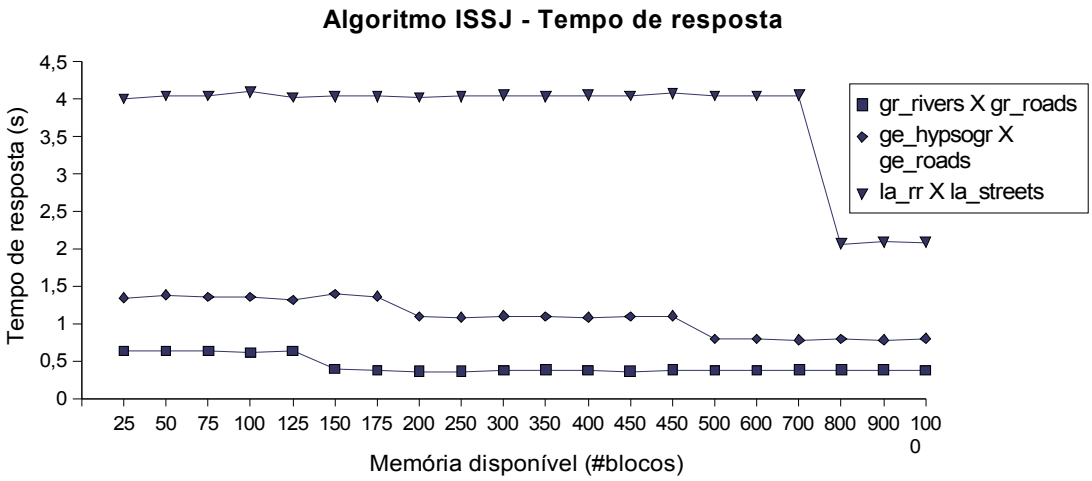



Figura 3.18 : Gráfico do tempo de resposta do algoritmo ISSJ, para três junções espaciais, variando a memória disponível.

O gráfico da figura 3.18 mostra o tempo de resposta do algoritmo ISSJ para três junções espaciais diferentes, variando a memória disponível, e incluindo a ordenação dos dois conjuntos. Na medida em que torna-se possível ordenar os conjuntos por um algoritmo interno, o tempo de resposta diminui sensivelmente, formando uma curva em degraus. Este gráfico combina adequadamente com os da figura 3.17, demonstrando que a previsão está de acordo com os valores medidos.

O gráfico da figura 3.19 mostra o tempo de execução de cada uma das etapas do algoritmo: ordenação do conjunto *A*, ordenação do conjunto *B* e junção pelo *plane-sweep*, para a junção entre os conjuntos *ge_hypso* e *ge_roads*. Nele, fica bem visível o ponto de redução devido à ordenação interna do conjunto *ge_roads*, depois à ordenação interna do conjunto *ge_hypso*. O tempo de execução do *plane-sweep* diminui porque os blocos de dados do segundo conjunto estão presentes no *buffer*, eliminado operações de E/S.

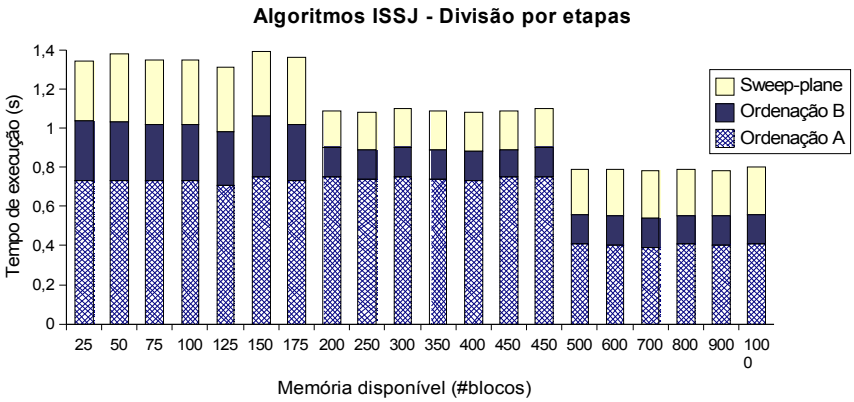



Figura 3.19 : Gráfico do tempo de execução de cada etapa do algoritmo ISSJ, para a junção *ge_hypso X ge_roads*, variando a memória disponível.

Como a leitura e gravação dos arquivos é sequencial, se considerada isoladamente, o tempo de operações de E/S representa apenas entre 1 e 2% do tempo total de execução, sendo as operações de CPU responsáveis, portanto, por 98 a 99% do tempo total.

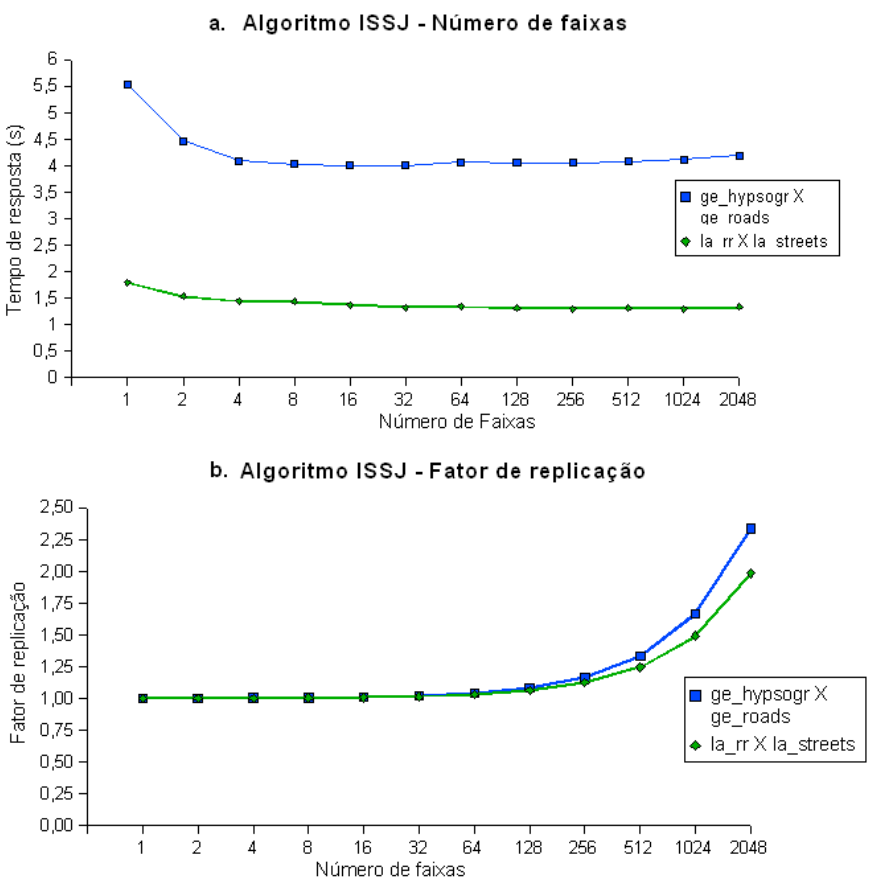



Figura 3.20 : Gráficos do tempo de resposta e fator de replicação, variando o número de faixas no algoritmo ISSJ, para duas junções espaciais.

Os gráficos na figura 3.20 mostram o efeito da variação do número de faixas no tempo de resposta (acima) e fator de replicação dos objetos (abaixo). Utilizar apenas uma faixa significa executar o algoritmo ISJ, como proposto originalmente. Aumentando o número de faixas, o tempo de processamento é reduzido até um ponto de estabilização, onde o aumento do fator de replicação, que aumenta o número de objetos processados, compensa o aumento de faixas. Na verdade, a junção entre *la_rr X la_street*, tem o ponto de inflexão com 256 faixas. Depois começa a aumentar o tempo de resposta, efeito do fator de replicação.

No gráfico b da figura 20, pode-se ver que o fator de replicação cresce de maneira exponencial. A junção com fator de replicação maior indica que envolve objetos maiores.

Embora seja difícil prever o ponto ótimo, no qual se obtém o menor tempo de resposta possível, como heurística, pode-se estabelecer que o número de faixas não deve ser menor que 4 e deve ser proporcional ao número de objetos envolvidos, aumentando na medida em que o número total de objetos cresce. O valor *default* escolhido para todos os testes foi de 16 faixas.

O número de objetos, quando mantido o número de comparações entre objetos, afeta o algoritmo de modo linear, como pode ser visto na figura 3.21. Nestes testes, a memória disponível utilizada foi sempre de 10.000 blocos, suficiente para ordenar os conjuntos em memória.

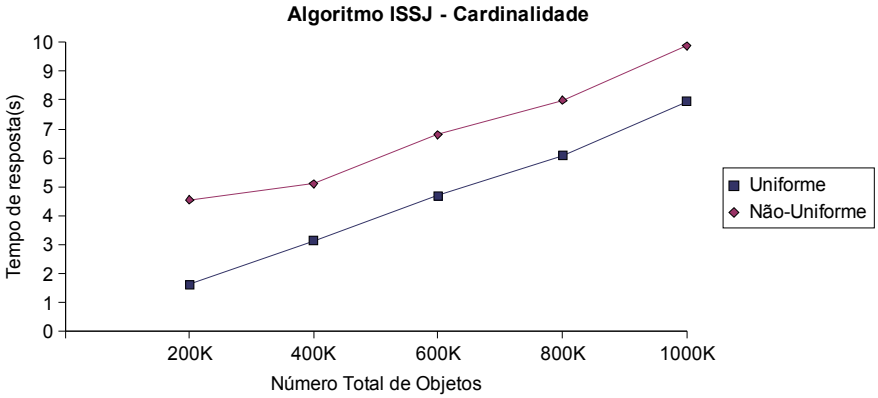



Figura 3.21 : Tempo de resposta do algoritmo ISSJ alterando a cardinalidade dos conjuntos.

Aumentar a densidade dos conjuntos resulta em aumento do tempo de resposta do algoritmo, embora para conjuntos com distribuição uniforme o impacto seja quase imperceptível, devido, apenas, a um incremento do fator de replicação, como pode ser visto na figura 3.22. Já para conjuntos não-uniformes, a taxa de crescimento é mais significativa, pois algumas faixas possuem mais objetos que outras, sendo que esta distribuição desigual também se reflete de maneira mais importante, com cada vez mais objetos em faixas já sobrecarregadas.

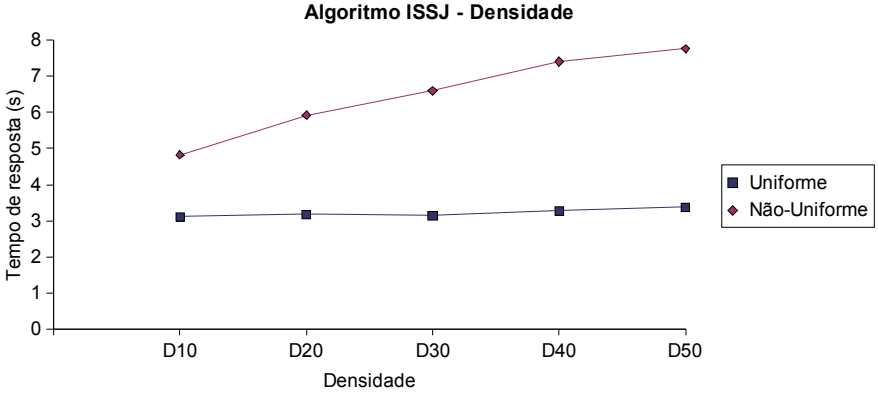



Figura 3.22 : Gráfico do tempo de resposta do algoritmo ISSJ alterando a densidade dos conjuntos de dados.

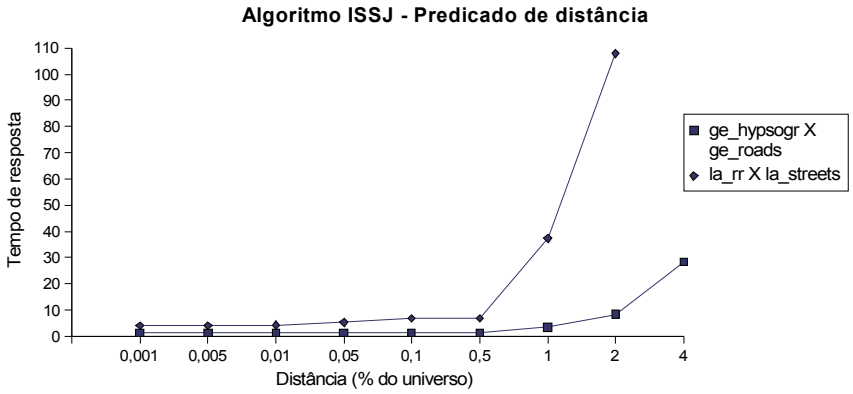



Figura 3.23 : Gráfico do tempo de resposta do algoritmo ISSJ, para o predicado de distância mínima entre objetos, variando a distância.

A figura 3.23 mostra o gráfico do tempo de resposta do algoritmo ISSJ, para o predicado de distância mínima entre objetos. Com o aumento da distância entre os objetos, aumenta o tempo de resposta, sendo este devido a etapa de *plane-sweep*, pois a ordenação dos conjuntos mantém-se estável. O incremento deve-se ao aumento do fator de replicação e à densidade dos objetos.

## 3.6 Cenários de execução

Para comparar o desempenho dos algoritmos, alguns cenários de execução podem ser definidos, a partir de variáveis de configuração do ambiente. Para cada variável, alguns valores são estabelecidos. Em cada cenário, o valor de uma das variáveis é alterado.

Para a *cardinalidade* considera-se a soma do número de objetos dos dois conjuntos. Foram definidos três valores:

- Pequenos: até 100.000 objetos
- Médios: entre 100.000 e 500.000 objetos
- Grandes: acima de 500.000 objetos

Para a *memória disponível* também foram definidos três valores diferentes:

- Pequena: até 100 blocos, ou seja, 400 Kb.
- Média: entre 100 e 500 blocos, correspondendo a 400Kb até 2Mb.
- Grande: acima de 500 blocos, portanto, de 2Mb.

O critério de divisão para cardinalidade de conjuntos depende de fenômenos geográficos, tendendo a permanecer estável ao longo do tempo. Por exemplo, a hidrografia de uma região pouco é alterada, assim como a cardinalidade dos conjuntos que a representam. Já o critério de divisão para memória disponível deve ser alterado constantemente, com a evolução de tecnologia de memória.

A figura 3.24 mostra o tempo de resposta dos três algoritmos principais: STT, PBSM e ISSJ, para quatro junções espaciais entre conjuntos reais. O algoritmo de laços aninhados não está incluído por apresentar tempo de resposta muito superior aos demais. Todas junções se enquadram como sendo de cardinalidade pequena. O algoritmo PBSM apresenta o menor tempo de resposta para dados reais em memória pequena. Conforme aumenta a memória, o algoritmo ISSJ se torna mais vantajoso. O algoritmo STT apresenta o menor tempo de resposta para três diferentes junções: (a), (b) e (c). Neste caso, o STT é seriamente afetado pelo *fanout* das árvores-R, que neste conjunto de testes foi mantido em 73 entradas.

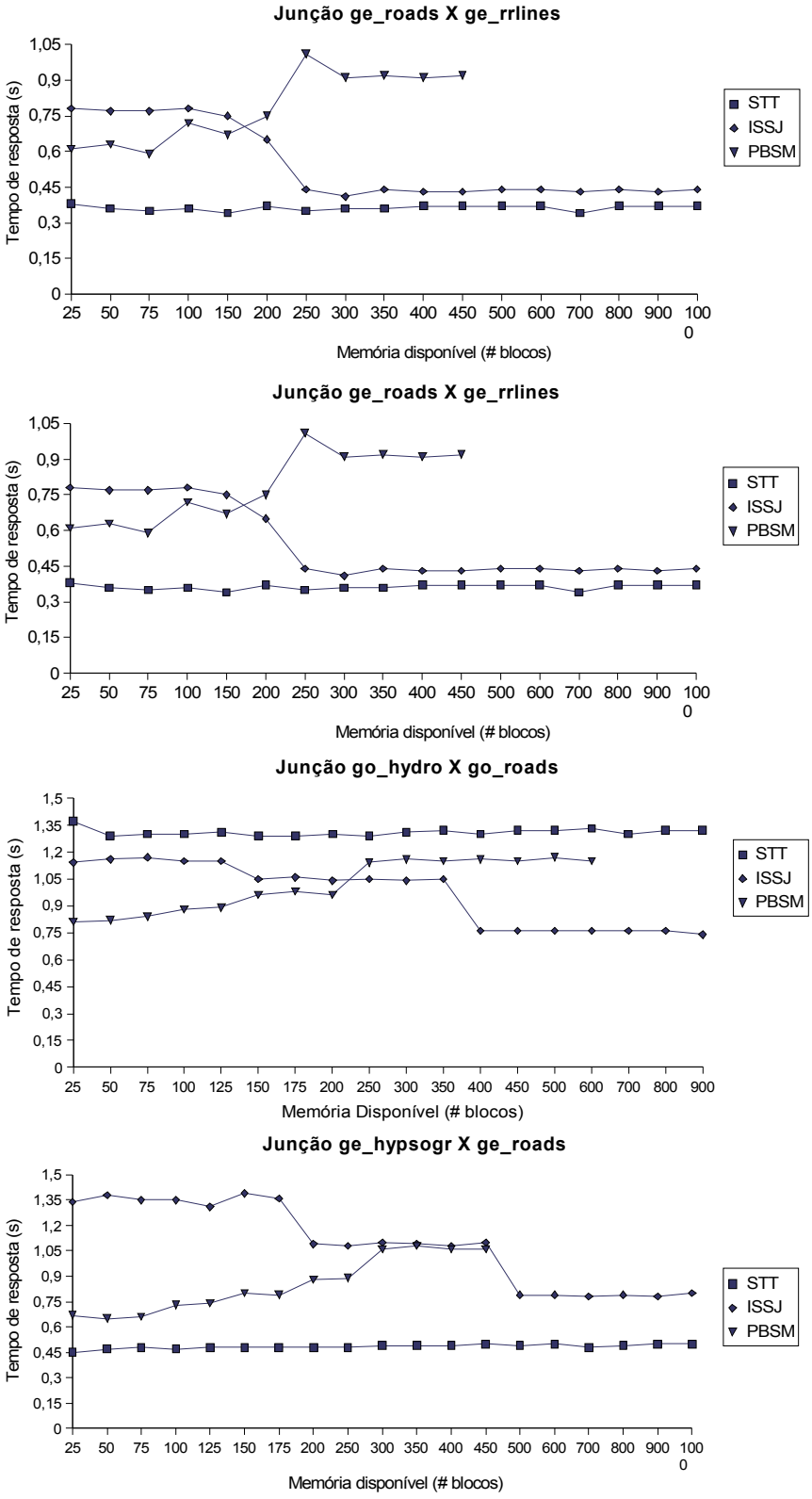
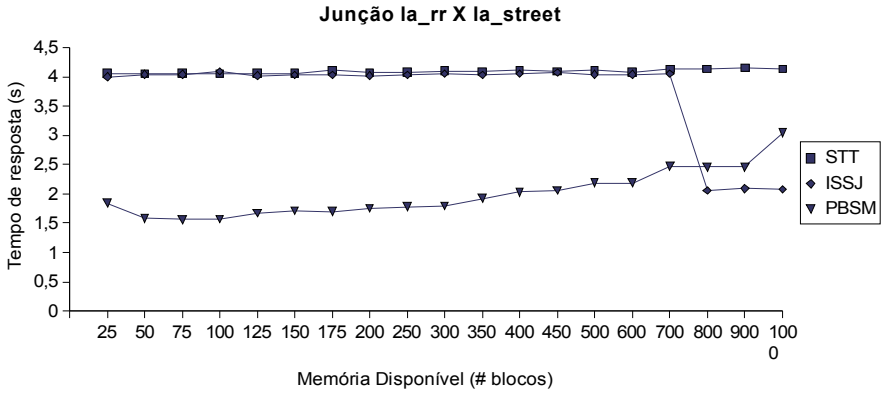
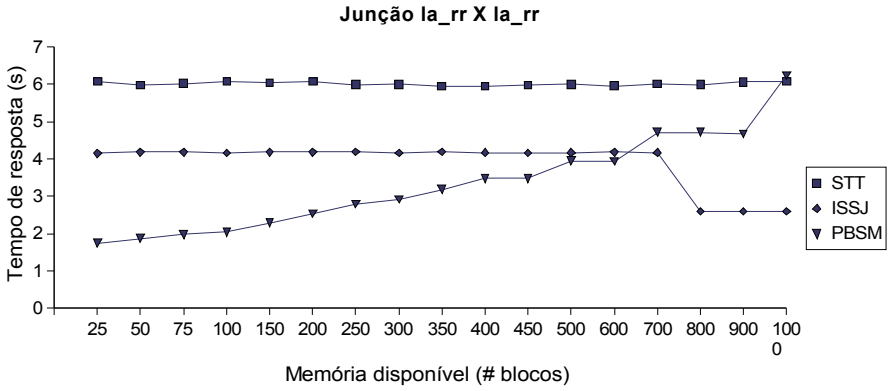
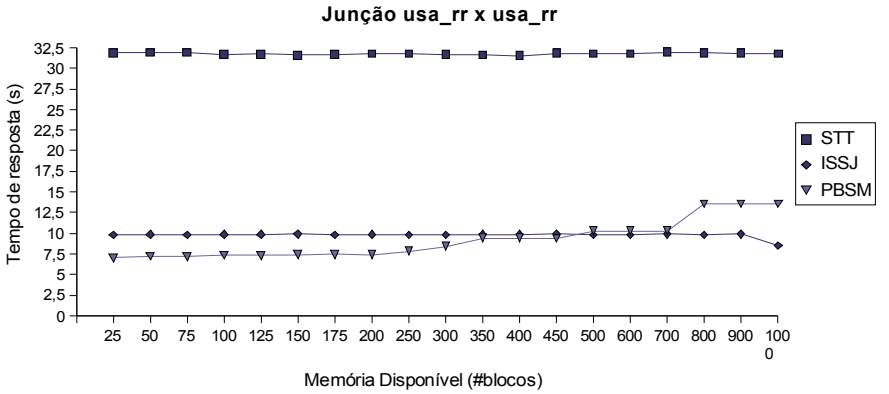


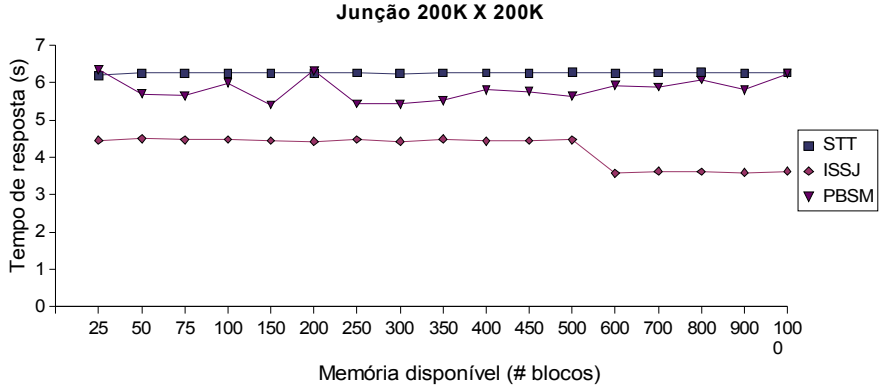

Figura 3.24 : Gráficos do tempo de resposta para junções de cardinalidade pequena.

Figura 3.25 : Gráficos do tempo de resposta para junções de cardinalidade média.




O tempo de resposta para três junções espaciais envolvendo conjuntos de cardinalidade média pode ser visto na figura 3.25. O algoritmo STT tem o maior tempo de resposta. O PBSM é a melhor escolha para memórias pequenas e médias, sendo o ISSJ o mais adequado para memórias grandes, onde é possível ordenar os conjuntos.

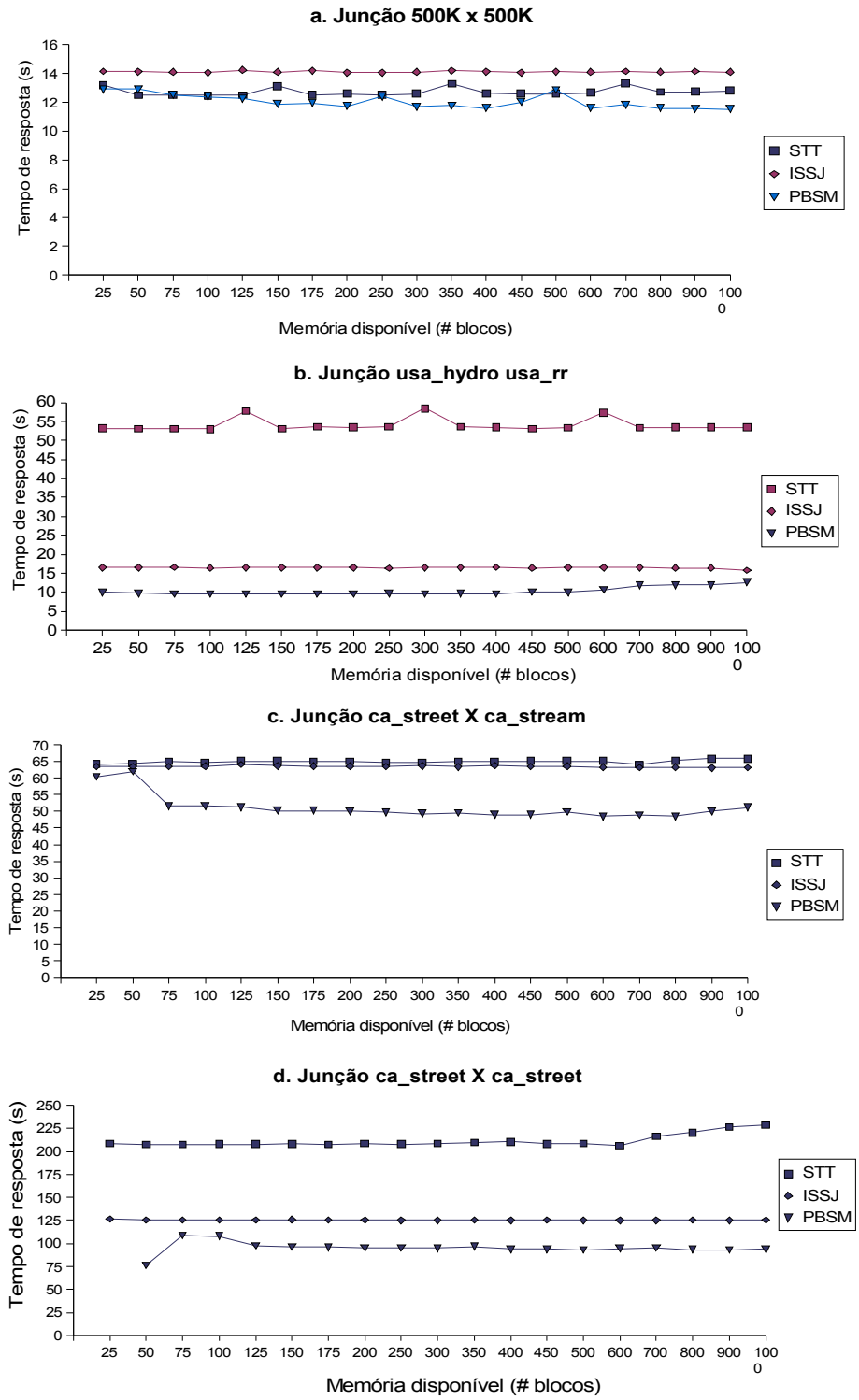



Figura 3.26 : Gráficos do tempo de resposta para junções de cardinalidade grande.

Os gráficos do tempo de resposta para conjuntos com cardinalidades grandes, podem ser vistos na figura 3.26. O algoritmo PBSM apresenta o menor tempo de resposta, em todas situações, nas junções espaciais (b), (c) e (d). Apenas no caso de conjuntos artificiais, a junção (a), mas com distribuição não-uniforme, que o PBSM alterna com o STT. O algoritmo ISSJ, para os tamanhos de memória mostrados nos gráficos, sempre realiza a ordenação dos conjuntos em memória externa. Com tamanhos bem mais

significativos, o ISSJ consegue ordenar os conjuntos em memória, reduzindo o tempo e tornando-se atrativo.

Para o predicado espacial de intersecção, a tabela 3.9, indica o algoritmo mais rápido em cada cenário, considerando várias junções espaciais que se enquadram em cada cenário.

Tabela 3.9 : Melhor algoritmo de junção espacial, para diferentes cenários de execução.

| *Cardinalidade* | *Memória pequena* | *Memória média* | *Memória grande* |
|---|---|---|---|
| Pequenos | STT | STT | STT |
| Médios | PBSM | PBSM | ISSJ |
| Grandes | PBSM | PBSM | PBSM |

O algoritmo PBSM apresentou o menor tempo de resposta em vários cenários, favorecido pela divisão dos objetos em partições. Quando a memória disponível aumenta, o algoritmo perde porque define partições maiores, com mais objetos, aumentando o tempo de resposta. O algoritmo ISSJ apresenta o melhor desempenho quando consegue ordenar em memória os conjuntos. O algoritmo STT é o ideal para conjuntos pequenos.

Tabela 3.10 : Melhor algoritmo de junção espacial, para diferentes cenários de execução, quando os dois conjuntos já estão ordenados.

| *Cardinalidade* | *Memória pequena* | *Memória média* | *Memória grande* |
|---|---|---|---|
| Pequenos | ISSJ | ISSJ | ISSJ |
| Médios | ISSJ | ISSJ | ISSJ |
| Grandes | PBSM | PBSM | ISSJ |

Caso os dois conjuntos estejam previamente ordenados, restando apenas a etapa de junção espacial pelo *plane-sweep*, o quadro se altera para as escolhas indicadas na tabela 3.10, sendo o algoritmo ISSJ o preferencial em quase todos cenários.

Em uma última situação avaliada, considerou-se a hipótese dos arquivos de partições do algoritmo PBSM existirem previamente, o que favorece este algoritmo, mesmo quando comparado ao ISSJ de arquivos já ordenados, como mostra a tabela 3.11.

Tabela 3.11 : Melhor algoritmo de junção espacial, para diferentes cenários de execução, quando os dois conjuntos já estão ordenados e os arquivos de partições já existem.

| *Cardinalidade* | *Memória pequena* | *Memória média* | *Memória grande* |
|---|---|---|---|
| Pequenos | PBSM | PBSM | ISSJ |
| Médios | PBSM | PBSM | ISSJ |
| Grandes | PBSM | PBSM | PBSM |

Para o predicado de distância mínima entre objetos, não houve alterações quanto aos cenários, pois todos algoritmos apresentam comportamento semelhante, aumentando o tempo de resposta à medida que a distância entre os objetos aumenta. Assim, para cenários semelhantes, a mesma escolha de algoritmos é válida.

## 3.7 Conclusão

A implementação dos algoritmos foi importante para verificar algumas questões que a análise de complexidade dos algoritmos deixou em aberto:

- o tempo de execução em CPU é a parcela mais significativa do tempo de resposta, para todos algoritmos, em todas situações.
- a relação entre o fator de replicação dos objetos, nos algoritmos PBSM e ISSJ, e o tempo de resposta da junção espacial.
- verificar a adequação da análise de complexidade à realidade, envolvendo conjuntos reais, com uma distribuição de objetos no espaço bastante irregular.

No próximo capítulo é apresentado um novo algoritmo que procura explorar as qualidades destes algoritmos, de modo a obter um tempo de resposta menor.

# 4 O ALGORITMO HHSJ

Este capítulo apresenta um novo algoritmo, chamado *Histogram-based Hash Stripped Join* (HHSJ), que procura solucionar algumas dificuldades dos algoritmos já existentes na literatura.

Implicitamente, o capítulo anterior apresentou as motivações do algoritmo HHSJ. O STT, embora eficiente e bastante estudado na literatura, apresenta um comportamento linear em relação à memória disponível. O PBSM sofre com a ocorrência de partições com *overflow* e o mau aproveitamento de memórias maiores, além da necessidade de criar os arquivos de partições a cada execução. O ISSJ introduz a idéia de dividir o espaço em faixas separadas, mas requer arquivos ordenados.

O algoritmo HHSJ baseia-se em três soluções para garantir um bom desempenho:

- Histogramas da distribuição dos objetos no espaço: com a divisão do espaço em células, pode-se contar o número de objetos em cada célula. Belussi et al (2004) propôs a idéia de criação destes histogramas, contendo o número de objetos que interseccionam cada célula, utilizando-os para prever o número de pares de objetos que atenderão ao predicado de junção espacial. Uma vez criados, estes histogramas podem ser úteis também para orientar o particionamento do espaço, reduzindo ou eliminando a ocorrência de partições com *overflow*.
- Armazenamento em arquivos *hash*[5]: o algoritmo PBSM divide os objetos em células, gerando novos arquivos de partições a cada execução. Uma alternativa possível é manter os objetos em arquivos organizados por *hash*, mapeando as células para *buckets*, o que permite um acesso direto aos objetos e evitaria a repetida criação de partições.
- Utilização de faixas: a idéia de divisão por faixas, como utilizada pelo ISSJ, para reduzir o processamento, sem acrescentar operações de E/S, demonstrou ser bastante eficaz.

Este capítulo descreve, inicialmente, a criação dos histogramas e arquivos *hash*, necessários para realizar a junção espacial. Após, é realizada uma análise de complexidade do algoritmo, que é comparado com os demais, tanto de maneira analítica quanto através de resultados de testes.

[5]Segundo Elmasri et al (2005, p.310), arquivo *hash* é o nome habitual dado a arquivos cuja "*organização primária baseia-se em técnicas de hashing, fornecendo acesso muito rápido aos dados sob certas condições de pesquisa*".

## 4.1 Construção de histogramas e arquivos *hash*

Para cada conjunto de objetos, é construído um histograma, chamado *HistQ,* que é mantido em uma árvore *quadtree* (GAEDE, 1998). Na verdade, cada nível da *quadtree* mantém um histograma, constituindo a *HistQ* em um conjunto de histogramas. Cada nodo da árvore contém um contador de objetos que interseccionam o segmento de espaço representado pelo subquadrante.

O nodo raiz da *HistQ* representa todo espaço em que os objetos estão localizados. Como no nodo raiz, há um contador do número de objetos no conjunto, ele mantém a cardinalidade do conjunto. Cada um dos subquadrantes define uma célula onde há objetos. Objetos que, em um mesmo nível, intersecionam mais de um segmento do espaço, são contados uma vez em cada nodo que interseccionam. Deste modo, a soma dos contadores dos quatro nodos filhos pode ser maior que o contador no respectivo nodo pai da árvore, devido a estes objetos transpassantes.

A *HistQ* é construída de maneira recursiva. Quando um novo objeto é inserido na base de dados, o contador do nodo raiz é incrementado em um. O nodo de cada subquadrante espacialmente interseccionado pelo envelope do objeto é visitado, sendo seu contador incrementado. Os filhos deste nodo que possuem intersecção com o envelope do objeto também são visitados, até atingir o nível dos nodos folhas.

A figura 4.1 mostra dois objetos, $o_1$ e $o_2$. O objeto $o_1$ interseciona apenas um subquadrante, uma única chamada recursiva do algoritmo é realizada. Já o objeto $o_2$ intersecciona dois subquadrantes, gerando duas chamadas recursivas do algoritmo.

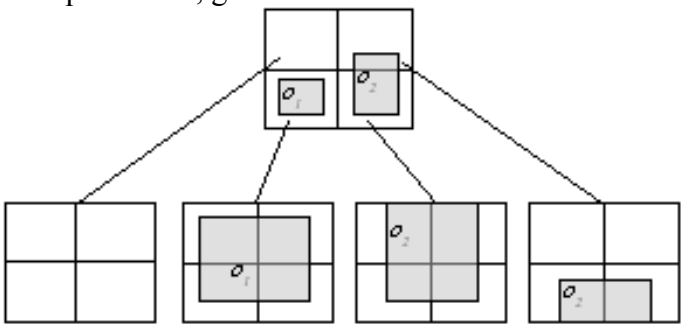



Figura 4.1 : Exemplo da construção do histograma *HistQ*.

Para agilizar a operação de junção espacial, um arquivo *hash* é criado para cada conjunto de objetos. Este arquivo é organizado em *buckets*, assumindo uma organização de *hash* estático (ELMASRI, 2005. p. 313)(KORTH, 2005, p.415 ). Para cada objeto, é gravado um descritor, contendo seu OID, envelope e um ponteiro para a descrição geométrica completa.

A função *hash* associa cada nodo folha da *HistQ* a um *bucket* específico, e cada *bucket* do arquivo *hash* é associado a somente um nodo folha. Se um objeto transpuser dois ou mais quadrantes da *HistQ*, seu descritor é replicado em dois (ou mais) *buckets*. O número de objetos replicados é definido durante a criação do arquivo *hash*, e depende da altura da *quadtree*. Isto permite ao usuário controlar o fator de replicação e, caso seja muito alto, ajustar reduzindo a altura da árvore.

A primeira etapa na construção do arquivo *hash* é, portanto, definir o número de *buckets* necessários, que pode ser feito dividindo a cardinalidade do conjunto pelo número de objetos que pode ser mantido em um *bucket,* seu *fanout.*

$$Buckets = \lceil \frac{n}{Fanout} \rceil$$

A altura da *HistQ* pode ser definida como, no mínimo:

$$h = \lceil \log_4 Buckets \rceil$$

O número de *buckets* é, então, definido em $4^h$. Por exemplo, para um conjunto com $n$ = 120.000, ocupando um bloco de disco com 4096 *bytes* para armazenar um *bucket*, temos um *fanout* de 146. O número mínimo de *buckets* é 822, indicando 5 níveis para a *HistQ*. O número inicial de *buckets* passa a ser, então, de 1024 ($2^5$). Este acréscimo no número de *buckets* reduz bastante a probabilidade de ocorrência de *buckets* com *overflow*, mesmo com a distribuição irregular de objetos no espaço e o surgimento de réplicas de objetos. No caso de *overflow*, o tratamento de colisões é por listas encadeadas de *bucket* (ELMASRI, 2005, p.313).

## 4.2 Junção espacial no HHSJ

Para realizar a junção espacial, o HHSJ, inicialmente, carrega para memória os histogramas dos dois conjuntos. Com o objetivo de definir as partições, o algoritmo, então, percorre as *HistQ*, de maneira sincronizada, dos nodos folhas para a raiz, enquanto a soma dos contadores de objeto, mantidos em cada nodo, é inferior ou igual à capacidade da memória. Desta maneira, garante-se que não ocorrerá partições com *overflow*, evitando uma etapa de reparticionamento. Se as duas *HistQ* possuem alturas diferentes, o algoritmo inicia percorrendo as duas árvores a partir do nível dos nodos folhas da menor árvore.

Para cada nodo não-folha, é possível concatenar dois ou três nodos filhos adjacentes, pois a soma de objetos pode ser menor que a memória disponível. A figura 4.2 ilustra as situações em que é possível esta combinação, para reduzir o número de partições e aproveitar melhor a memória disponível.

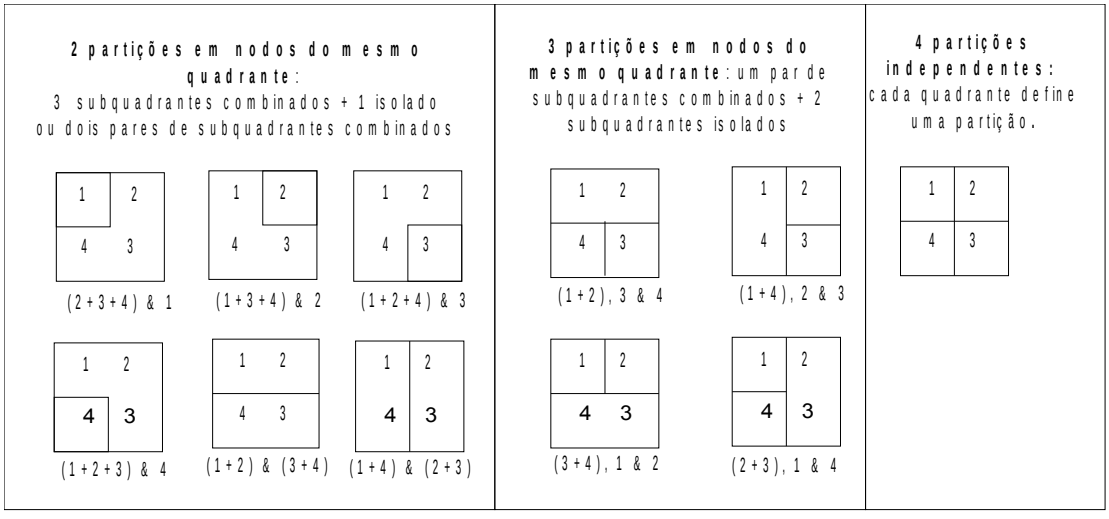



Figura 4.2 : Combinações possíveis de subquadrantes para definir partições.

Cada partição é espacialmente definida por um conjunto de quadrantes selecionados nas duas *quadtrees*. O mesmo conjunto de quadrantes é selecionado em ambas árvores para definir uma partição. Ao final do particionamento, um conjunto de partições e seus limites estão definidos, sem acessar os objetos, apenas os histogramas.

Considerando, por exemplo, $HistQ_A$ e $HistQ_B$ na figura 4.3, e supondo que 50.000 descritores de objetos possam ser mantidos em memória ao mesmo tempo, a seguinte seqüência de eventos ocorrerá durante o processo de particionamento. O número no círculo indica o número do nodo.

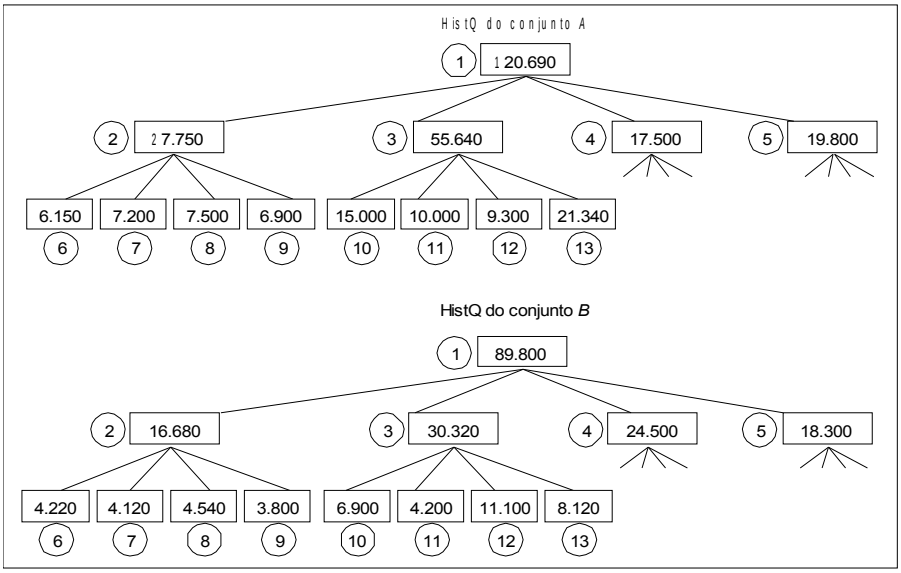



Figura 4.3 : *HistQ* de dois conjuntos de objetos.

1. No nível das folhas, a soma do par de nodos 6 (6.150 + 4.220) é 10.370, abaixo da capacidade da memória.
2. O mesmo ocorre para os pares de nodos 7, 8 e 9, que totalizam, respectivamente, 11.320, 12.040 e 10.700, todos abaixo do limite.
3. A soma dos quatro pares é computada, resultando 44.430, que é, também, menor que a capacidade do *buffer* em memória. Isto indica que é possível agrupar todos quadrantes em uma única partição, mas, antes, o algoritmo testa outras possibilidades de agrupamento.
4. O número de objetos no par de nodos 10 é 21.900, abaixo do limite.
5. O mesmo ocorre para os pares de nodos 11, 12 e 13, que somam 14.200, 20.400 e 29.460, respectivamente.
6. A soma de todos pares (10 até 13) é 85.960, acima do limite, indicando que não é possível agrupar todos em uma única partição.
7. Combinações de pares de quadrantes são testadas, como mostra a figura 4.2, e a combinação escolhida é dos nodos (10+11) e (12+13), porque o número de objetos em cada uma é 36.100 e 49.860, abaixo do limite.
8. O algoritmo procede da mesma maneira com os nodos 14 até 17 e 18 até 21, que não são mostrados na figura, mas existem. Considere que, nos dois casos, os quadrantes podem ser agrupados.
9. O algoritmo, então, sobe um nível, considerando os nodos 2, 3, 4 e 5. Para o nodo 3, a situação já está resolvida com o particionamento no nível inferior.

10. A soma dos pares de nodos 2 é 44.430, dos nodos 4 é 42.000, e, dos nodos 5 é 38.100, todos abaixo do limite.

11. O algoritmo tenta, então, combinar os nodos 2, 4 e 5, de acordo com a figura 4.2, mas todas alternativas excedem a capacidade de memória. Portanto, cada nodo define uma partição diferente.

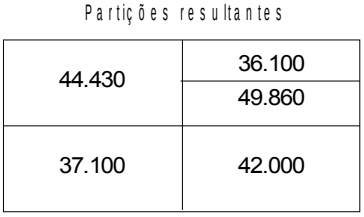



Figura 4.4 : Partições resultantes a partir das *HistQ* da figura anterior.

Na figura 4.4 podem ser vistas as partições resultantes, com a correspondente divisão do espaço.

```
1.  Procedure HHSJ(A, B: conjuntos de objetos
espaciais)
2.  quadA, quadB: quadtree
3.  lim: MBR
4.  nniveis, numpart: integer
5.  c_buckets: conjunto de inteiros
6.  partSet: conjunto de partições
7.  p: partição
8.  b: integer

9.  Begin
10. quadA = ler HistQ do conjunto A
11. quadB = ler HistQ do conjunto B
12. lim = limites do espaço
13. nniveis = min( altura da quadA, altura da quadB)
14. numpart = 0
15. partSet = Função de definição das partições
16. For each p in partSet do
17.    If length(p) > bufferSize then
18.          reparticionar p
19.    End-if
20. End-for
21. For each p in partSet do
22.    c_buckets = identificar buckets relativos à p
23.        s = número de faixas
24.    For each b in c_buckets
25.          ler MBRs do conjunto A, bucket b
26.          ler MBRs do conjunto B, bucket b
27.    End-for
28.       For each s
29.             Ordenar objetos da faixa
30.       Realizar o sweep-plane e identificar pares de
objetos que atendem ao predicado espacial
31.       If (par de objetos satisfaz ao RPM and
32.             é diferente do par anterior) then
33.          gravar no Resul Set
34.        End-if
35.       End-for
36. End-for
37. End-proc
```

Figura 4.5 : Algoritmo HHSJ

A figura 4.5 mostra o algoritmo para executar a junção espacial. O procedimento recebe por parâmetro os dois conjuntos de objetos (linha 1). A primeira etapa é carregar

as respectivas *HistQ* para memória (linhas 10 e 11). Após inicializar os limites do espaço e o número de níveis (linhas 12 e 13), é chamado o algoritmo para definir as partições, percorrendo as duas quadtrees de maneira sincronizada (linha 15). Este procedimento retorna um conjunto de partições e altera a variável *numpart,* que mantém o número de partições. O laço nas linhas 16 a 20 garante que nenhuma partição possui mais objetos que o limite da memória. Se isto ocorrer, a partição deve ser redividida (linha 18). O laço que nas linhas 21 a 36 realiza a junção espacial em cada partição separadamente. Primeiro, identifica-se os buckets abrangidos por uma partição (linha 22) e, após, carrega-se os objetos presentes em cada bucket para memória (linhas 24 a 27). Para cada faixa, é necessário ordenar os objetos da faixa (linha 29) e realizar a técnica de *plane-sweep*, para identificar pares de objetos que atendam ao predicado espacial (linha 30). Se o predicado espacial for verdadeiro para um par de objetos, faz-se o teste para evitar duplicatas (linhas 31 e 32) e, se passar, grava-se no conjunto de resultados (linha 33).

Para evitar a duplicação de pares de objetos no conjunto resposta, duas técnicas são utilizadas. A primeira técnica evita a duplicação quando réplicas de objetos transpassantes são alocadas na mesma partição. Neste caso, antes do objeto ser processado pelo algoritmo de *plane-sweep*, é verificado se ele é igual ao anterior. Como, na entrada do *plane-sweep* os objetos já estão classificados, réplicas ficam em posições subsequentes.

A segunda é o *Reference Point Method* (DITTRICH, 2000), apresentado na seção 2.3.2. Um par de objetos só é incluído no conjunto resposta se, e somente se, o ponto superior esquerdo da intersecção entre os objetos estiver dentro da região do espaço da faixa que estiver sendo processada. Esta técnica evita a duplicação ocasionada por objetos transpassantes cujas réplicas são alocadas em partições ou faixas diferentes. O conjunto de respostas obtido é final, sem duplicações.

## 4.3 Análise de desempenho

O número de operações de E/S do algoritmo HHSJ depende, fundamentalmente, do armazenamento em arquivo *hash* dos conjuntos envolvidos. Para realizar a junção, o algoritmo carrega para memória os dois conjuntos de histogramas, que são arquivos pequenos, pois mantém apenas contadores, números inteiros de quatro *bytes*. O tamanho, em *bytes*, de uma *HistQ* de altura *h* é

$$Len_{HistQ} = 4 \times \sum_{i=0}^{h-1} 4^i$$

O número de blocos é função do tamanho em bytes dividido pelo tamanho do bloco.

$$Len_{HistQ} = \lceil \frac{Len_{HistQ}}{Len_{bloco}} \rceil$$

A tabela 4.1 mostra o tamanho da *HistQ*, em *bytes,* em número de blocos e o número máximo de objetos que é possível manter em *buckets* associados, para algumas alturas diferentes:

Tabela 4.1 : Tamanho de uma *HistQ* para altura entre 2 e 8.

| *Altura* | *Tamanho em bytes* | *Tamanho em blocos* | *Número máximo de objetos* |
|---|---|---|---|
| 2 | 20 | 1 | 584 |
| 3 | 84 | 1 | 2336 |
| 4 | 340 | 1 | 9344 |
| 5 | 1364 | 1 | 37376 |
| 6 | 5460 | 2 | 149504 |
| 7 | 21844 | 6 | 598016 |
| 8 | 87380 | 22 | 2392064 |

O número de blocos do arquivo *hash* é, em princípio, $4^h$. Porém, devido a ocorrência de *overflow*, deve-se somar o número de *buckets* de *overflow, representado por* $b_o$, totalizando $4^h + b_o$ Para realizar a junção espacial, cada *bucket* é lido uma única vez, o que permite prever que o número de operações de E/S do algoritmo é:

$$disco_{HHJS} = b^A_{HistQ} + b^B_{HistQ} + 4^{h^A} + b^A_o + 4^{h^B} + b^B_o \quad (23)$$

A complexidade do algoritmo é definida pelo algoritmo de *plane-sweep* realizado em memória. O número de objetos processados por vez é, no máximo, a capacidade da memória dividida pelo número de faixas, considerando uma distribuição uniforme de objetos entre as faixas. Ou seja, cada vez que o *plane-sweep* é executado, processa-se $M_o/\Phi$ objetos, resultando em uma complexidade de

$$O(\frac{c}{P\Phi} + \frac{M_o}{\Phi}\log\frac{M_o}{\Phi})$$

O *plane-sweep* é repetido, para cada partição, para cada faixa, ou seja, *P.l* vezes. O algoritmo apresenta, assim, uma ordem de complexidade de

$$O(c_{Irreg} + P\,M_o \log\frac{M_o}{\Phi})$$

O fator $P.M_o$ representa o número máximo de objetos que podem estar envolvidos na junção espacial, que é a soma do número de objetos de cada conjunto $(r^A n^A + r^A n^B)$, sendo *r* o fator de replicação de cada conjunto. O número de comparações entre objetos é expresso por $c_{Irreg}$, pois a divisão do espaço por partições é irregular. Para simplificar, pode-se utilizar um fator de replicação que represente aos dois conjuntos, de maneira proporcional

$$cpu_{HHSJ} = O(c_{Irreg} + r(n^A + n^B)\log\frac{M_o}{\Phi}) \quad (24)$$

## 4.4 Testes de desempenho do HHSJ

Para verificar o desempenho do algoritmo em diversas situações, ele também foi implementado no ambiente de testes, como um módulo adicional para realizar junção espacial. Antes, é necessário criar os arquivos *hash* e *HistQ*, que mantém os dados e histogramas. A criação de ambos é feita em uma única etapa. O tempo de criação,

utilizando um *buffer* de memória para 500 blocos, pode ser visto no gráfico da figura 4.6.

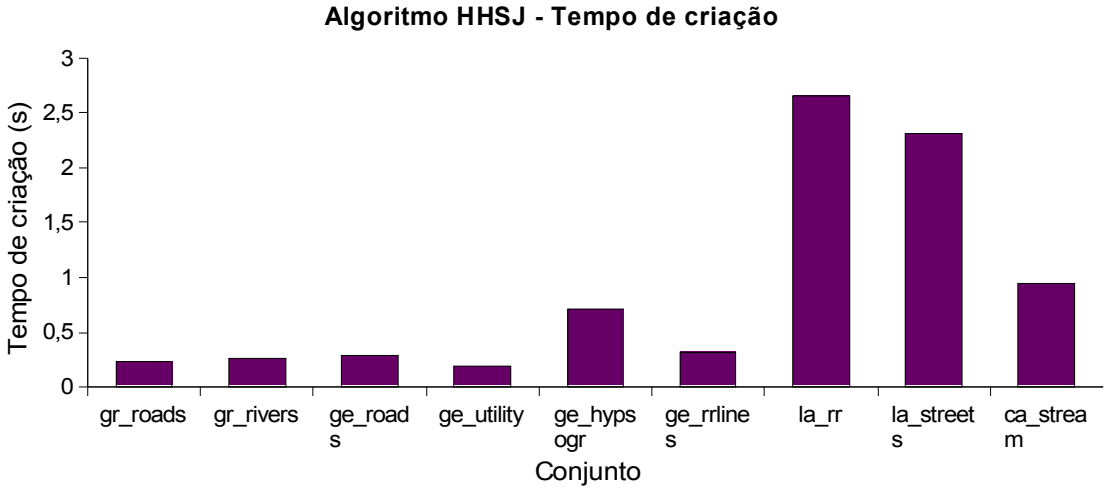



Figura 4.6 : Gráfico do tempo de criação do arquivo *hash* e *HistQ*.

O tempo de criação do conjunto *ca_street*, o maior deles, foi de 77,52s. Como a inclusão de objetos a *buckets* é aleatória, o mesmo *bucket* deve ser gravado diversas vezes. O *buffer* de memória é bastante importante, pois evita sucessivos acessos a disco.

O efeito da variação do tamanho do *buffer*, no tempo de criação de dois conjuntos, pode ser visto no gráfico da figura 4.7. Para o conjunto menor (*ge_rrlines*), o efeito não é perceptível, mas para o conjunto maior (*la_rr*), pode-se observar que a medida que o *buffer* aumentou, o tempo de criação diminui, em virtude da redução do número de operações de E/S, que neste caso varia 41.429 até 4.429.

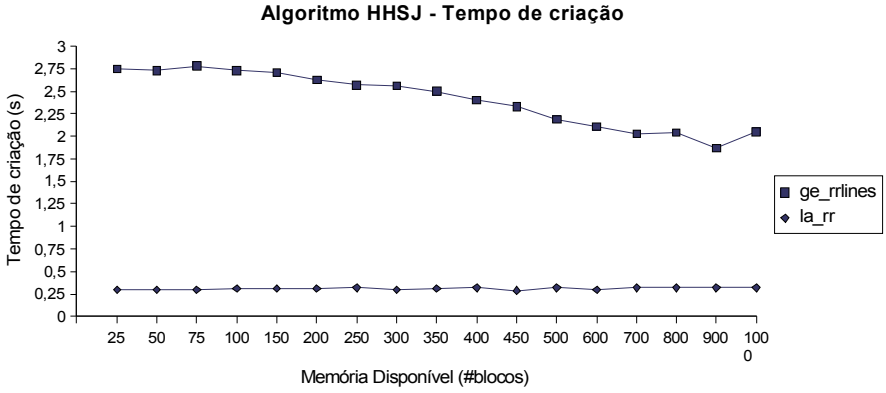



Figura 4.7 : Gráfico do tempo de criação variando a memória disponível, para dois arquivos.

Quando a junção espacial envolve um conjunto pequeno e outro grande, a diferença na altura das *HistQ* é significativa. Durante a junção, por falta de um histograma mais detalhado, o algoritmo de particionamento define partições muito grandes, resultando em *overflows* na maioria das partições, o que prejudicava o desempenho do HHSJ. A solução encontrada foi definir, para todos conjuntos, a altura da *HistQ* como, no mínimo, 5.

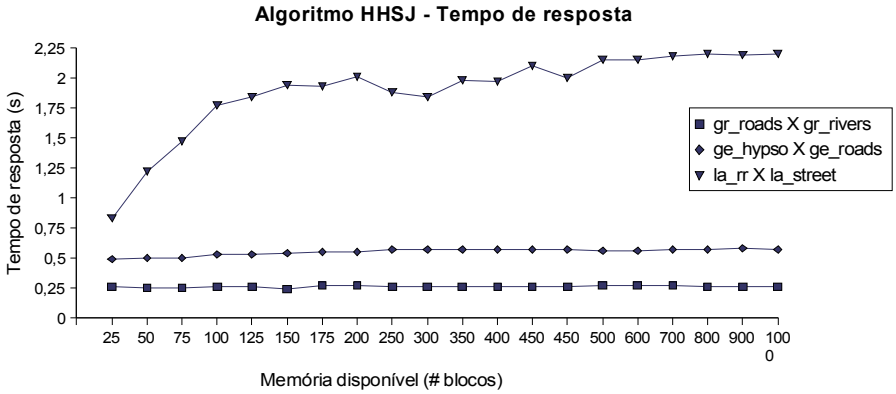



Figura 4.8 : Gráfico do tempo de resposta do algoritmo HHSJ, variando a memória disponível.

Uma vez criadas as estruturas necessárias, pode-se realizar junções espaciais. O tempo de resposta, utilizando o predicado de intersecção, para três pares de conjuntos, variando a memória disponível pode ser visto nas figura 4.8. Quando o número de objetos é pequeno, o algoritmo se mostrou bastante estável, aproveitando a divisão em faixas, para realizar sucessivos *plane-sweep* envolvendo poucos objetos. Porém, com conjuntos médios, como a junção entre *ll_rr* e *la_street*, já se pode notar que o tempo de resposta cresce, estabilizando após um certo ponto.

Para conjuntos grandes, como a junção entre *ca_street* e *ca_stream*, na figura 4.9, este ponto de estabilização não foi alcançado, mesmo com uma memória de 1000 blocos (4Mb). De qualquer modo, os menores tempo de resposta foram obtidos com o menor tamanho de memória disponível.

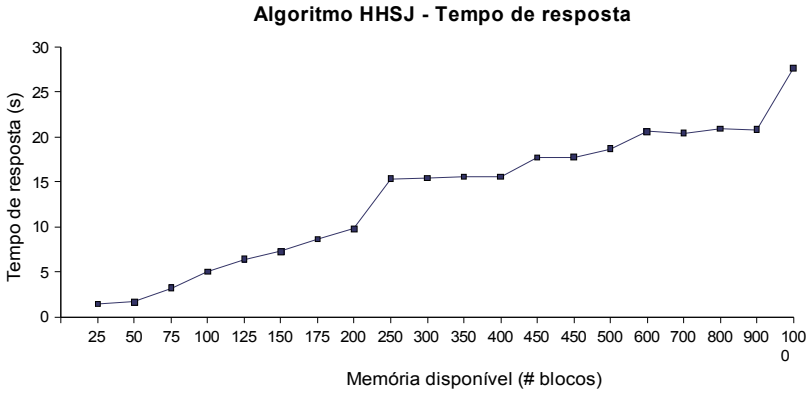



Figura 4.9 : Tempo de resposta para a junção entre *ca_street* e *ca_stream*, variando a memória disponível.

Tabela 4.2 : Tempo de operações de E/S e operações de CPU para o algoritmo HHSJ.

| ***Junção*** | ***Tempo E/S (s)*** | ***Tempo CPU (s)*** | ***Tempo total (s)*** | ***% Tempo CPU*** |
|---|---|---|---|---|
| *ge_roads X ge_hypso* | 0,072 | 0,388 | 0,46 | 84,3% |
| *la_rr X la_street* | 0,074 | 0,866 | 0,94 | 92,2% |
| *usa_rr X usa_rr* | 0,638 | 5,321 | 5,96 | 89,3% |

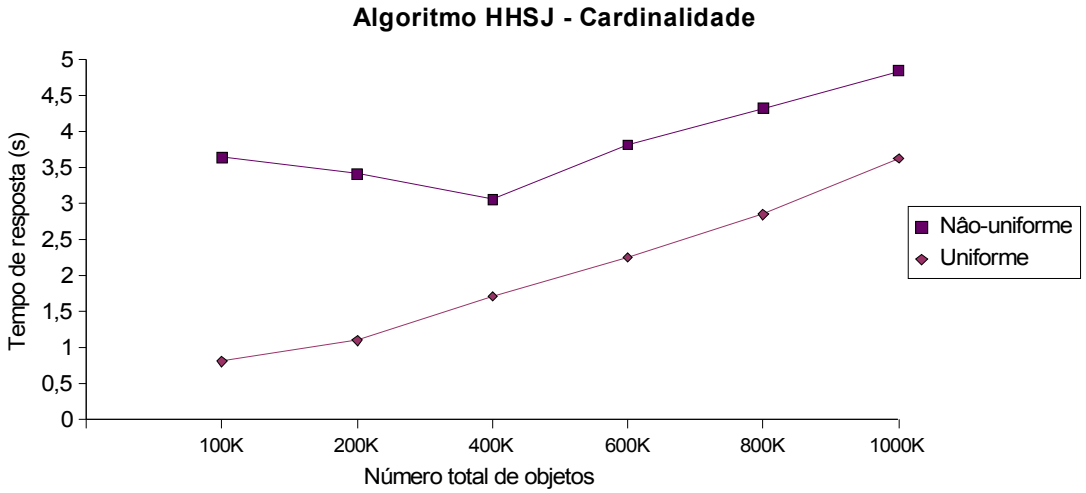



Figura 4.10 : Gráfico do tempo de resposta variando a cardinalidade dos conjuntos de objetos.

Este algoritmo, por reduzir o máximo possível o número de operações em CPU, apresenta, entre os algoritmos estudados, o tempo de operações de E/S proporcionalmente mais significativo. A tabela 4.2 mostra os resultados para algumas junções. Estes resultados referem-se a execuções utilizando 50 blocos de memória.

Para verificar o efeito do aumento da cardinalidade, mantendo-se, de maneira aproximada, o número de comparações, foram utilizados os mesmos conjuntos com os quais se traçou os gráficos incluídos nas figuras 3.5, 3.11 e 3.18 para os demais algoritmos. O gráfico na figura 4.10 mostra o tempo de resposta em cada caso. O decréscimo inicial deve-se ao aumento do número de partições, que reduz o número de comparações em memória. Este efeito, no entanto, não é suficiente para sustentar a redução quando o número de objetos aumenta.

O gráfico na figura 4.11 mostra o tempo de resposta quando a densidade dos conjuntos é aumentada. Para conjuntos não uniformes o crescimento é maior, porque torna áreas densas ainda mais densas, que exige um esforço de processamento significativo. Em conjuntos uniformes, a distribuição evita áreas sobrecarregadas, que consumem muito tempo de execução.

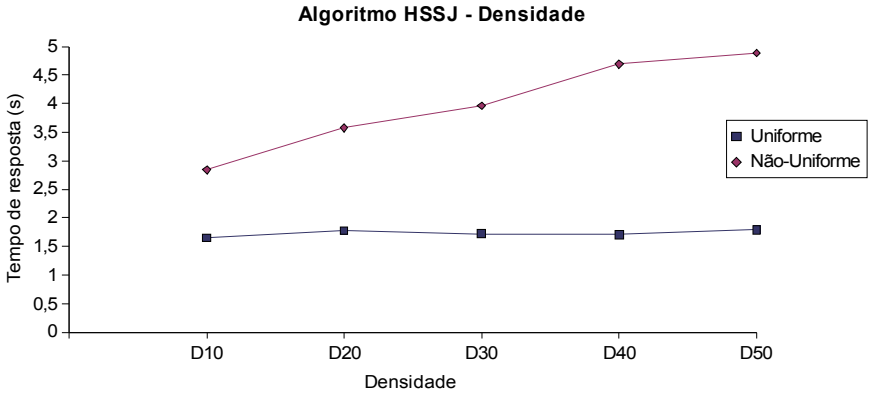



Figura 4.11 : Gráfico do tempo de resposta variando a densidade dos conjuntos, para o algoritmo HHSJ.

A figura 4.12 mostra o gráfico do tempo de resposta do HHSJ para o predicado de distância, alterando a distância mínima entre os objetos. Obviamente, o aumento do conjunto resposta implica no aumento do número de operações de E/S, mas o principal

incremento deve-se ao número de pares de objetos que são verificados. A ocorrência de réplicas de objetos devido à divisão do espaço em faixas também contribui, especialmente quando a distância passa a ser 4% do universo, porque o espaço está dividido em 32 faixas. Cada faixa abrange 3,125% do espaço. Assim, somando ao objeto a distância mínima, o fator de replicação é superior a 2, para todos conjuntos.

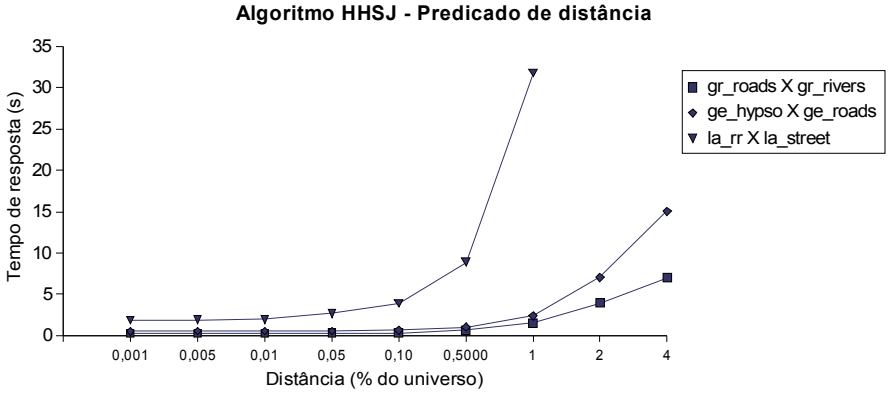



Figura 4.12 : Gráfico do tempo de resposta do algoritmo HHSJ, utilizando o predicado de distância entre objetos.

O efeito do número de faixas pode ser visto nos gráficos da figura 4.13. Há quatro casos de junção: o primeiro envolve os conjuntos *la_rr* e *la_streets*, com uma distância de 1%, o segundo com uma distância de 2%; o terceiro e quarto é a junção dos conjuntos *gr_hypso* e *gr_rrlines*, também com distâncias de 1% e 2% do espaço total. Aumentar o número de faixas, inicialmente, reduz, o tempo de resposta, porém, há um ponto a partir do qual o número de réplicas se torna tão importante que o tempo de resposta aumenta. O mesmo pode ocorrer utilizando o predicado de intersecção, porém, com o predicado de distância, o efeito ocorre com um número de faixas menor.

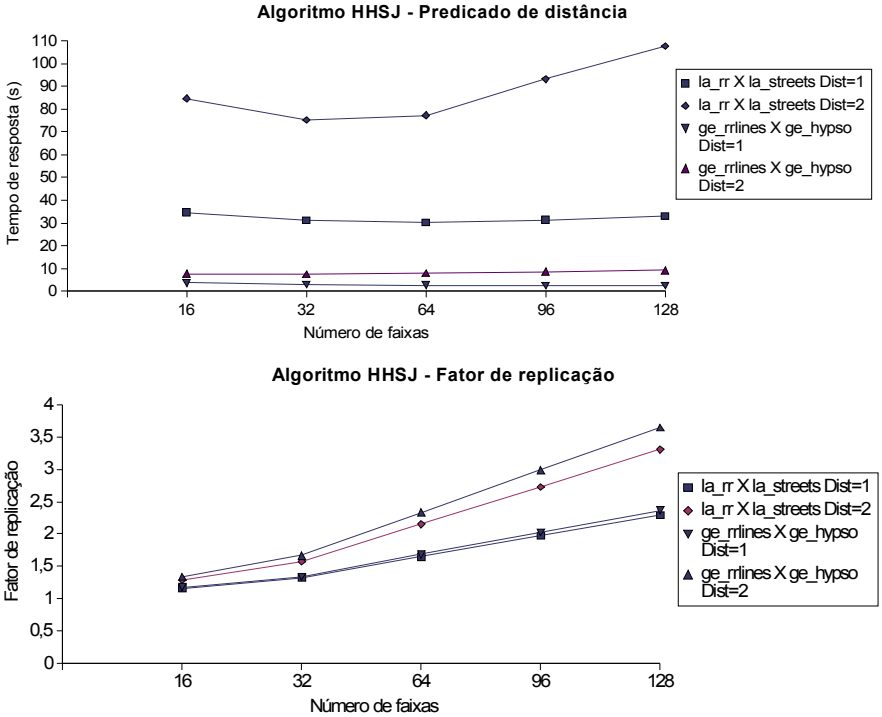



Figura 4.13 : Gráfico do tempo de resposta e fator de replicação dos algoritmo HHSJ, predicado de distância, variando o número de faixas.

## 4.5 Comparação com os demais algoritmos

Para realizar a comparação do HHSJ com os demais algoritmos é necessário recuperar expressões de desempenho e número de operações de E/S definidas no capítulo 2. A comparação é realizada com apenas alguns algoritmos, para não tornar o trabalho extenso.

O desempenho dos algoritmos baseados em ordenação, como o ISJ, no melhor caso, é

$$cpu_{ISJ} = O(c + n^A \log n^A + n^B \log n^B + (n^A + n^B)\log(n^A + n^B)) .$$

É simples verificar que $cpu_{HHSJ} < cpu_{ISJ}$, pois o algoritmo ISJ inclui a ordenação dos conjuntos. Mesmo que já estejam ordenados, na última parcela temos o logaritmo, onde $Mo/\Phi < n^A + n^B$., resultando um valor menor para o HHSJ. O fator de replicação do HHSJ, nos casos testados, manteve-se próximo 1, não representando um acréscimo que possa favorecer o ISJ.

A comparação com o algoritmo ISSJ também é relativamente simples, pois sua expressão é

$$cpu_{ISSJ} = O(c_{Reg} + r(n^A + n^B)\log(\frac{(r(n^A + n^B))}{\Phi}))$$

A expressão inclui a ordenação dos dois conjuntos. O fator de replicação, comum às duas expressões, no entanto, possui valores diferentes, pois no ISSJ ele é resultado apenas da divisão em faixas, enquanto no HHSJ também inclui a divisão em *buckets*. Assim, é justo pensar que $r_{ISSJ} < r_{HHSJ}$. Na prática, os dois valores são pouco superiores a 1. No entanto, o número de comparações entre objetos, tende a favorecer o HHSJ, pois o espaço é dividido em faixas e partições, enquanto o ISSJ apenas divide em faixas. Assim, se o número de faixas for idêntico, $c_{HHSJ} < c_{ISSJ}$. Além disso, o logaritmo indica a quantidade de objetos manipulados em cada faixa. Como esta classe de algoritmos é útil se a memória for menor que o total de objetos, $M_o < (n^A + n^B)$ , o HHSJ é favorecido. Na medida em que $M_o$ aumentar, poderá não compensar a diferença nos fatores de replicação, pois o algoritmo tende a colocar todos objetos em memória. A alternativa é limitar o uso da memória, mesmo que haja uma disponibilidade maior.

Em relação ao algoritmo PBSM, cuja expressão de desempenho é

$$cpu_{PBSM} = O(c_{PBSM} + r(n^A + n^B)\log r\frac{(n^A + n^B)}{P}) .$$

Um raciocínio semelhante pode ser realizado. Os fatores de replicação são diferentes, mas o número de objetos a cada execução do *plane-sweep*, com uma memória disponível menor, favorece o HHSJ, pois $M_o < r(n_A + n_B)$.

A comparação quanto ao número de operações de E/S não é simples de ser realizada, pois a expressão (30), do algoritmo HHSJ, envolve variáveis particulares, que não ocorrem em outros algoritmos. No mínimo, o número de *buckets* (blocos) do arquivo *hash* é $4^h$. Este é, em geral, maior que o número de blocos do mesmo arquivo quando manipulado pelos demais algoritmos. A diferença, representada por *diff*, entre o número de blocos dos arquivos de objetos e *hash* varia entre 1 e 4, por tratar-se de uma *quadtree*, em que o nível das folhas está sempre completo.

Na melhor hipótese, quando é possível ordenar os dois conjuntos em memória, o algoritmo ISSJ realiza $3(b_A+b_B)$ operações de E/S, ou seja, se *diff*>3, tem-se que $disco_{ISJ}<disco_{HHSJ}$. Porém, em outras situações, com memória insuficiente para realizar a ordenação, o número de E/S do algoritmo ISSJ cresce, favorecendo o HHSJ. Se os conjuntos já estiverem ordenados, o ISSJ realizam apenas um acesso aos dados, sendo mínimo em qualquer circunstância.

O algoritmo PBSM realiza $(2or^2+2r+1)(b^A+b^B)$ operações de E/S. Considerando que o fator de replicação é pouco superior a 1, e a porcentagem de partições com *overflow* ($o$) é baixa, pode-se esperar um número de acessos a disco pouco superior a $3(b^A+b^B)$. Portanto, para o HHSJ ter vantagem, *diff* deve ser menor que 3.

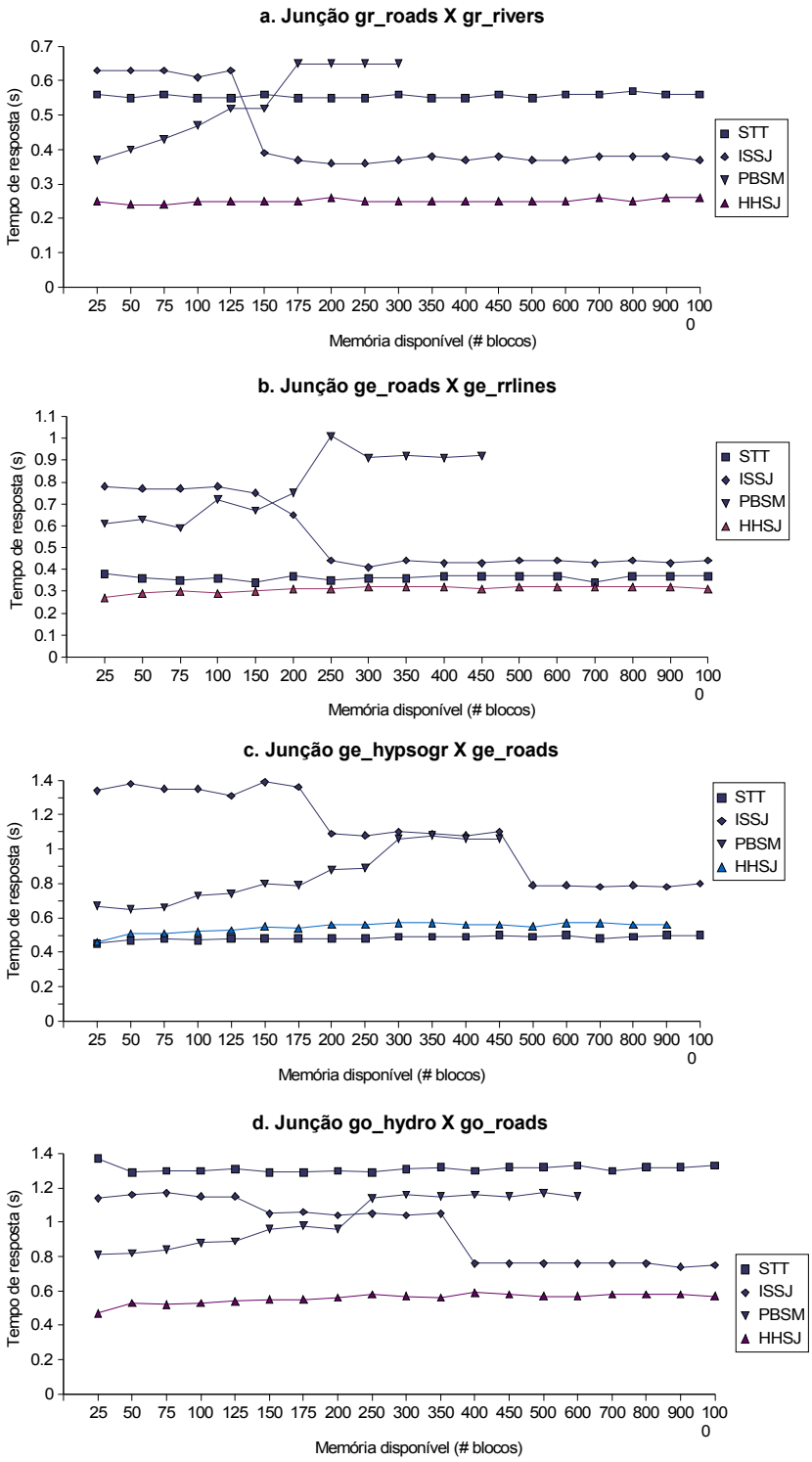



Figura 4.14: Gráficos do tempo de resposta dos quatro algoritmos, para diferentes junções espaciais de cardinalidade pequena.

A figura 4.14 mostra quatro gráficos de junção espacial, de diferentes conjuntos até 100.000 objetos, com o tempo de resposta de cada algoritmo, variando a memória disponível. O algoritmo HHSJ apresenta menor tempo de resposta em três junções (a),(b) e (d), e é um pouco superior ao algoritmo STT no gráfico (c).

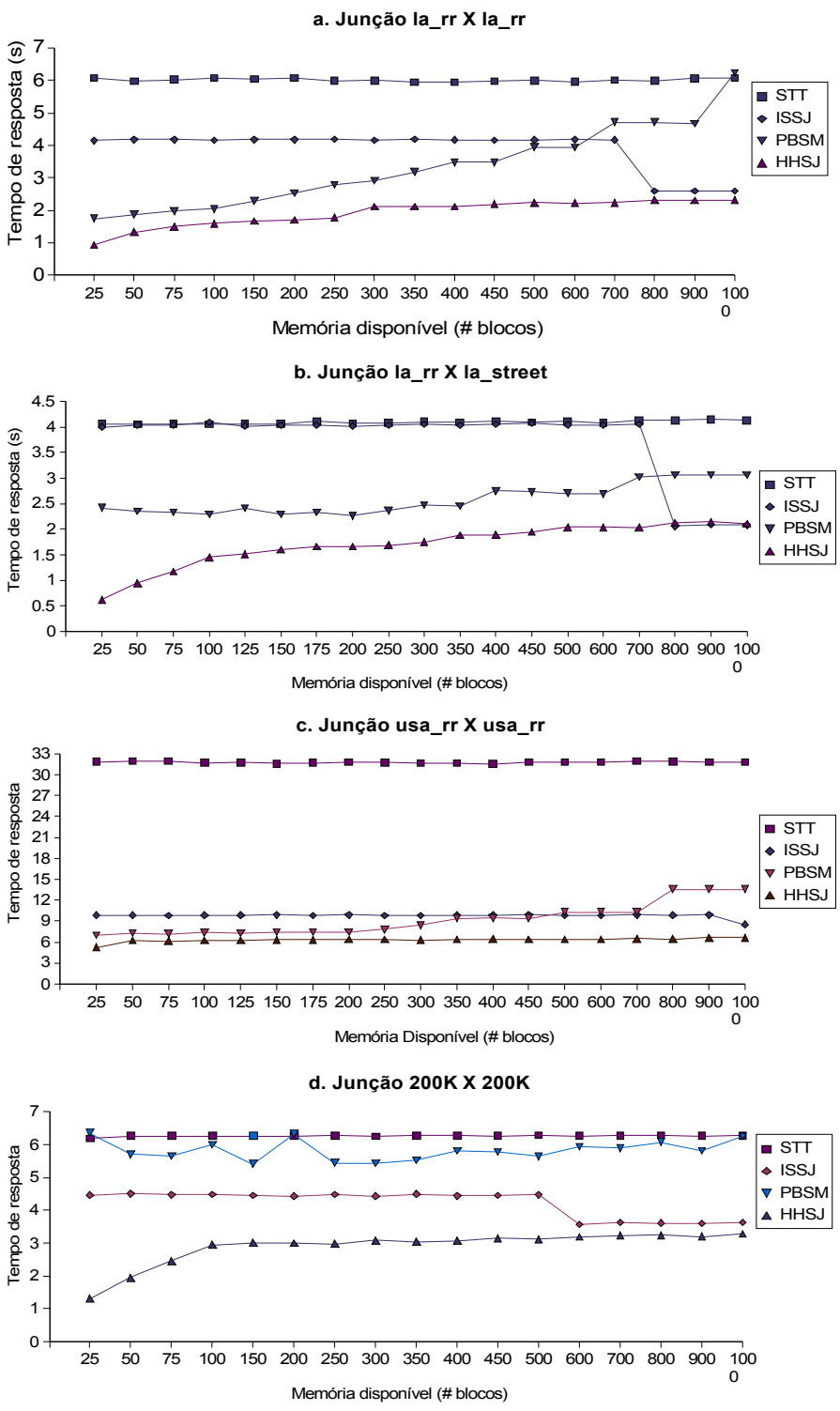



Figura 4.15: Gráficos do tempo de resposta dos quatro algoritmos, para diferentes junções espaciais de cardinalidade média.

A figura 4.15 mostra, também, o tempo de resposta para quatro junções espaciais, agora envolvendo conjuntos de cardinalidade média, entre 100.000 e 500.000 objetos na soma dos conjuntos. O algoritmo HHSJ é o mais adequado nos quatro casos. Nos gráficos (a), (b) e (d), apenas para tamanhos de memória maiores o algoritmo ISSJ se

aproxima. Na auto-junção do gráfico (c), a diferença é menos significativa em relação aos algoritmos PBSM e ISSJ.

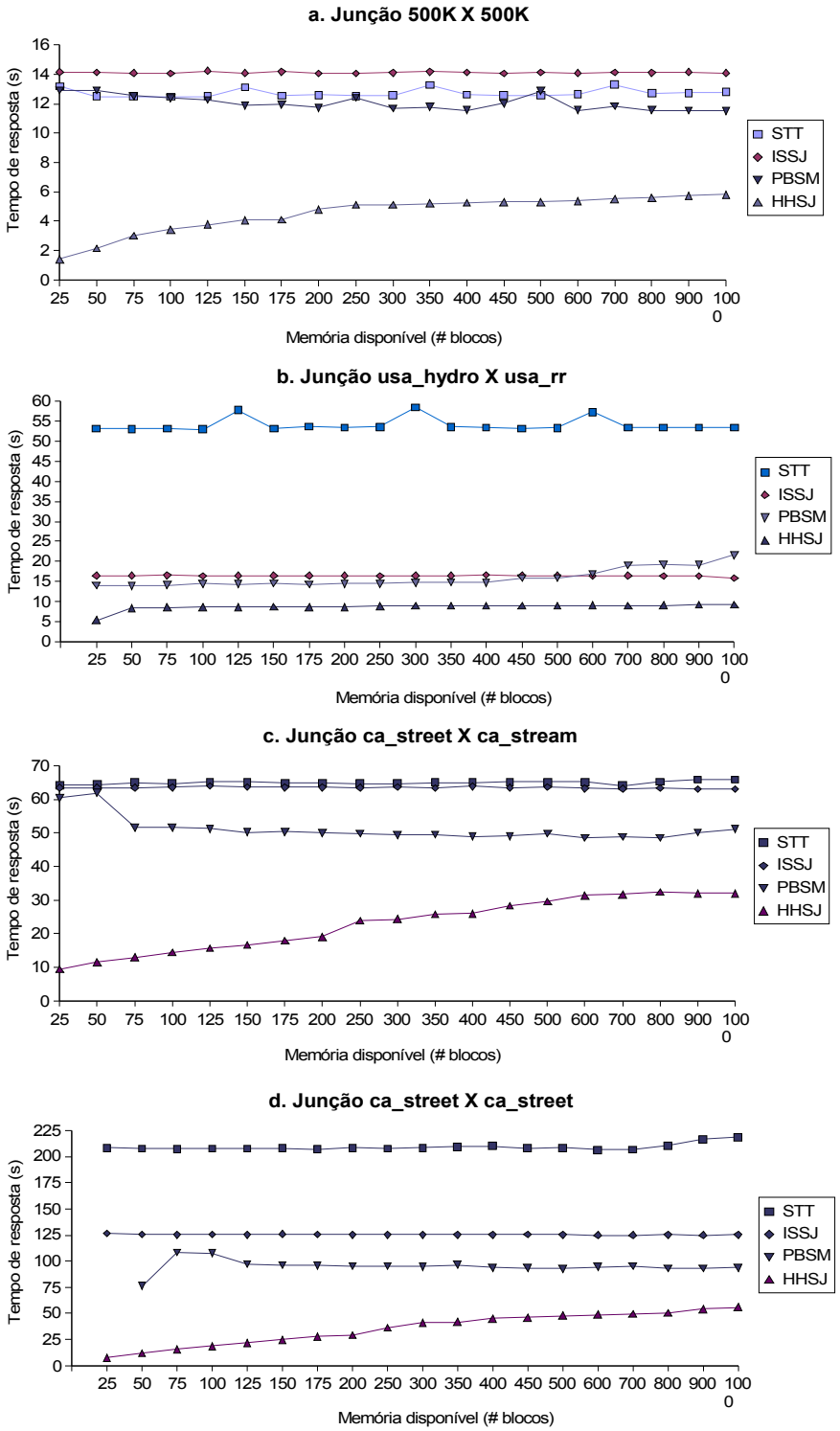



Figura 4.16: Gráficos do tempo de resposta dos quatro algoritmos, para diferentes junções espaciais de cardinalidade média.

Na figura 4.16, os conjuntos possuem maior número de objetos, com a junção espacial envolvendo acima de 500.000 objetos. Na junção entre conjuntos artificiais de 500.000 objetos distribuídos não-uniformemente, gráfico (a), o algoritmo HHSJ apresenta tempos de resposta bem menores que os demais. Nos demais, o HHSJ realiza

a operação em menos tempo quando a memória é pequena, mantendo-se competitivo nas demais situações. O STT, claramente, não é uma opção adequada.

O motivo principal do algoritmo HHSJ ser uma opção adequada em todas situações, é a realização de sucessivos *plane-sweep* com um número pequeno de objetos, permitindo executar a etapa mais crítica do algoritmo em menor tempo. Inclusive, compensando um número maior de operações de E/S. Além disso, a organização dos objetos em arquivos *hash* permite adaptar as partições de acordo com a necessidade, mas sem criar os arquivos de partição. A utilização de histogramas contribui para definir partições com um número de objetos menor que a capacidade da memória, evitando reparticionamentos custosos.

Na comparação com o algoritmo ISSJ, no caso dos dois arquivos já estarem ordenados, o HHSJ tem vantagem para conjuntos maiores. Com conjuntos pequenos, o ISSJ realiza a junção espacial em menor tempo. Para conjuntos médios, o HHSJ tem vantagem se a memória disponível for pequena.

O tempo de criação do arquivo *hash* e *HistQ* é maior que o tempo de criação de árvores-R pelo algoritmo STR, de construção otimizada. Porém, um algoritmo *Build&Match* baseado no HHSJ tem vantagem devido a diferença de desempenho entre o HHSJ e o STT, exceto no caso de conjuntos médios e memória grande, onde o desempenho dos algoritmos se aproxima.

A figura 4.17 mostra o tempo de resposta dos conjuntos alterando a cardinalidade. O HHSJ, além de ser mais rápido, apresenta a menor taxa de crescimento, sendo, portanto, menos afetado pelo aumento do número de objetos. O STT apresenta tempo de resposta variado devido à dificuldade para ajustar o *fanout* ideal para todas situações.

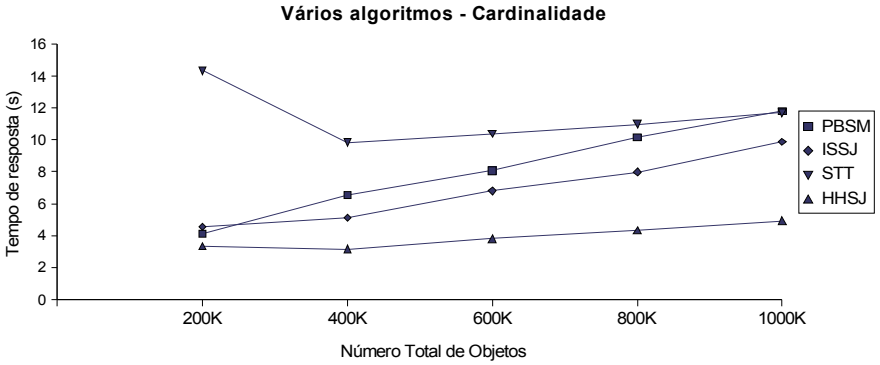



Figura 4.17 : Gráfico do tempo de resposta dos quatro algoritmos, variando a cardinalidade dos conjuntos.

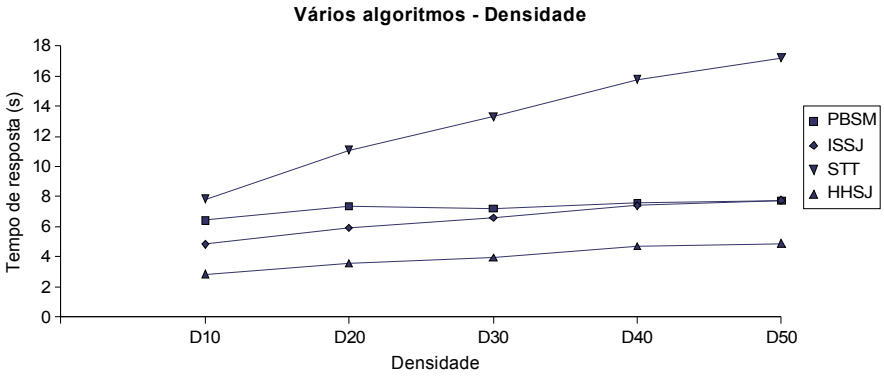



Figura 4.18 : Gráfico do tempo de resposta dos quatro algoritmos, variando a densidade dos conjuntos.

Na figura 4.18, pode-se observar o tempo de resposta de cada algoritmo quando a densidade dos conjuntos varia. O HHSJ, também neste caso, possui menor tempo de resposta, embora cresça de maneira mais acentuada que os algoritmos PBSM e ISSJ.

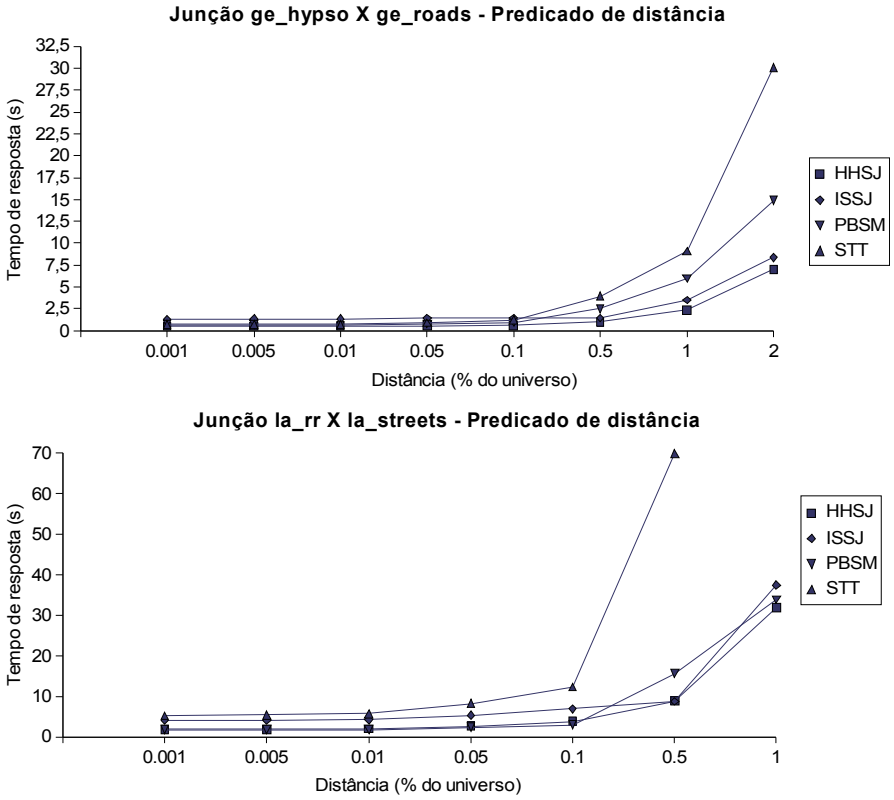



Figura 4.19 : Gráficos do tempo de resposta utilizando o predicado de distância entre objetos, para duas junções espaciais.

A figura 4.19 mostra os gráficos de duas junções quando o predicado espacial é a distância entre objetos. Todos algoritmos apresentam um crescimento exponencial, pois o número de pares que atendem ao predicado tem este comportamento.

## 4.6 Conclusão

Neste capítulo foi apresentado um novo algoritmo, desenvolvido com o objetivo de reduzir o tempo de resposta a partir da solução de alguns gargalos existentes nos demais algoritmos.

Na comparação com os demais algoritmos, o HHSJ é capaz de realizar operações de junção espacial em menor tempo, em quase todas situações, o que o torna uma alternativa interessante. A diferença de tempo é maior quando a capacidade de memória para ser utilizada como *buffer* é menor, o que torna o algoritmo especialmente interessante para ser utilizado em ambientes com restrição de memória, como dispositivos móveis e servidores compartilhados.

# 5 MECANISMO DE OTIMIZAÇÃO DE CONSULTAS

Considerando os gráficos mostrados nas figuras 4.14 a 4.16, pode-se perceber que não há um algoritmo que seja a melhor escolha em todas situações. Como SGBD, tradicionalmente, ao executarem uma consulta, utilizam um módulo de otimização de consultas para avaliar estratégias alternativas de execução e escolher uma estratégia ótima para executar cada consulta, pode-se sugerir que este módulo deva incluir um procedimento capaz de definir o melhor algoritmo para realizar a etapa de filtragem de uma junção espacial.

A figura 5.1 mostra as etapas típicas realizadas por um SGBD durante a execução de uma consulta escrita em uma linguagem de alto nível. A consulta é submetida pelo usuário e, inicialmente, analisada quanto a sua correção. O analisador gera, então, uma forma intermediária da consulta. O otimizador de consultas, com base em informações estatísticas mantidas no dicionário de dados, gera um plano de execução, que é realizado pelo motor (*engine)* do SGBD, que retorna o resultado da consulta ao usuário.

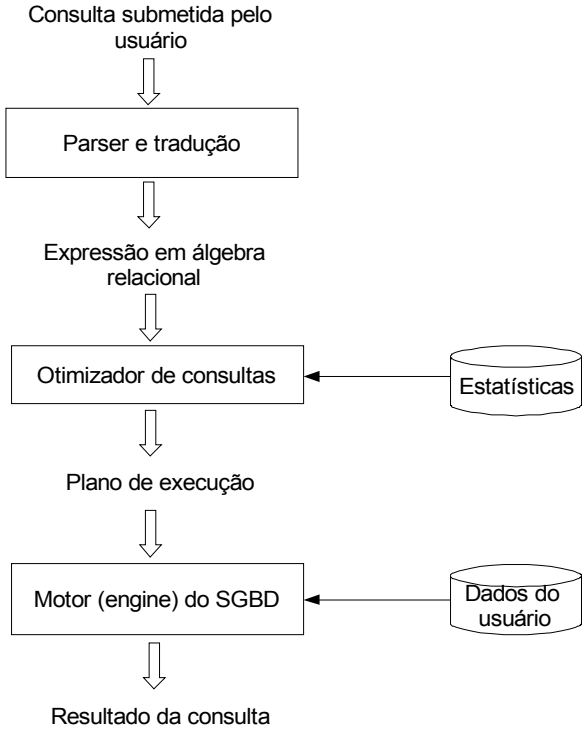



Figura 5.1 : Etapas na execução de um consulta em um SGBD

O objetivo deste capítulo é descrever um módulo de otimização de consultas capaz de escolher entre diferentes algoritmos, o mais rápido, para realizar a etapa de filtragem em uma operação de junção espacial. A partir das expressões de custo obtidas no capítulo 2, este otimizador baseia-se em um modelo de custos para realizar a escolha. O módulo pode ser um componente do otimizador de consultas do SGBD. Como dados de entrada, recebe o nome dos conjuntos envolvidos e o predicado de junção espacial, se aproveita das informações estatísticas disponíveis e indica o melhor algoritmo para realizar a operação. O otimizador de consultas está preparado para escolher entre quatro algoritmos: HHSJ, STT, ISSJ e PBSM.

No dicionário de dados, devem ser mantidas as seguintes informações:

- cardinalidade dos conjuntos;
- histograma de distribuição dos objetos no universo;
- número de blocos dos conjuntos;
- número de nodos e tamanho médio dos nodos de uma árvore-R, que indexe espacialmente os conjuntos;
- número de blocos do arquivo hash, incluindo os blocos de *overflow*; e
- características do equipamento, como os tempos médios de leitura/gravação de dados em disco, considerando acessos aleatórios e sequencias, tempo médio para comparação entre as coordenadas de dois objetos e entre dois números inteiros.

## 5.1 Modelo do equipamento

Para poder estimar o tempo de resposta de um algoritmo é necessário conhecer o tempo médio de operações de disco e CPU. As soluções atuais de armazenamento, mesmo as mais simples, possuem comportamento complexo, devido a características como:

- existência de *buffer* nas controladoras de disco;
- leitura avançada dos próximos setores do disco; em vários modelos, toda uma trilha é lida e mantida no *buffer* da controladora, na expectativa que os próximos acessos serão realizados a setores da mesma trilha;
- número variável de setores por trilha;
- se for um equipamento RAID, há possibilidades de paralelismo, dependendo da configuração selecionada;
- configurações dos diferentes sistemas de arquivos que podem ser utilizados em um mesmo sistema operacional, como Ext2, Ext3 e ReiserFS para Linux, que afetam o desempenho do disco.

Assim, torna-se muito difícil obter um modelo adequado de comportamento, que possa ser útil em um grande número de situações. A opção mais viável é executar uma seqüência de operações de disco e obter o tempo médio de realização em um determinado equipamento. Para os dados obtidos serem verazes, é necessário desligar o mecanismo de *buffer* do sistema operacional. São necessários, inicialmente, quatro diferentes valores:

- tempo de leitura em arquivos sequenciais ( $t_{rseq}$ )

- tempo de leitura em arquivos de acesso randômicos ( $t_{rrand}$ )
- tempo de gravação em arquivos sequenciais ( $t_{wseq}$ )
- tempo de gravação em arquivos de acesso randômicos( $t_{wrand}$ )

No entanto, o tamanho dos arquivos afeta o tempo de realização das operações. Para arquivos pequenos, a leitura e gravação de dados com acesso sequencial ou randômico é semelhante, devido ao *buffer* da controladora de disco, que não pode ser desligado. Assim, o modelo de custo é um pouco mais complexo e esta demonstrado no gráfico da figura 5.2. Para arquivos pequenos, o acesso randômico ou sequencial a um bloco tem o mesmo tempo médio. A medida que o arquivo cresce, a partir do ponto P1, há uma diferenciação, até atingir o ponto P2, de tamanho do arquivo, que o tempo médio de acesso randômico estabiliza no seu valor máximo.

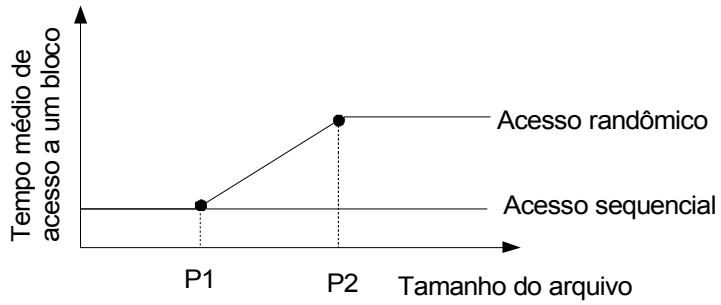



Figura 5.2 : Gráfico representando o modelo de custo para acesso sequencial e randômico a setores de um arquivo.

No equipamento utilizado, um programa foi executado que realizou milhões de acesso de cada tipo a arquivos de tamanhos diferentes, e os valores encontrados foram:

- Tempo de leitura em acesso sequencial = 0,6 ms
- Tempo de leitura em acesso randômico = 6,2 ms
- Tempo de gravação em acesso sequencial = 3,2 ms
- Tempo de gravação em acesso randômico = 24,9 ms
- P1 = 100 setores do disco
- P2 = 980 setores do disco

O tempo de gravação em arquivos de acesso randômicos é o valor máximo, para arquivos acima de P2. Para arquivos ente P1 e P2 utiliza-se uma aproximação linear do valor, e para arquivos com tamanho menor que P1, utiliza-se o tempo de acesso sequencial.

Quanto às operações em CPU, também há diferentes operações a serem consideradas, de acordo com o que é realizado pelos algoritmos. Assim, definiu-se os seguintes parâmetros:

- tempo para comparar duas coordenadas ( $t_{2\text{coord}}$ )
- tempo para decidir se dois objetos possuem ou não intersecção ( $t_{inter}$ )
- tempo para decidir se dois objetos possuem ou não intersecção, calcular o ponto de intersecção e verificar se ele esta dentro da partição em processamento ( $t_{rpm}$ )

Apesar de todas operações envolverem comparações de dois valores em ponto flutuante, o fato do mesmo valor ser utilizado em duas comparações diferentes, mas realizadas em seqüência, permite que o processador e o compilador otimizem o acesso a este valor, que não é lido, a cada vez, do seu endereço em memória. Para obter estes valores, também foi utilizado um programa específico, que executou milhões de vezes uma mesma operação e calculou o tempo médio de cada operação. No equipamento utilizado, encontrou-se os seguintes valores:

- tempo para comparar duas coordenadas ( $t_{2coord}$ ) = 2,1 x $10^{-9}$ s
- tempo para decidir se dois objetos se interseccionam ( $t_{inter}$ ) = 102,5 x $10^{-9}$ s
- tempo para decidir se dois objetos se interseccionam, calcular o ponto de intersecção e verificar se está dentro da partição em processamento ( $t_{rpm}$ ) = 130,3 x $10^{-9}$ s

## 5.2 Modelo de custos dos algoritmos

Nos capítulos anteriores, foram apresentadas as expressões de custo para os quatro algoritmos. Estas expressões devem ser completadas, com a inclusão dos tempos obtidos experimentalmente. Também deve-se considerar todas etapas dos algoritmos, não apenas a mais importante, porque as etapas secundárias podem representar 20 ou 30% do tempo de resposta. Como conseqüência, desprezar estas etapas pode resultar na escolha incorreta do algoritmo.

### 5.2.1 Número de pares de objetos comparados

A primeira etapa na estimativa do tempo de resposta dos algoritmos é calcular o número de pares de objetos que serão testados. Depois, este valor é aproveitado de diferentes maneiras, de acordo com o algoritmo.

Para tanto, é utilizado um algoritmo adaptado de Belussi et al. (2003, 2004), que baseia-se no histograma de distribuição de objetos de cada conjunto. O universo é dividido em um certo número de faixas, cada faixa com o tamanho médio dos objetos do conjunto. Para cada faixa, é mantido o número de objetos do conjunto que interseccionam a faixa, refletindo a não uniformidade na distribuição dos objetos. Em uma faixa, supõe-se que os objetos estejam regularmente distribuídos. Este histograma é mantido no dicionário de dados e é mais simples que a *HistQ* do algoritmo HHSJ.

O algoritmo carrega para memória o histograma do conjunto que apresenta o maior tamanho no eixo das coordenadas, seguindo a sugestão de Belussi et al. (2004). Os objetos do segundo conjunto são lidos, um por vez, sequencialmente. Para cada objeto, é definida a faixa mais próxima do seu início, onde há um certo número de objetos do primeiro conjunto já contabilizados, sendo este o número de objetos do outro conjunto com os quais será comparado. O número total de pares de objetos comparados é o somatório do número de comparações individuais.

A tabela 5.1 mostra o número de comparações previsto e o efetivamente realizado para algumas junções. Em alguns casos, o índice de acerto esta acima de 90%, para outros, o erro é maior. Porém, como todos algoritmos utilizam esta variável nas suas previsões, ainda é possível que o otimizador de consulta faça a escolha correta, pois o erro afeta a todas previsões.

Tabela 5.1 Número de pares de objetos comparados, valores estimados e medidos.

| *Conjunto 1* | *Conjunto 2* | *Medido* | *Estimado* | *Medido/Estimado* |
|---|---|---|---|---|
| *gr_roads* | *gr_rivers* | 3.900.041 | 3.955.942 | 1,01 |
| *ge_hypso* | *ge_roads* | 6.098.512 | 4.323.775 | 0,71 |
| *ge_hypso* | *ge_utility* | 4.970.096 | 5.338.095 | 1,07 |
| *la_rr* | *la_streets* | 21.492.516 | 24.658.010 | 1,15 |
| *usa_hydro* | *usa_rr* | 526.286.433 | 666.708.544 | 1,27 |

### 5.2.2 Algoritmo PBSM

A expressão (35), transcrita abaixo, indica o número de operações de E/S do algoritmo PBSM. O número de blocos de cada conjunto é uma informação mantida no dicionário de dados.

$$disco_{PBSM} = (2\,o\,r^2 + 2r + 1)(b^A + b^B)$$

O fator de replicação pode ser estimado, considerando o tamanho médio dos objetos e o número de células que dividem o espaço, que define o tamanho horizontal e vertical das células.

$$r = \left(\frac{s_y^{Cel} - s_y}{s_y^{Cel}}\right)\left(\frac{s_x}{s_x^{Cel}}\right) + \left(\frac{s_x^{Cel} - s_x}{s_x^{Cel}}\right)\left(\frac{s_y}{s_y^{Cel}}\right) + 3\left(\frac{s_x}{s_x^{Cel}}\right)\left(\frac{s_y}{s_y^{Cel}}\right) + 1$$

Para concluir a previsão, é necessário estimar a porcentagem de partições que sofrem *overflow*, assim como o número de objetos envolvidos. Após calcular o número de partições, sabe-se o número de faixas horizontais e verticais que definem as células e o mapeamento de células para partições. Dentro de uma faixa, considera-se uma distribuição regular, permitindo estimar o número de objetos em cada célula, o número de partições que podem sofrer *overflow* e o número de objetos nestas partições.

Como o acesso aos arquivos, tanto na leitura quanto gravação é, essencialmente, sequencial, para obter o tempo previsto basta multiplicar o número de acessos pelo tempo médio de um acesso a disco, separando gravações de leituras.

$t_{PBSM}^{disco} = (o\,r^2 + r + 1)(b)\,t_{rseq} + (o\,r^2 + r)(b)\,t_{wseq}$, sendo $b = b^A + b^B$.

O tempo de operações de CPU baseia-se na expressão (36), acrescendo o tempo para realizar o particionamento e reparticionamento dos objetos e a ordenação dos objetos pertencentes a uma partição, quando carregados em memória. Estas operações, além da manipulação de objetos nas listas de ativos do *plane-sweep*, são realizadas pela comparação de valores de coordenadas.

O número de pares de objetos comparados é influenciado pela divisão do espaço e pelo número de partições, sendo menor que o número total, inicialmente calculado. Para cada par de objetos é necessário verificar se possuem intersecção e, em caso positivo, aplicar o *Reference Point Method*. Assim, o tempo de CPU do algoritmo pode ser previsto pela expressão

$$t_{PBSM}^{CPU} = 3.5\, t_{Coord}(r+o)(n^A+n^B)+ + t_{RPM}\, c_{PBSM}+ + 2\, t_{Coord}\, r(n^A+n^B)\log\frac{r(n^A+n^B)}{P}$$

A primeira parcela corresponde ao tempo para particionar e reparticionar objectos. A constante 3.5 ajusta o desempenho desta etapa do algoritmo. A segunda parcela corresponde a comparação de pares de objetos. A terceira parcela é relativa à ordenação de objetos e a manipulação de objetos durante a realização do *plane-sweep*.

O tempo total do PBSM pode ser calculado por

$$t_{PBSM} = t_{PBSM}^{IO} + 3\, t_{PBSM}^{CPU}$$

A inclusão de constantes, nas fórmulas, adequa a previsão de tempo à realidade. Ao obter as expressões de custo apresentadas nos capítulos 2 e 4, algumas simplificações são necessárias e apenas as instruções mais relevantes são consideradas. De fato, outras instruções auxiliares são executadas e, de alguma forma, consomem recursos do sistema. Uma alternativa possível seria investigar o código *assembler* gerado pelo compilador e analisar o número de ciclos de CPU de cada operação e trecho de código, mas esta solução é por demais dependente do hardware e compilador utilizados. Assim, as constantes foram obtidas através de um processo de ajuste à realidade.

### 5.2.3 Algoritmo STT

Para estimar o tempo de resposta do algoritmo STT, o otimizador necessita de informações sobre as árvores-R envolvidas, como número e tamanho médio dos nodos, em cada nível da árvore-R.

Todas operações de E/S realizadas são leituras randômicas de nodos, devido à forma como os nodos são gravados em disco, e ao padrão de acesso aos nodos, sempre do nodo raiz para as folhas. Assim, o tempo para operações de E/S é

$$t_{STT}^{Disco} = IO_{STT} * t_{rw}$$

O valor de $IO_{STT}$ é obtido a partir da expressão (4), incluindo a probabilidade de um certo nodo estar no *buffer* em memória, evitando a operação de leitura do nodo. Huang et al (1998) sugere uma distribuição exponencial. Uma alternativa mais simples para calcular o número de acessos a disco é dada abaixo. O símbolo $\alpha$ representa uma constante, experimentalmente obtida, como sendo 5,3. $M_b$ é o tamanho do *buffer* de memória, medido em número de blocos.

$$IO_{STT} = n_R^A + n_R^B + \frac{\alpha(n_R^A+n_R^B)}{M_b} \quad .$$

O tempo de CPU pode ser obtido a partir da expressão (9). A primeira parcela, o número de pares de objetos comparados, é multiplicado pelo tempo para verificar se dois objetos possuem intersecção ($t_{inter}$). A segunda parcela representa a manipulação das listas de objetos ativos, envolvendo a comparação de duas coordenadas ($t_{Coord}$). O tempo de processamento em CPU, é

$$t_{STT}^{CPU} = t_{inter}\, c_{STT} + t_{2\mathrm{Coord}}\, Comp \times (2\, Fanout \log 2\, Fanout) \,. \qquad (27)$$

Somando as expressões e ajustando o desempenho através de constantes, o tempo estimado para uma junção espacial pelo algoritmo STT é

$$t_{STT} = 5\, t_{STT}^{IO} + 2.85\, t_{STT}^{CPU} \tag{28}$$

### 5.2.4 Iterative Stripped Spatial Join

O número de operações de E/S do algoritmo ISSJ, expresso por (10) e (11), deve ser diferenciado entre leituras e escritas. A ordenação precisa carregar os objetos para memória (leitura) e escrever em disco o resultado, para que o *plane-sweep* percorra (leitura) sequencialmente os arquivos.

Se for utilizada ordenação interna, em memória, a leitura e escrita é sequencial. Se ocorrer ordenação externa, são lidos e gravados subconjuntos dos dados de maneira sequencial, ocorrendo uma quebra de sequência (leitura/gravação randômica) apenas ao trocar de uma parte do arquivo para outra. Como o número de acessos não sequenciais é muito pequeno sobre o todo, pode-se simplificar e considerar todos sequenciais, obtendo-se a expressão

$$t_{ISSJ}^{Disco} = t_{sr}\, 2(b^A + b^B) + t_{sw}(b^A + b^B) \tag{29}$$

ou, se ambos arquivos forem ordenados externamente,

$$t_{ISSJ}^{Disco} = t_{sr}(b^A + b^B) + (b^A \lceil \log_M b^A \rceil)(t_{sr} + t_{sw}) + (b^B \lceil \log_M b^B \rceil)(t_{sr} + t_{sw}) \ . \tag{30}$$

O tempo de processamento em CPU pode ser estimado a partir de (13). As comparações entre objetos incluem o *Reference Point Method*, sendo, portanto, necessário utilizar $t_{RPM}$. A fórmula de custo completa é

$$t_{ISSJ}^{CPU} = t_{RPM}\, c_{ISSJ} + t_{Coord}\, r\,(n^A + n^B) \log r \frac{(n^A + n^B)}{\Phi} \ . \tag{31}$$

Finalmente, para obter o tempo de resposta do algoritmo é necessário somar os tempos parciais e incluir as constantes adequadas.

$$t_{STT} = \alpha\, t_{STT}^{IO} + 10.5\, t_{STT}^{CPU} \ . \tag{32}$$

O símbolo $\alpha$ representa um valor dependente do algoritmo de ordenação, externo ou interno. Se ambos conjuntos forem ordenados externamente, $\alpha = 12$. Se os dois forem ordenados internamente, $\alpha = 9{,}6$. E, se apenas um deles, digamos o conjunto $A$, for ordenado internamente, temos uma expressão de cálculo que representa a proporção do menor conjunto sobre o total de objetos:

$$\alpha = \frac{n^A}{n^A + n^B}(12 - 9{,}6) + 9{,}6$$

A utilização de uma variável se justifica porque, no caso de ordenação externa, há um acréscimo de complexidade do algoritmo, porque é necessário controlar a passagem de objetos em memória para o disco e do disco para memória.

### 5.2.5 Histogram-Hash based Spatial Join

O algoritmo HHSJ realiza a subdivisão do espaço através de um histograma, chamado *HistQ*, que deve ser carregado para memória. Esta carga (leitura) é realizada através de acessos sequenciais. Após, os *buckets* dos arquivos de *hash* devem ser lidos para memória, de maneira não sequencial e alternando entre os dois conjuntos. Portanto, o tempo para realizar operações de E/S pode ser calculado com base em (22), mas diferenciando acessos sequenciais e randômicos, o que resulta em

$$t_{HHSJ}^{Disco} = t_{sr}(b_{HistQ}^{A} + b_{HistQ}^{B}) + t_{rr}(4^{h^{A}} + o_{Hash}^{A} + 4^{h^{B}} + o_{Hash}^{B}) . \quad (36)$$

O tempo de CPU é influenciado, indiretamente, pelo tamanho médio das partições. A subdivisão do espaço, no algoritmo HHSJ, é realizada considerando as *HistQ* dos dois conjuntos, em um algoritmo de certa complexidade. Como resultado da subdivisão do espaço, tem-se o número correto de objetos em cada partição. Porém, incorporar este algoritmo ao otimizador de consultas poderia impactar negativamente o tempo de execução do otimizador. Assim, optou-se por reaproveitar o número de partições estimado para o PBSM e considerar que a subdivisão do espaço se daria de maneira uniforme, ou seja, que todas partições teriam o mesmo tamanho. Em um dos eixos, este tamanho é

$$s_x^{Part} = \frac{s_x^{Univ}}{\sqrt{P}} . \quad (37)$$

A raiz quadrada em (37) deve-se forma como a divisão do espaço é realizada utilizando-se uma *quadtree*, que divide o espaço, em um dos eixos, sempre pela metade. O tamanho médio das partições é necessária para calcular o número de pares de objetos comparados, como definido em (23).

O tempo de CPU pode ser estimado por

$$t_{HHSJ}^{CPU} = t_{RPM}\, c_{HHSJ} + t_{Coord}\, r(n^{A} + n^{B}) \log r \frac{(n^{A} + n^{B})}{P\Phi} . \quad (38)$$

Combinando (37) e (38), temos que o tempo de resposta do algoritmo pode ser estimado por

$$t_{HHSJ} = 14.8\, t_{HHSJ}^{IO} + 17.5\, t_{HHSJ}^{CPU} . \quad (39)$$

## 5.3 Resultados obtidos

Para verificar a utilidade do software desenvolvido, duas métricas são importantes: eficácia, definida como o número de acertos sobre o total de previsões; e eficiência, o tempo de execução do otimizador de consultas sobre o tempo economizado pela escolha de um algoritmo mais rápido.

Antes de mostrar resultados numéricos de tempo estimado, é importante ressaltar que o objetivo do otimizador é escolher o melhor algoritmo para realizar a operação de junção espacial, fase de filtragem. Assim, a precisão do tempo previsto, embora desejável, não é o principal objetivo a ser alcançado.

Ao relatar os resultados obtidos, a mesma divisão entre diferentes cenários será utilizada. A figura 5.3 mostra os resultados da junção espacial em três casos diferentes, envolvendo conjuntos de cardinalidade pequena. A esquerda estão os valores estimados, à direita os valores medidos.

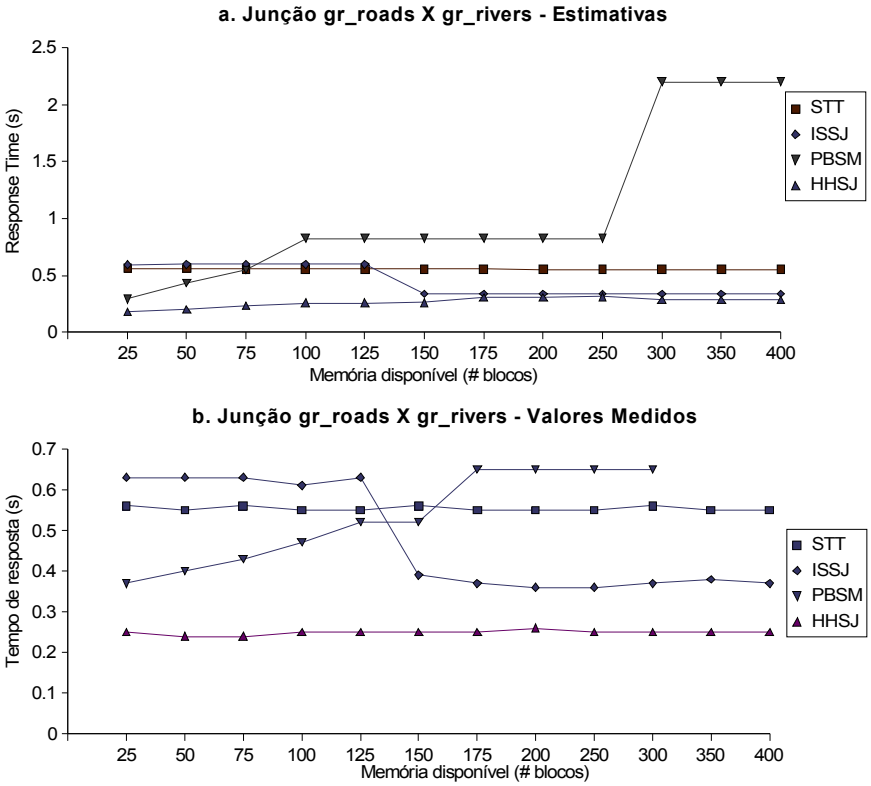



Figura 5.3 : Gráficos de tempo estimado e medido para junção espacial entre os conjuntos *gr_roads* e *gr_rivers*.

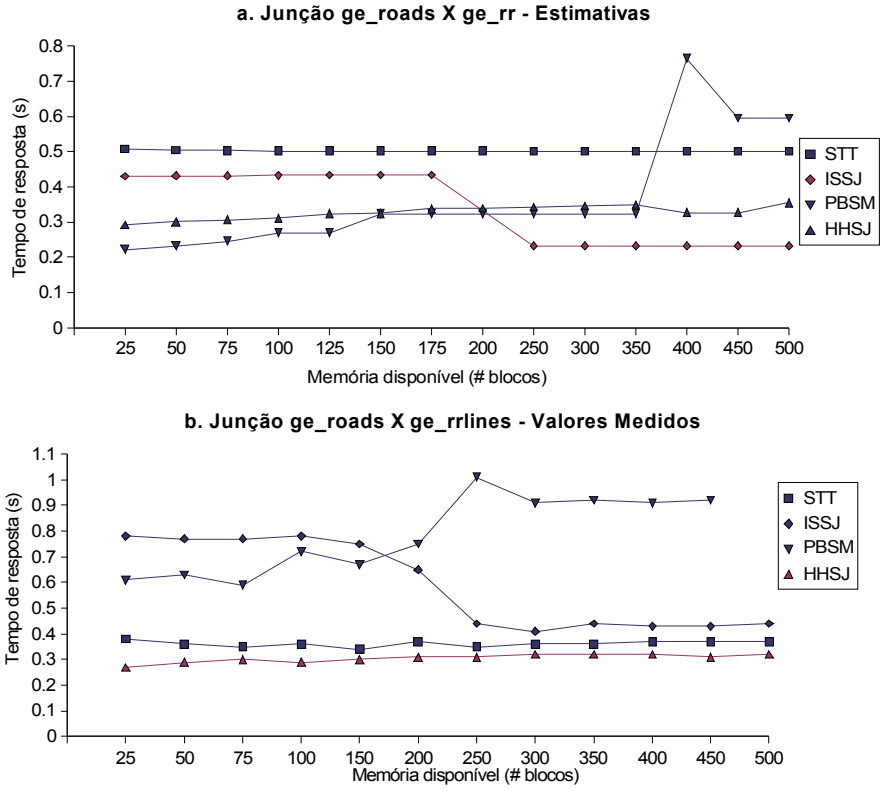



Figura 5.4: Gráficos de tempo estimado e medido para junção espacial entre os conjuntos *ge_roads* e *ge_rrlines*.

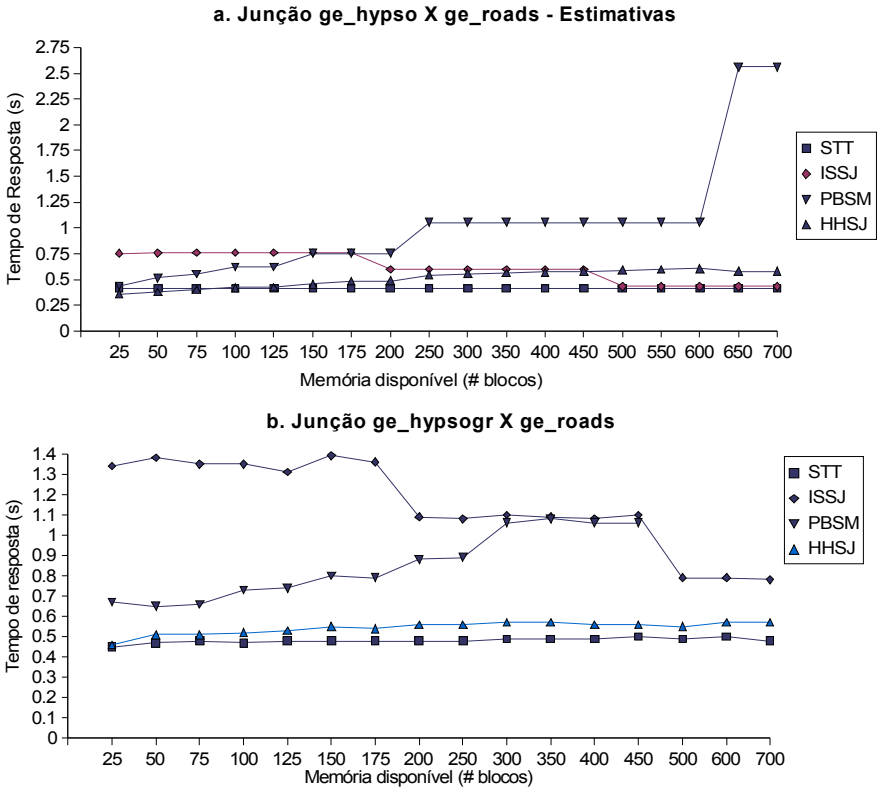



Figura 5.5: Gráficos de tempo estimado e medido para junção espacial entre os conjuntos *ge_roads* e *ge_hypsogr*.

Os gráficos das figuras 5.3 a 5.5 permitem comparar o tempo estimado e o tempo de resposta medido para três junções espaciais diferentes, envolvendo conjuntos pequenos, até 100.000 objetos. Na junção entre *gr_roads* e *gr_rivers*, o algoritmo HHSJ é escolhido como o mais rápido em todos casos, correspondendo aos valores medidos. Na junção *ge_roads* X *ge_rrlines*, a estimativa indica o algoritmo ISSJ como o melhor para memória de tamanho médio, quando o HHSJ é mais rápido. Na junção *ge_hypso* X *ge_roads*, os algoritmos STT e HHSJ apresentam os menores tempo, com vantagem para o STT. A estimativa reflete esta realidade, embora estime um valor menor que a realidade para o ISSJ em memórias de tamanho médio.

Os gráficos das figuras 5.6 a 5.8 comparam o tempo estimado e o tempo de resposta medido para junções espaciais envolvendo conjuntos de cardinalidade média. Nos três casos, o algoritmo HHSJ apresenta o menor tempo de resposta, corretamente previsto, embora com o aumento da memória disponível, que reduz o tempo do ISSJ, os dois se aproximem, com uma tendência de inverter a relação, ou seja, o ISSJ se tornar mais rápido. As estimativas de tempo mostram claramente este comportamento, indicando o HHSJ e reduzindo o tempo do ISSJ, a ponto de ambos se aproximarem. Tornando a escolha entre ambos quase indistinta.

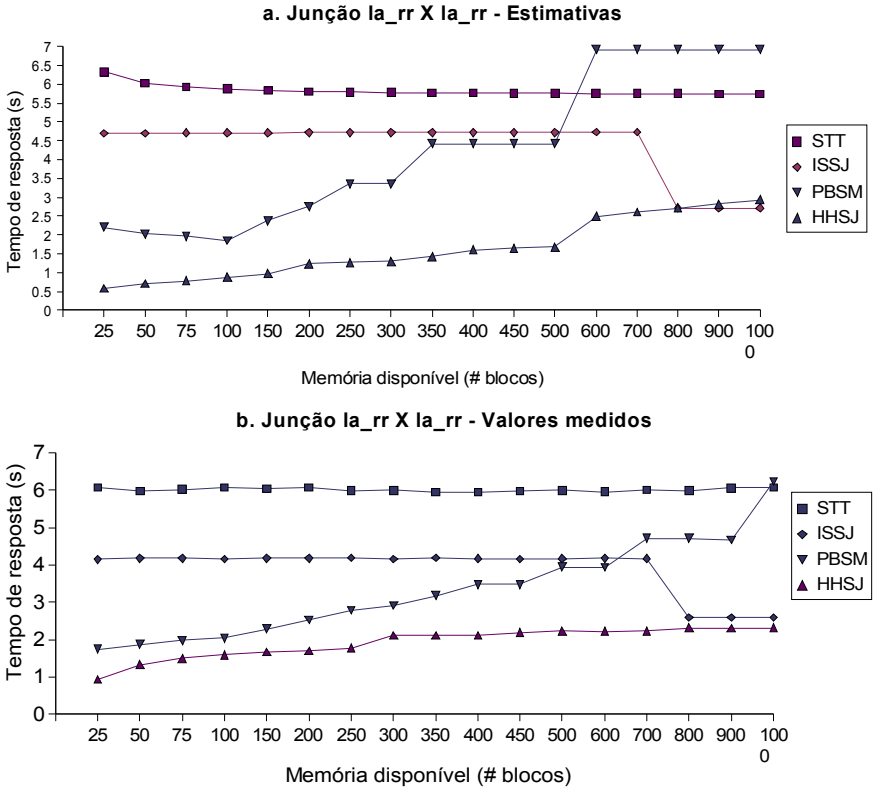



Figura 5.6: Gráficos de tempo estimado e medido para junção espacial entre os conjuntos *la_rr* e *la_rr*.

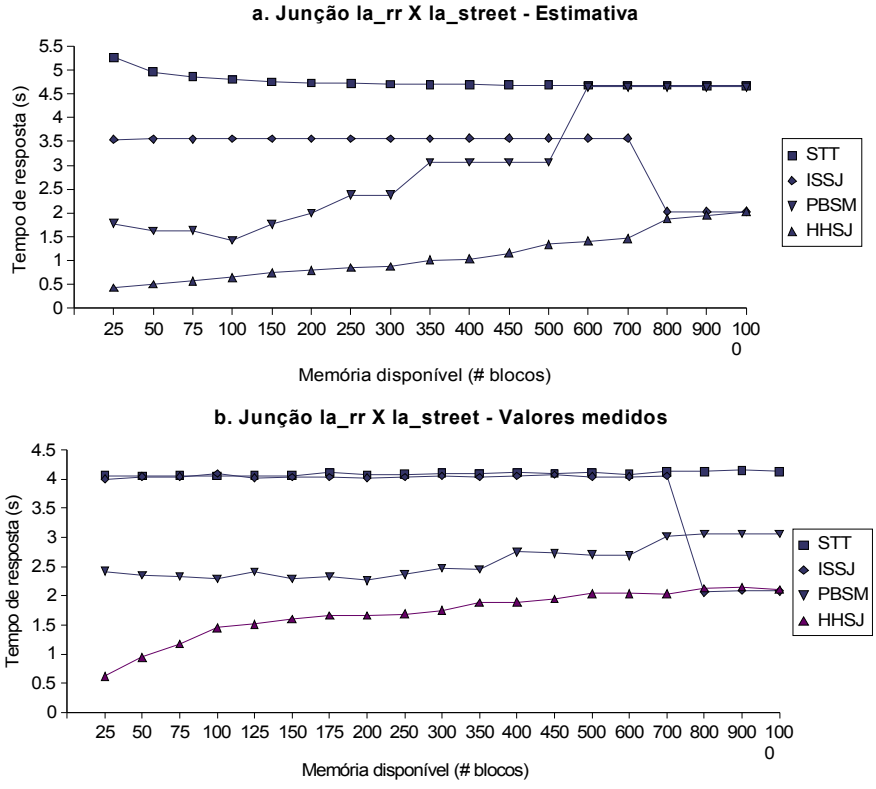



Figura 5.7: Gráficos de tempo estimado e medido para junção espacial entre os conjuntos *la_rr* e *la_street*.

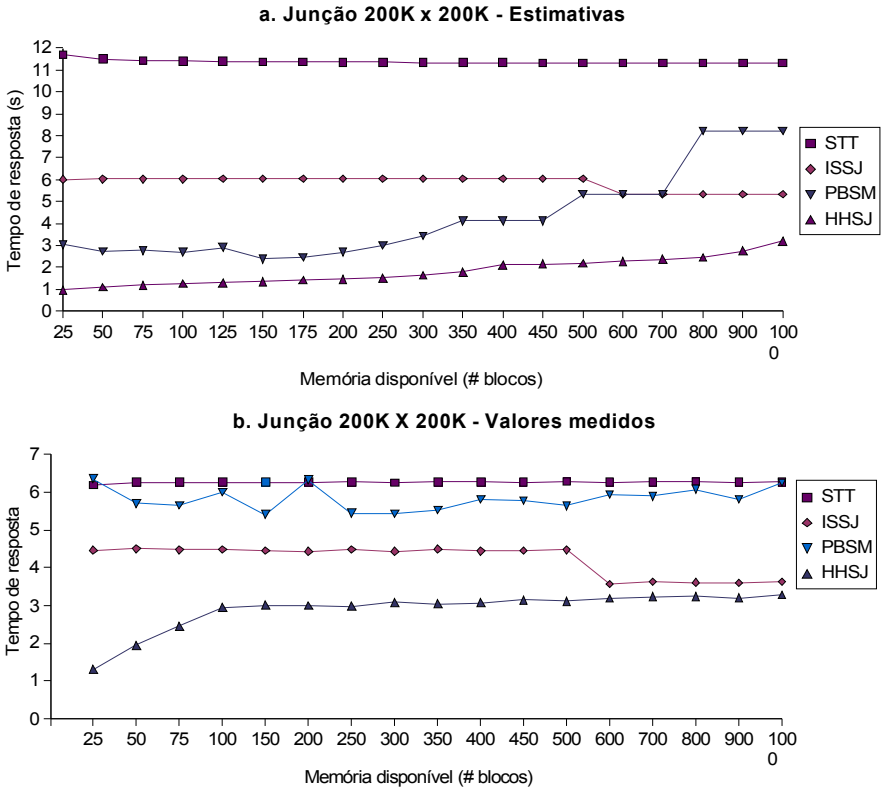



Figura 5.8: Gráficos de tempo estimado e medido para junção espacial entre os conjuntos sintéticos de 200K objetos.

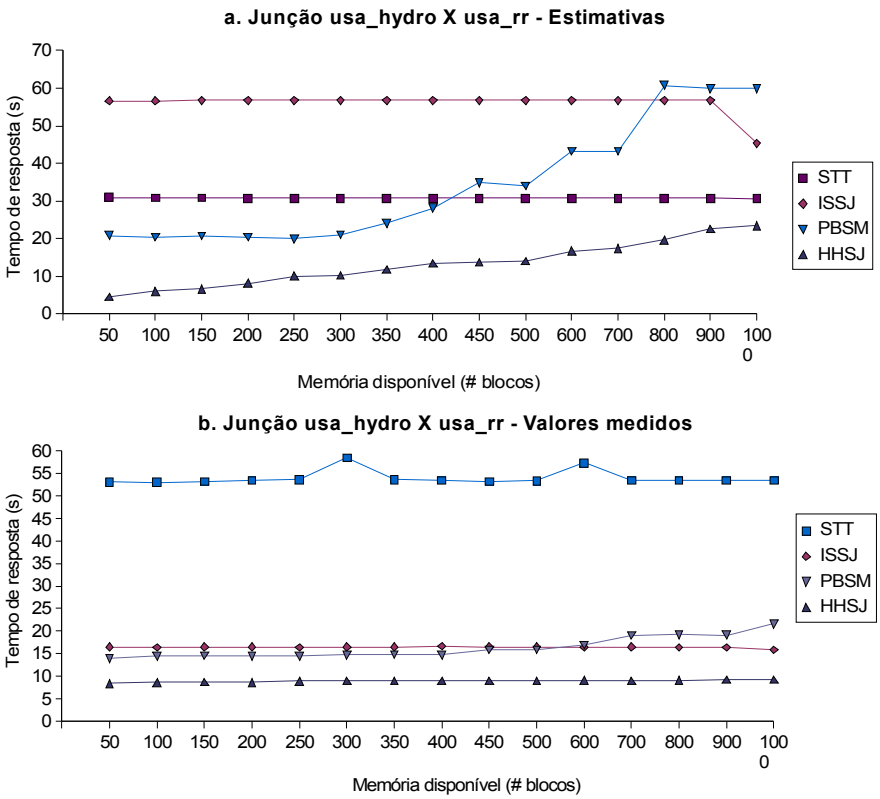



Figura 5.9: Gráficos de tempo estimado e medido para junção espacial entre os conjuntos *usa_hydro* e *usa_rr*.

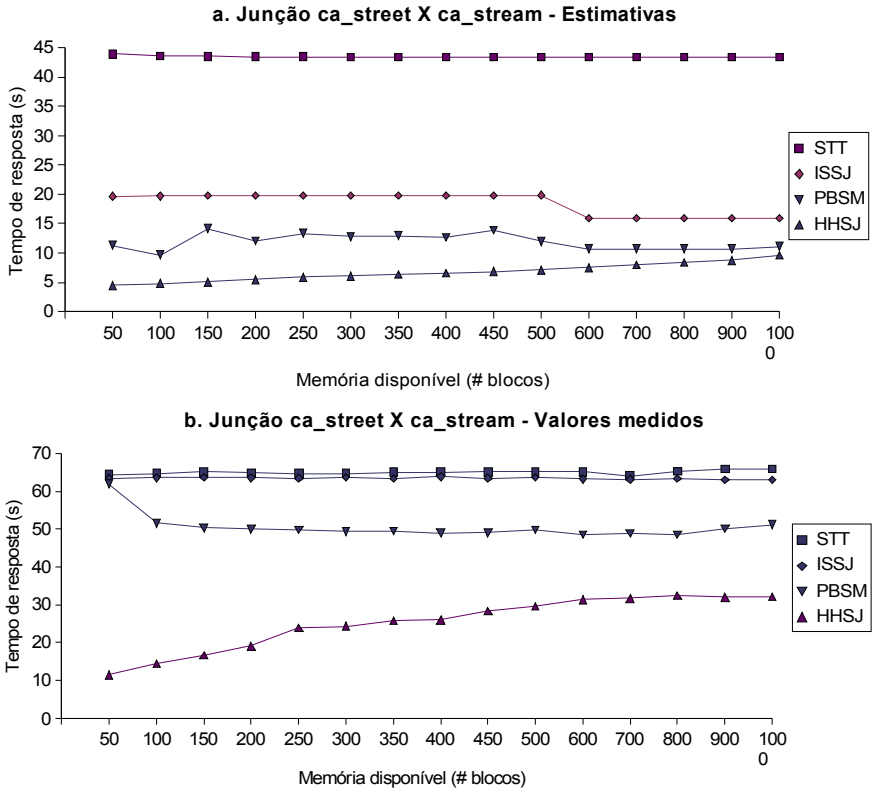



Figura 5.10: Gráficos de tempo estimado e medido para junção espacial entre os conjuntos *usa_hydro* e *usa_rr*.

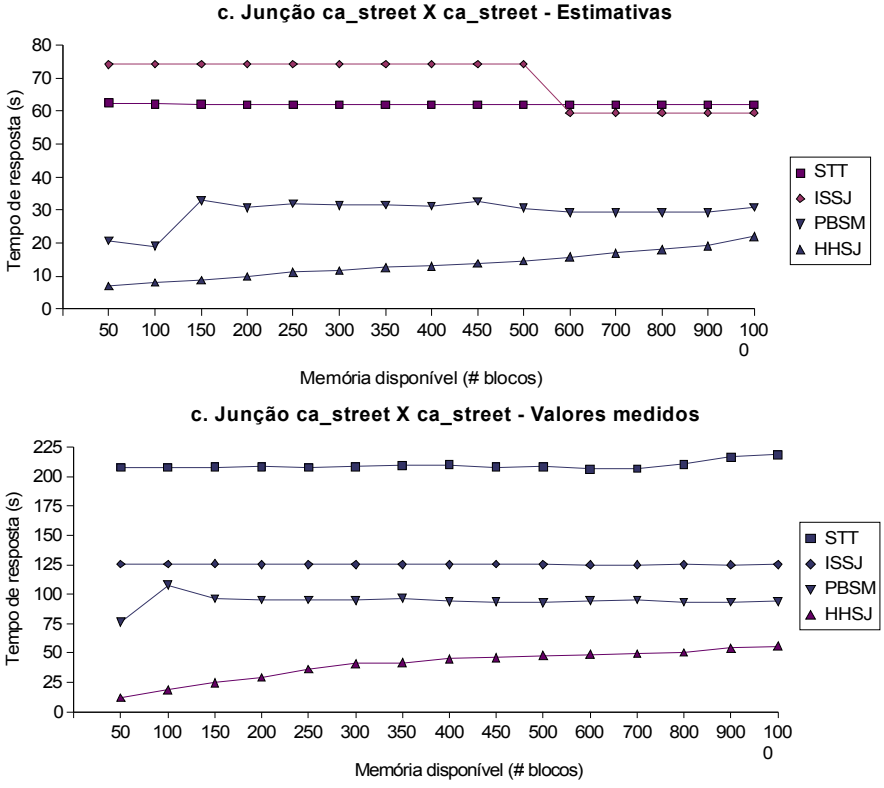



Figura 5.11: Gráficos de tempo estimado e medido para junção espacial entre os conjuntos *usa_hydro* e *usa_rr*.

Na figura 5.9 a 5.11 podem ser vistos os gráficos relativos a estimativas e tempo de resposta medido para junções envolvendo conjuntos grande número de objetos. Nos testes realizados, o algoritmo HHSJ apresentou melhor desempenho, o que é refletido corretamente nas estimativas. O tempo de resposta estimado, em geral, é menor que o tempo medido, mas não altera a escolha do melhor algoritmo.

No ajuste de valores constantes para estimar o tempo de resposta dos algoritmos, foi dado preferência para a correção em junções de tamanho grande e médio, em detrimento de junções com poucos objetos. Isto se justifica porque as diferenças no tempo de resposta para conjuntos grandes são de 10 ou mais segundos, enquanto em conjuntos pequenos, as diferenças são de 0,10, 0,20 segundos, ou seja, 100 vezes menor.

Utilizando conjuntos de dados reais e sintéticos, foram realizados 207 operações de junção espacial, alterando os conjuntos envolvidos e o tamanho de memória disponível. Alterando os conjuntos, as características espaciais, como formato, área de cobertura densidade e cardinalidade, são modificadas. Alguns conjuntos, como *usa_counties* cobrem toda área continental dos EUA, havendo áreas vazias apenas nas regiões marítimas. Outros, como *ge_utility* cobrem parcialmente a área de abrangência, havendo muitos espaços vazios entre os objetos. A conseqüência é que o número de pares de objetos comparados pode variar bastante, mesmo para dois conjuntos com cardinalidades semelhante.

O otimizador de consultas escolheu o algoritmo correto em 184 operações, errando em apenas 23 operações. Portanto, a taxa de acerto é de 88,9%. Os erros ocorreram, predominantemente, nas junções envolvendo conjuntos pequenos, em 17 operações. Este fato é intencional, pois o ajuste foi realizado visando conjuntos grandes e médios.

### 5.3.1 Eficiência

O segundo parâmetro para avaliar o otimizador de consultas é a eficiência, definida como tempo para executar o processo de otimização sobre o tempo de processamento reduzido através da escolha correta do algoritmo.

O otimizador, em uma primeira etapa, calcula o número de pares de objetos comparados, a variável *c*. Como explicado, ele carrega para memória o histograma do conjunto com objetos de maior tamanho médio e, após, percorre sequencialmente todos os objetos do segundo conjunto. Esta é a operação mais complexa, por envolver um certo número de operações de leituras em disco. Ao final, as demais fórmulas podem ser calculadas, consumindo apenas operações de CPU.

Para medir o tempo de execução de todos programas, foram utilizados alguns recursos da linguagem C, que fornece o número de ciclos de CPU entre duas chamadas da função *clock( )*. Mas, para evitar uma sobrecarga do sistema, esta função tem uma precisão de 1 milissegundo. Assim, não é possível obter valores menores que este limite mínimo. Para conjuntos pequenos, o tempo de execução do otimizador de consultas, foi sempre abaixo do milissegundo. Para conjuntos grandes, como a junção entre *usa_hydro X usa_rr*, o tempo de execução foi de 2 milissegundos, e para a junção *ca_street X ca_stream,* foi 7 milissegundos. Nos dois casos, este valor é uma fração muito pequena sobre a economia de escolher o algoritmo correto, justificando sua utilização.

## 5.4 Ajuste de desempenho

Com base nas expressões de custo, é possível também ajustar o desempenho dos algoritmos, sempre que for possível variar o tamanho da memória alocada e outros parâmetros de configuração. Estas regras são interessantes para SGBD Geográficos que possuam apenas um ou dois destes algoritmos implementados, sendo mais importante otimizar o desempenho destes algoritmos que escolher entre um deles.

Para tanto, um conjunto de estratégias de ajuste está definido. Estas estratégias podem ser utilizadas de maneira isolada, de acordo com a existência, ou não, de um certo algoritmo.

**Estratégia 1:** para reduzir o tempo do *plane-sweep*, o SGBDG pode ordenar os objetos por $x_{min}$ ou $y_{min}$, escolhendo o eixo com menor valor estimado para $c$.

Como mostrado anteriormente, o formato dos objetos altera o número de réplicas e de pares de objetos comparados pelo *plane-sweep*. Se o SGBDG manter em seu dicionário de dados dois histogramas por faixa, um para o eixo $x$, outro para o eixo $y$, poderá estimar $c$ para cada eixo e escolher a alternativa com menor valor estimado.

Para validar esta estratégia, realizamos junções espaciais ordenando pelos dois eixos. Sempre que a diferença entre os valores estimados para $c$ era maior que 5%, o eixo com menor tempo d eresposta foi escolhido, independente do algoritmo de junção espacial.

**Estratégia 2**: para otimizar o algoritmo STT, o número de entradas por nodo, o *fanout*, deve ser ajustado.

Esta estratégia deve ser incorporada na criação da árvore-R. Nossa heurística é escolher um *fanout* de 73 entradas, nodos com 2Kb, para conjuntos pequenos e médios, e um *fanout* de 146 entradas, nodos de 4Kb, para conjuntos grandes.

Como é possível o usuário solicitar uma junção entre um conjunto pequeno e outro grande, o algoritmo STT deve ser adaptado para lidar com a possibilidade de combinar árvores-R com *fanout* diferentes..

**Estratégia 3**: como o tempo de resposta do algoritmo STT é pouco dependente da memória disponível, é possível alocar pouca memória para ele, viabilizando que outras operações, mais dependentes de memória, aloquem um espaço maior.

Pelos testes realizados, são necessários apenas 4 vezes a altura da maior árvore-R para obter um tempo de resposta estável, atingindo um desempenho quase ótimo do algoritmo.

**Estratégia 4**: para otimizar o algoritmo ISSJ, o número de faixas é importante e possui um valor ótimo.

Um número alto de faixas pode gerar grande número de réplicas, prejudicando o algoritmo. Uma pequena quantidade de faixas não divide o problema em um número suficiente para haver ganhos.

Com base nos testes realizados, optamos por um valor *default* de 16 faixas.

**Estratégia 5**: o algoritmo ISSJ necessita de memória para classificar os objetos de cada conjunto em memória.

Esta estratégia é evidente quando observamos os gráficos de tempo de resposta do algoritmo.

**Estratégia 6**: aumentar o número de partições do algoritmo PBSM

O PBSM tem melhor desempenho quando a memória disponível é menor porque ele define um número maior de partições. Porém, se a grade regular utilizada para subdividir o espaço tiver mais linhas e colunas, o fator de replicação é aumentado. Assim, o desempenho ótimo mantém a grade regular e define um grande número de partições, permitindo alocar menos memória.

A grade regular *default* utilizada era de 32 x 32, como sugerido pelos autores (PATEL e DEWITT, 1996).

**Estratégia 7**: para o algoritmo HHSJ, aumentar o número de partições.

O raciocínio é o mesmo da estratégia anterior. Aumentar o número de partições permite alocar menos memória e reduzir o tempo de resposta. Uma diferença importante entre o PBSM e o HHSJ é o fator de replicação. No HSSJ, é definido na criação dos arquivos *hash* e não varia em tempo de execução.

**Estratégia 8**: para o algoritmo HHSJ, aumentar o número de faixas em memória.

Esta estratégia é similar à 4, para o algoritmo ISSJ. O valor *default* utilizado foi de 8 faixas, um pouco menor que o ISSJ porque o espaço já é dividido em partições. Assim, evita-se um número de réplicas excessivo.

## 5.5 Conclusão

Neste capítulo, foi apresentado um otimizador de consultas capaz de escolher o melhor algoritmo para realizar cada junção espacial, a partir de informações estatísticas mantidas no dicionário de dados. Este otimizador pode ser incorporado como um sub-módulo especializado de um SGBDG, que já possui o seu otimizador para as demais consultas.

Ainda, um conjunto de estratégias heurísticas que permitem ajustar o desempenho de algoritmos individuais foi definido, podendo ser utilizado em SGBDG que implementam apenas um ou dois dos algoritmos estudados.

# 6 CONCLUSÃO E TRABALHOS FUTUROS

Ao longo deste texto foi descrito o trabalho de pesquisa realizado na área de algoritmos para a fase de filtragem da junção espacial. Suas principais contribuições podem ser resumidas na obtenção de expressões de desempenho, em termos de operações de E/S e processamento, para todos algoritmos analisados; no desenvolvimento de um novo algoritmo, chamado HSSJ; e de um otimizador de consultas capaz de indicar o algoritmo mais rápido para realizar uma operação específica.

A obtenção de expressões de desempenho teve por objetivo avaliar os algoritmos a partir de seus parâmetros principais, estabelecendo, *a priori*, características que favorecem ou prejudicam seu desempenho. Como há um grande número de algoritmos, a divisão em classes, apresentada na figura 2.6, é importante, uma vez que, dentro de uma classe, o desempenho dos vários algoritmos tende a ser semelhante. A partir deste estudo, foram escolhidos quatro[6] algoritmos para serem implementados em um ambiente de testes e demonstrar a correção das expressões. Os testes realizados permitiram, também, verificar como algumas variáveis que não aparecem nas expressões, como densidade dos objetos no espaço, influenciam os algoritmos.

O grande número de testes realizados, envolvendo conjuntos de dados reais e artificiais, permitiu que a análise de cada algoritmo fosse aprofundada e a comparação entre eles envolvesse diferentes situações. Os resultados do algoritmo ISSJ são, pelo nosso conhecimento, os primeiros apresentados na literatura, embora o algoritmo seja uma evolução natural dos outros dois da mesma classe.

Os tempos de resposta obtidos sugerem que o algoritmo mais popular, o STT, não é uma escolha adequada para conjuntos com cardinalidades médias e grandes, por apresentar um tempo de resposta mais alto. O algoritmo ISSJ reduz o tempo de resposta na medida em que é possível ordenar os conjuntos em memória, reduzindo muito o número de operações de disco necessárias para ordenação externa. Já o algoritmo PBSM apresenta seu melhor desempenho quando há pouca memória, o que obriga a criação de muitas partições, dividindo o problema de junção em várias partes.

O algoritmo HHSJ, apresentado no capítulo 4, aproveita as características positivas dos algoritmos estudados. Este algoritmo é o único que mantém os envelopes de objetos em arquivos hash, como forma de agilizar o acesso a dados, e também é o único a

[6]O algoritmo de Laços Aninhados foi implementado, mas os resultados inicias confirmaram a expectativa que os tempos de respostas seriam muito superiores, assim, não há um detalhamento dele ao longo do texto. Alguns destes resultados estão no anexo B.

aproveitar histogramas da distribuição dos objetos para otimizar o processo de particionamento. Os resultados obtidos demonstraram que ele apresenta o menor tempo de resposta na maioria das operações de junção espacial realizadas.

O HHSJ é mais rápido que os demais para a junção com conjuntos grandes. Por exemplo, na operação com mais objetos envolvidos, entre *ca_stream* e ca*_stream*, o HHSJ realizou a operação em aproximadamente 25s, enquanto que o segundo algoritmo realizou em torno de 100s, ou seja, é quatro vezes mais rápido. Para conjuntos médios e pequenos, em geral, o HHSJ também é mais rápido, embora a diferença de tempo de resposta não seja sempre tão alta.

A partir da existência de vários algoritmos, surge a necessidade de um módulo de otimização de consultas que seja capaz de indicar o melhor algoritmo para realizar determinada junção espacial. Com este objetivo, foi desenvolvido um otimizador de consultas baseado em custos que aproveita as expressões de desempenho formuladas no capítulo 2.

Os testes realizados indicam que o otimizador tem uma taxa de acerto em torno de 90%, errando, preferencialmente, nas operações com conjuntos pequenos. Ainda, o tempo de execução do otimizador é uma fração mínima da diferença de tempo entre o melhor algoritmo e a segunda alternativa. Por fim, foi possível estabelecer algumas estratégias para otimizar os algoritmos existentes, quando implementados em um SGBDG.

Portanto, entendemos que o trabalho de pesquisa atingiu os objetivos propostos: estudar a fase de filtragem dos diversos algoritmos de junção espacial já descritos na literatura, identificando seus pontos fortes e fracos; e propor novos algoritmos, que permitam realizar a junção espacial em menor tempo ou com menos recursos de memória e disco em cenários de processamento específicos.

A hipótese principal foi confirmada, pois a partir da análise de desempenho e testes de execução, foi possível estabelecer critérios quantitativos e heurísticos para escolher o melhor algoritmo de junção espacial. A segunda hipótese também foi comprovada, pois o algoritmo HHSJ foi desenvolvido aproveitando as melhores características de cada algoritmo estudado.

Bandi e Sun (2004) e Sun et alli. (2003) propuseram métodos de aceleração da algoritmos espaciais baseados em características de placas gráficas. Embora este hardware esteja mais barato, a integração com software de bancos de dados requer um esforço maior de implementação. As soluções desenvolvidas ao longo deste trabalho, por serem baseadas apenas em software, são mais facilmente integráveis a diferentes softwares existentes.

## 6.1 Trabalhos publicados

Ao longo do desenvolvimento do trabalho, alguns resultados já foram descritos em artigos submetidos e aceitos em conferências de bancos de dados e geoinformática:

- em Fornari e Iochpe (2003a) foi apresentado um algoritmo intitulado PBSM-Enhanced, que introduzia alterações no cálculo do número de partições do PBSM (PATEL e DeWITT, 1996), para ajustar seu desempenho, e a inclusão do *Reference Point Method*, para evitar duplicatas no conjunto resposta. Os testes incluíam apenas o número de operações de E/S.

- em Fornari e Iochpe (2003b) foi apresentado um algoritmo intitulado PBSM-NRQB que estende o anterior, acrescentando o uso de informações de geoestatística, através de histogramas, para obter um particionamento do espaço ótimo, adaptado à distribuição espacial dos objetos existentes nos dois conjuntos. Ao longo do artigo é desenvolvida a análise de desempenho do PBSM-NRQB e outros algoritmos e demonstrado que, na maioria dos casos estudados, o PBSM-NRQB apresenta um desempenho melhor que os concorrentes.
- em Fornari e Iochpe (2004), o trabalho anterior foi complementado, ao acrescentar testes de desempenho, relatando o tempo de resposta dos algoritmos, à análise anteriormente realizada. Assim, foi possível comprovar o desempenho do algoritmo frente aos demais, e posicioná-lo como uma alternativa vantajosa para ambientes onde há pouca memória disponível, como *palm-tops* ou servidores compartilhados de bases de dados geográficas.
- em Fornari, Comba e Iochpe (2006a) foi descrito um otimizador de consultas capaz de prever o tempo de resposta de diferentes algoritmos de junção espacial para realizar uma operação submetida pelo usuário. O otimizador baseia-se em estimativa de custos e considera variáveis que descrevem os conjuntos de dados, o ambiente computacional e fórmulas de desempenho em memória e acesso a disco. Nos testes realizados, o otimizador indicou o algoritmo correto em quase 90% das operações.
- em Fornari, Comba e Iochpe (2006b) um conjunto de regras heurísticas para a otimização de cada algoritmo, individualmente considerado, é apresentado. As regras permitem reduzir o tempo de resposta, podendo ser implementadas em SGBDG que incluam apenas um ou dois dos algoritmos de junção espacial estudados.

## 6.2 Trabalhos futuros

A partir do trabalho realizado, há várias alternativas para a sua continuidade.

O HHSJ utiliza uma estrutura de armazenamento de dados diferente dos demais, que pode ser aproveitada para realizar outras operações espaciais, como consulta por janela e busca por vizinhos próximos.

Como diversas estruturas de organização de dados espaço-temporais baseiam-se em árvores-R, este estudo pode ser estendido para incluir, também, este tipo de informação. Conjuntos de dados espaço-temporais possuem um número de objetos muito maior, justamente o tipo de conjunto para o qual o algoritmo STT apresentou os piores tempos de resposta e a maior diferença em relação ao HHSJ. Mesmo alguns trabalhos recentes estudaram a otimização de consultas espaciais complexas (MAMOULIS,2004), envolvendo combinações de seleções e junções espaciais, ou junções envolvendo muitos conjuntos (PAPADIAS e ARKOUMANIS, 2002)(CORRAL, 2004). São trabalhos importantes, mas cuja solução baseia-se no algoritmo STT, envolvendo árvores-R.

Com o aumento na utilização de arquiteturas paralelas ou clusterizadas, para os quais há somente algoritmos baseados em árvores-R, torna-se interessante investigar adaptações e otimizações para o HHSJ se aproveitar das características destas arquiteturas. Um fator favorável é a divisão do espaço de maneira regular e a existência de histogramas, que permitem dividir a junção espacial em tarefas menores, com complexidade idênticas, de maneira relativamente fácil.

Ainda, pode-se estender a comparação com outros algoritmos incluindo um trabalho recentemente publicado (LEE et al, 2006), que transforma objetos com extensão para

pontos em um espaço transformado. Estes pontos são indexados e, com base neles, realizada a junção espacial. Ou, como sugerem os autores, buscar novos algoritmos que manipulem pontos diretamente neste espaço transformado.

Por fim, acreditamos que este trabalho apresenta argumentos suficientes para justificar a inclusão de novos algoritmos e formas de organizar dados espaciais nos SGBDG atuais, uma vez que eles se restringem à variações do STT. Assim, um trabalho que pode ser benéfico para toda comunidade é incluir o HHSJ em um software livre, como o PostGIS.

# REFERÊNCIAS


ARGE, L. et al. Scalable Sweeping-Based Spatial Join. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATABASES, VLDB, 24., 1998, New York,USA. **Proceedings...** San Francisco,USA:Morgan Kaufmann, 1998. p. 570—581.

ARGE, L. et al. A Unified Approach for Indexed and Non-Indexed Spatial Joins. In: INTERNATIONAL CONFERENCE ON EXTENDING DATABASE TECHNOLOGY, EDBT, 7., 2000, Konstanz, Germany. **Proceedings...** Berlin:Springer-Verlag, 2000. p. 413-429.

ARONOFF, S. **Geographic Information Systems:** a Management Perspective. Ottawa,Canada: WDL Publications, 1989.

AZEREDO, P. A. de **Métodos de Classificação de Dados e Análise de suas Complexidades**. Rio de Janeiro: Campus, 1996.

AZEVEDO, L.G. et al. Approximate Spatial Query Processing Using Raster Signatures. In: BRAZILIAN SYMPOSIUM OF GEOINFORMATICS, GEOINFO, 7., 2005, Campos do Jordão. **Proceedings...** São José dos Campos:INPE:SBC, 2005. p. 403-421.

AZEVEDO, L.G. **Processamento Aproximado de Consultas em Bancos de Dados Espaciais Usando Assinaturas Raster.** 2005. 117 f. Tese (Doutorado em Ciências da Engenharia de sistemas e Computação) -. COPPE/UFRJ, Rio de Janeiro.

BANDI, N. et al. Hardware Accelaration in Commercial Databases: a Case Study of Spatial Operation. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATABASES, VLDB, 30., 2004, Toronto,Canada. **Proceedings...** New York:Springer-Verlag, 2004.

BECKMANN, N. et al. The R*-tree: an Efficient and Robust Access Method for Points and Rectangle. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 16., 1990, Atlantic City:USA. **Proceedings...** New York:ACM Press, 1990. p. 322-331.

BELUSSI, A.; BERTINO, E.; NUCITA, A. Grid Based Methods for Spatial Join Estimation. In: SYMPOSIUM OF ADVANCED DATABASES SYSTEMS, 11., 2003, Cetraro, Italy. **Proceedings...** Soveria Manelli,Italy:Rubbertino, 2003. p.49-60.

BELUSSI, A.; BERTINO, E.; NUCITA, A. Grid Based Methods for Estimating Spatial Join Selectivity. In: INTERNATIONAL SYMPOSIUM ON ADVANCES IN GEOGRAPHIC INFORMATION SYSTEMS, 12., 2004, Washington,DC,USA. **Proceedings...** New York:ACM Press, 2004. p.92-100.

BRAKATSOULAS, S.; PFOSER, D.; THEODORIDIS, Y. Revisiting R-Tree Construction Principles. In: EAST EUROPEAN CONFERENCE ON ADVANCES IN DATABASES AND INFORMATION SYSTEMS, 6., 2002. **Proceedings...** Berlin:Springer-Verlag, 2002. p. 149-162. (Lecture Notes in Computer Science, v. 2435).

BRINKHOFF, T.; KRIEGEL, H.P.; SEEGER, B. Efficient Processing of Spatial Joins Using R-Trees. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 19., 1993, Washington D.C, USA. **Proceedings...** New York:ACM Press, 1993. p. 237-246.

BRINKHOFF, T. A Robust and Self-Tuning Page-Replacement Strategy for Spatial Database Systems, In: INTERNATIONAL CONFERENCE ON EXTENDING DATABASE TECHNOLOGY, 8., 2002, Prague. **Advances in Database Technology**: Proceedings... Berlin:Springer Verlag, 2002. p. 533-552. (Lecture Notes in Computer Science, v. 2287)

BRINKHOFF, T. et al. Multi-Step Processing of Spatial Joins. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 30., 2004, Paris, France. **Proceedings...** New York:ACM Press, 2004. p. 197-208.

CAMARA, G. et al. Towards a Unified Framework for Geographical Data Models. In: SIMPÓSIO BRASILEIRO DE GEOINFORMÁTICA, 2, 2000, Campos do Jordão. **Anais...** [S.l.:s.n.], 2000. Disponível em: <www.geoinfo.info>. Acesso em 28 set. 2006.

CHEN, L.; CHOUBEY,R.; RUNDENSTEINER, E.A. Merging R-Trees: Efficient Strategies for Local Bulk Insertion. **Geoinformatica**, London, v.6, n.1, Mar., 2002. p.7-34.

CHRISMAN, N. **Exploring Geographic Information Systems.** New York, USA: John & Wiley Sons, 1997.

CHOUBEY, R.; CHEN, L.; RUNDENSTEINER, E.A. GBI: A Generalized R-Tree Bulk-Insertion Strategy. In: SYMPOSIUM ON VERY LARGE SPATIAL DATABASES, VLDB, Edinburgh, 1999. **Proceedings...** [S.l.:s.n.], 1999, p.91-108. Disponível em <http:// citeseer.ist.psu.edu/197560.html >. Acesso em 28 set. 2006.

CORRAL, A. et al. Multiway Distance Join Queries in Spatial Databases. **Geoinformatica**, London, v.8, n.4. p.373-402, Dec. 2004.

DAS, A.; GEHRKE, J.; RIEDEWALD, M.; Approximation Techniques for Spatial Data. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 30., 2004, Paris. **Proceedings...** New York:ACM Press, 2004. p. 695-706.

DE BERG, M. et al. **Computational Geometry.** 2 ed. Berlin:Springer-Verlag, 2000.

DITTRICH, J.P.; SEEGER, B. Data Redundancy and Duplicate Detection in Spatial Join Processing. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 16., 2000, San Diego, USA. **Proceedings...** New York:IEEE Press, 2000. p. 535-543.

EGENHOFER, M.; SHARMA, J. Reasoning About Topological Consistency. In: SYMPOSIUM ON LARGE SPATIAL DATABASE, 2., 1991, Zurich, Switzerland. **Proceedings...** [S.l.]:Springer-Verlag, 1991. p. 143-160. (Lecture Notes in Computer Science, v. 525) Disponível em <http://www.spatial.maine.edu/~max/RC7.html>. Acesso em 28 set. 2006.

ELMASRI, R.; NAVATHE, S.B. **Sistemas de Bancos de Dados:** Fundamentos e Aplicações. 4. ed., São Paulo:Pearson - Addison Wesley, 2005.

FORNARI, M.R.; IOCHPE, C. A New Algorithm for Spatial Join Based in Space Subdivision. In: SYMPOSIUM OF ADVANCED DATABASES SYSTEMS, 11., 2003, Cetraro:Italy. **Proceedings...** Soveria Manelli,Italy:Rubbertino, 2003. p.69-80.

FORNARI, M.R.; IOCHPE, C. Estendendo o Algoritmo PBSM com Partições Adaptáveis. In: SIMPÓSIO BRASILEIRO DE GEOINFORMÁTICA, GEOINFO, 5., 2003, Campos do Jordão.. **Anais...** São Paulo:INPE, 2003. 1 CD-ROM.

FORNARI, M.R.; IOCHPE, C. A Spatial Hash Join Algorithm Suited for Small Buffer Size. In: INTERNATIONAL SYMPOSIUM ON ADVANCES IN GEOGRAPHIC INFORMATION SYSTEMS, ACM GIS, 12., 2004, Washington, USA. **Proceedings:** GIS 2004. [S.l:s.n.], 2004. p.118-126.

FORNARI, M.R.; COMBA, J.L.; IOCHPE, C. Query Optimizer for Spatial Join Operations. In: INTERNATIONAL SYMPOSIUM ON ADVANCES IN GEOGRAPHIC INFORMATION SYSTEMS, ACM GIS, 14., 2006, Arlington, USA. **Proceedings...** A ser publicado.

GAEDE, V.; GÜNTHER, O. Multidimensional Access Methods. **ACM Computing Surveys**, New York, v.30, n.2, p.170-231, Apr. 1998.

GARCIA-MOLINA, H. **Implementação de Bancos de Dados** Rio de Janeiro: Campus, 2001.

GARCÍA, Y.J.; LÓPEZ, M.A.; LEUTENEGGER, S.T. A Greedy Algorithm for Bulk Loading R-Trees. In: ACM INTERNATIONAL SYMPOSIUM ON ADVANCES IN GEOGRAPHIC INFORMATION SYSTEMS, 6., 1998, Washington, D.C. **Proceedings...** New York:ACM Press, 1998. p. 163-174.

GATTI, S.D.; MAGALHÃES, G.C. A Comparison Among Different Syncronized Tree Transversal Algorithms for Spatial Joins. In: SIMPÓSIO BRASILEIRO DE GEOINFORMÁTICA, GEOINFO, 2., 2000, São Paulo. **Anais...** [S.l.:s.n.], 2000. Disponível em: <http://www.geoinfo.info/geoinfo2000/papers/013.pdf>. Acesso em: 28 set. 2006.

GURRET, C.; RIGAUX, P. The Sort/Sweep Algorithm: a New Method for R-Tree Based Spatial Joins. In: **Statistical and Scientific Database Management,** [S.l:s.n.] 2000, p.153-165. Disponível em <http:// citeseer.ist.psu.edu/314464.html>. Acesso em 28 set. 2006.

GUTTMAN, A. R-Trees: a Dynamic Index Structure for Spatial Searching. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 10., 1984, Boston, USA. **Proceedings...** New York:ACM Press, 1984. p.47-57.

HADZILACOS, T.; TRYFONA, N. A Model for Expressing Topological Integrity Constraints in Geographic Databases. In: MANOPOLULOS, Y.; NÁVRAT, P. (Ed) **Theories and Methods of Spatio-Temporal Reasoning in Geographic Space.** New York: ACM Press, 1992. p. 252-268.

HUANG, Y-W.; JING, N. A Cost Model for Estimating the Performance of Spatial Joins Using R-Trees. In: INTERNATIONAL CONFERENCE ON INFORMATION AND KNOWLEDGE MANAGEMENT, 9., 1997, Olympia, USA. **Proceedings...** , [S.l:s.n.], 1997. Disponível em: <http://citeseer.ist.psu.edu/huang97cost.html>. Acesso em 28 set. 2006.

IBM. **IBM DB2 Spatial Extender- User´s Guide and Reference Version 8**. Disponível em: <http://www-306.ibm.com/software/data/spatial/db2sb.pdf>. Acesso em 16 ago. 2005.

JACOX, E.H.; SAMET, H. Iterative Spatial Join. **ACM Transactions Database Systems**, New York, v.28, n.3, p. 230—256, Sept. 2003.

KOZIARA, T.; BICANIC, N. Sweeping Approach to Bounding Box Intersection. In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL PLASTICITY, 8., 2005, Barcelona, Spain. **Proceedings...** [S.l:s.n.], 2005. p.1-4.

KORTH, H. F.; SILBERSCHATZ, A.; SUDARSHAN, S. **Sistemas de Bancos de Dados.** 4 ed., São Paulo: Makron Books, 2005.

KOTHURI, R.K.; RAVADA, S. Efficient Processing of Large Spatial Queries Using Interior Approximations. In: INTERNATIONAL SYMPOSIUM ON ADVANCES IN SPATIAL AND TEMPORAL DATABASES, 7., 2005, Redondo Beach, USA. **Proceedings...** (Lecture Notes in Computer Science v. 2121), London:Springer, 2001. p. 404-424.

KOUDAS, N.; SEVCIK. K.C. Size Separation Spatial Join. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 17., 1997, Tucson, USA. **Proceedings...** New York:ACM Press, 1997. p. 324-335.

KOUDAS, N. **Fast Algorithms for Spatial and Multidimensional Joins.** 1998.110 f. (PhD Thesis). - University of Toronto, Department of Computer Science, Toronto, Canada.

KRIEGEL, H-P et al. Spatial Query Processing for High Resolution. In: INTERNATIONAL CONFERENCE OF DATABASES SYSTEMS FOR ADVANCED APPLICATIONS, 8., 2003, Tokyo, Japan. **Proceedings...** New York:IEEE Press, 2003. p.17-26.

LEE, T.; MOON, B.; LEE.S. Bulk Insertion for R-Tree by Seeded Clustering. In: INTERNATIONAL CONFERENCE ON DATABASE AND EXPERT SYSTEM APPLICATIONS, 14., 2003, Prague, Czech Republic. **Proceedings..**. Berlin:Springer-Verlag, 2003. p.129-138.

LEE, M.-J.; WHANG, K.-Y.; HAN, W.-S.; SONG, I.-Y., Transform-Space View: Performing Spatial Join in the Transform Space Using Original-Space Indexes. **IEEE Transactions on Knowledge and Data Engineering**, New York, v.18, n.2, p. 245-260, Feb. 2006.

LEUTENEGGER, S. T.; EDGINGTON, J. M.; LOPEZ, M.A. **STR: A Simple and Efficient Algorithm for R-Tree Packing.** Denver, USA: University of Denver, Mathematical and Computer Science Department, 1997. (TR 97-14). Disponível em: <http://citeseer.ist.psu.edu/leutenegger97str.html>. Acesso em: 28 set. 2006.

LO, M.-L.; RAVISHANKAR, C.V. Spatial Joins Using Seeded Trees. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 20., 1994, Minneapolis, USA. **Proceedings...** New York:ACM Press, 1994. p. 209—220.

LO, M.-L.; RAVISHANKAR, C.V. Generating Seeded Trees from Data Sets. In: INTERNATIONAL SYMPOSIUM ON ADVANCES IN SPATIAL DATABASES, 4., 1995, Portland, USA. **Proceedings...** Berlin:Springer-Verlag, 1995. p. 328-347.

LO, M.-L.; RAVISHANKAR, C.V. Spatial Hash-Joins In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 22., 1996, Quebec, Canada. **Proceedings...** New York:ACM Press, 1996. p. 247—258.

MAMOULIS, N.; PAPADIAS, D. Slot Index Spatial Join. **IEEE Transactions on Knowledge and Data Engineering,** New York, v.15, n.1, p. 211-231, Jan. 2003.

MAMOULIS, N.; PAPADIAS, D.; ARKOUMANIS, D. Complex Spatial Query Processing. **Geoinformatica**, London, v. 8, n. 4, p.311-346, Dec. 2004.

ORACLE. **Oracle Spatial User´s Guide and Reference 10g Release 1.** Dec. 2003. Disponível em: <http://download-west.oracle.com/docs/cd/B14117_01/appdev.101/b10826.pdf>. Acesso em 28 set. 2006.

ORENSTEIN, J.A. Spatial Query Processing in an Object-Oriented Database System. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 12., 1986, Washington, D.C., USA. **Proceedings...** New York:ACM Press, 1986. p. 326-336.

PAPADIAS, D.; ARKOUMANIS, D. Approximate Processing of Multiway Spatial Joins in Very Large Databases. In: INTERNATIONAL CONFERENCE ON

EXTENDING DATABASE TECHNOLOGY, 8., 2002, Prague, Czech Republic. **Proceedings...** Berlin:Springer Verlag, 2002. p. 179-196. (Lecture Notes in Computer Science, v. 2287)

PATEL, J. M.; DEWITT, D.J. Partition Based Spatial-Merge Join. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 22., 1996, Quebec, Canada. **Proceedings...** New York:ACM Press, 1996. p. 259—270.

PFOSER, D.; JENSEN, C.S. Trajectory Indexing Using Movement Constraints. **Geoinformatica**, London, v.9, n.2, p. 93-116, June 2005.

PULLAR, D.; EGENHOFER, M. Toward Formal Definitions of Topological Relations Among Spatial Objects. In: INTERNATIONAL SYMPOSIUM ON SPATIAL DATA HANDLING, 3., 1998. **Proceedings...** [S.l.:s.n.], 1998. p. 225-241.

REFRACTIONS RESEARCH, **PostGIS Manual**. Disponível em: <http://postgis.refractions.net/docs/postgis.pdf>. Acesso em: 28 set. 2006.

RIGAUX, P.; SCHOLL, M.; VOISARD, A. **Spatial Databases with Applications to GIS**. San Franscisco:Morgan Kaufmann, 2000.

ROCHA, S. R. **Análise do Desempenho de Junções Espaciais: uma Comparação Entre os Métodos Analítico e Experimental.** 2003. 107 f. Dissertação (Mestrado em Ciência da Computação) - Universidade Estadual de Campinas, UNICAMP, Campinas.

RUSCHEL, C.; IOCHPE, C.; LISBOA FILHO, J. Modelagem de Processos de Análise Geográfica Utilizando o *Framework* GeoFrame. In: SIMPÓSIO BRASILEIRO DE GEOINFORMÁTICA, GEOINFO, 5., 2003, Campos do Jordão, **Anais...** São Paulo:INPE, 2003. 1 CD-ROM.

SAFE SOFTWARE, **FME – The Feature Manipulation Engine**. Disponível em: <http://www.safe.com/products/fme/index.php>. Acesso em: 17 mar. 2006.

SUN, C.; AGRAWAL,D.; ABBADI, A.E. Hardware Selections and Joins. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 29., 2003, San Diego, USA. **Proceedings...** New York:ACM Press, 2003. p. 455-466.

THEODORIDIS, Y.; STEFANAKIS, E.; SELLIS, T. A Cost Models for Join Queries in Spatial Databases. In: IEEE INTERNATIONAL CONFERENCE DATA ENGINEERING, ICDE, 14., 1998, Orlando, USA. **Proceedings...** New York:IEEE Press, 1998. p.476-483.

THEODORIDIS, Y.; SELLIS, T. A Model for the Prediction of R-Tree Performance. In: ACM SIGACT-SIGMOD-SIGART SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS, ACM PODS, 16., 1996, Quebec, Canada. **Proceedings...** New York:ACM Press, 1996. p.161-171.

THEODORIDIS, Y.; STEFANAKIS, E.; SELLIS, T. An Efficient Cost Model for Spatial Joins Using R-Trees. **IEEE Transactions on Knowledge and Data Engineering**, New York, v.12, n.1 , p.19-32, Jan. 2000.

# ANEXO A ESTIMATIVA DE CUSTOS DE ALGUNS ALGORITMOS DE JUNÇÃO ESPACIAL

Este anexo complementa o capítulo 2 ao apresentar a estimativa de custos de alguns algoritmos de junção espacial que foram apenas descritos ao longo daquele capítulo.

## A.1 Algoritmo de Junção por Laços Aninhados

O algoritmo de junção por laços aninhados aproveita apenas o fato de que os objetos são armazenados em disco por blocos. Um bloco, normalmente, terá vários objetos. Considerando que a memória tem capacidade para $M_b$ blocos, o algoritmo lê repetidamente $M_b$-*1* blocos do conjunto *A* para memória. Em seguida, percorre todos os blocos do conjunto *B*, lendo um de cada vez para a única posição de memória disponível. Neste ponto, o algoritmo para verificar o predicado espacial entre objetos de dois conjuntos é executado. Após ler todos blocos do conjunto *B*, novos $M_b$-*1* blocos do conjunto *A* são lidos para memória. O algoritmo repete-se até esgotar todos blocos do conjunto *A* e do conjunto *B*. A figura A.1 mostra este algoritmo.

```
Procedure Laços-Aninhados(A,B: conjunto de objetos
                               espaciais)
Begin
For each grupo de (M-1) blocos in A do
    Ler grupo para memória
    Ordenar objetos por xmin
    For each bloco in B do
        Ler bloco para memória
        Ordenar os objetos por xmin
        Teste do predicado de junção por plane sweep
        Para cada par de objetos que atender ao
            predicado, gravar no conjunto resposta
    End-for
End-for
End-Proc
```

Figura A.1 : Algoritmo de junção espacial por laços aninhados.

Uma vez que os objetos estejam em memória, é possível verificar o predicado para pares de objetos, utilizando um algoritmo baseado em uma *sweep line*. Após a leitura de *M-1* blocos de objetos do conjunto *A* para memória, estes objetos devem ser ordenados por um dos eixos, utilizando, por exemplo, a coordenada $x_{min}$. A cada bloco do conjunto *B* lido para memória, seus objetos também devem ser ordenados pela mesma

coordenada. O algoritmo de *plane sweep* terá então duas listas ordenadas de objeto como entrada. Para manipulá-los, deverá manter duas listas de objetos ativos, $\varepsilon_A$ para objetos do conjunto $A$, $\varepsilon_B$ para objetos do conjunto $B$. Ao ler um objeto do conjunto $A$, este deve ser verificado com objetos na lista $\varepsilon_B$, e vice-versa.

Este algoritmo é uma adaptação direta do algoritmo de laços aninhados já conhecido em bancos de dados relacionais, alterando apenas a forma como o predicado é verificado. O número de operações de disco, considerando $b_A$ e $b_B$ o número de blocos dos conjuntos $A$ e $B$, respectivamente, é dado por

$$disco_{LA} = |b^A| + \frac{|b^A|}{(M_b - 1)} \times |b^B|$$

O desempenho do algoritmo de laços aninhados em memória é baseado em três etapas: ordenar objetos do conjunto $A$, ordenar objetos de cada bloco do conjunto $B$ e realizar o algoritmo de *plane sweep* entre objetos presentes em memória. Sendo $f_A$ o fator de bloco de um conjunto $A$, ou seja, o número de objetos em um bloco de disco, a ordenação dos objetos do conjunto $A$ pode ser realizada por um algoritmo da ordem de $O(f^A \times (M_b - 1) \log(f^A \times (M_b - 1)))$. O número de grupos de objetos do conjunto $A$ que devem ser formados é $\lceil \frac{n^A}{(f^A \times (M_b - 1))} \rceil$.

Esta operação é realizada para repetidos grupos de objetos do conjunto $A$, resultando um desempenho final da ordem de $O(n^A \log(f^A \times (M_b - 1)))$.

A ordenação dos objetos de cada bloco do conjunto $B$ possui um desempenho da ordem de $O(f^B \log f^B)$. Esta operação é repetida para cada bloco, toda vez que a página é lida. Uma otimização possível é, na primeira vez, gravar a página já ordenada, evitando subseqüentes reordenações, mantendo o algoritmo com um desempenho mais aceitável, na ordem de $O(n^B \log f^B)$.

O algoritmo de *plane sweep* é repetido para cada grupo de objetos do conjunto $A$ e cada bloco do conjunto $B$, num total de $Rep = (b^A + b^B)/(M_b - 1)$ vezes. A cada repetição, o número de objetos do conjunto $A$ é $f^A \times (M_b - 1)$ e $f^B$ objetos do conjunto $B$. Como os dois conjuntos possuem a mesma estrutura $f^A = f^B = f$. Considerando que o número de intersecções seja semelhante em cada repetição, o desempenho do algoritmo, a cada vez, será $O(c/Rep + f.M_b \log f.M_b)$. Finalmente, para processar todos objetos, o desempenho do algoritmo em memória será da ordem de

$$cpu_{LA} = O(c + \frac{((n^A + n^B).M_b)}{(M_b - 1)} \log f.M_b).$$

As principais vantagens deste algoritmo são a simplicidade de implementação e a independência de qualquer outra estrutura de armazenamento, o que torna possível de ser utilizado em qualquer situação.

Se um dos conjuntos for menor que a memória disponível, digamos o conjunto $A$, sendo possível mantê-lo completo em memória, bastará uma passagem pelo conjunto $B$, melhorando bastante seu desempenho. O número de operações de disco fica reduzido para $b^A + b^B$, que seria o mínimo possível. Ao invés de colocar um bloco do conjunto B por vez, em memória, pode-se colocar $M_b - b^A$ blocos. O número de repetições será

$Rep = \frac{b^B}{(M_b - b^A)}$. A cada repetição, são manipulados $f^b.M_b$ objetos, e o algoritmo de *plane sweep* apresenta um desempenho da ordem de $O(\frac{c}{Rep} + f.M_b \log f.M_b)$. Assim, a realização completa do algoritmo de junção espacial teria um desempenho na ordem de

$$O(c + \frac{(n^B . M_b)}{(M_b - b^A)} \log f.M_b) .$$

## A.2 Junção quando apenas um conjunto possui árvore-R

Um caso especial de junção espacial baseada em árvores-R ocorre quando apenas um dos conjuntos estiver indexado por uma árvore-R. Nesta subseção, considera-se o conjunto *A* como indexado por uma árvore-R, e o conjunto *B* como não-indexado, em todas situações, apenas para facilitar o entendimento.

### A.2.1 *Scan Index*

O algoritmo *Scan Index* utiliza, para cada objeto do conjunto *B*, seu envelope como "janela" e realiza uma consulta por janela na árvore de índice do conjunto *A*, para localizar objetos que atendam ao predicado de junção.

O número mínimo de acessos a disco para a realização da junção espacial é dado pela simples multiplicação do número de objetos do conjunto *B* pela altura da árvore *A*, $(n_B.h_A)$, considerando-se que cada objeto de *B* possui intersecção apenas com objetos que estejam todos em uma única folha da árvore *A*. Na verdade, um objeto do conjunto *B* pode intersecionar vários nodos folhas da árvore-R do conjunto *A*, resultando em um número de operações de E/S aproximado de

$$disco_{SI} = n^B \times \sum_{i=1}^{h^A - 1} n^A (s^A_{x,i} + s^B_x)(s^A_{y,i} + s^B_y)$$

A existência de um *buffer* de memória favorece este algoritmo, pois os nodos da árvore-R são repetidamente lidos. Na hipótese de haver memória suficiente para manter toda árvore-R, há uma redução no número de operações de E/S para

$$disco_{SI} = b^B + n^A_R$$

Em termos de operações de CPU, deve-se considerar que o número de nodos da árvore *A* que serão comparados com um objeto da árvore *B* é dado pelo segundo termo da multiplicação. A cada comparação, deve-se percorrer as entradas do nodo até uma possuir a coordenada mínima no eixo *x* maior que a coordenada máxima no mesmo eixo do objeto *B*. Como previsão, pode-se dizer que esta busca ocorrerá em metade das entradas de um nodo, em média, totalizando um desempenho na ordem de

$$cpu_{SI} = O(n^B \times (\frac{Fanout}{2}) \sum_{i=1}^{h^A - 1} n^A (s^A_{x,i} + s^B_x)(s^A_{y,i} + s^B_y))$$

Esta estratégia é dependente da memória disponível para manter nodos da árvore-R. Quanto maior a memória disponível, menor o número de operações de E/S. Porém, o processamento necessário não é afetado pela memória disponível.

### A.2.2 *Build&Match*

A estratégia do algoritmo de *Build&Match* consiste em construir a árvore-R correspondente ao conjunto *B* como uma primeira etapa do processamento, e, depois, aplicar o algoritmo STT apresentado na seção 2.5.

Como todos objetos do conjunto *B* já são conhecidos, pode-se utilizar a estratégia de construção otimizada, que realiza um pequeno número de operações de disco e processamento.

O número de operações de E/S é soma da construção da árvore-R (13) e da junção espacial entre as duas árvores-R (17).

$$disco_{BM} = O(b^B + 2\mathrm{n}_R^B - 1) + disco_{STT}$$

Quanto ao processamento em memória, a ordem do algoritmo é

$$cpu_{BM} = O(n^B \log n^B + cpu_{STT})$$

Um ganho adicional é que os últimos nodos construídos, justamente o nodo raiz e níveis imediatamente inferiores, possuem alta probabilidade de se encontrarem no *buffer* quando do início do algoritmo de junção espacial. Além disso, a árvore-R construída pode ser mantida e aproveitada em outras operações de junção espacial, o que justificaria o consumo de recursos na sua construção.

### A.2.3 *Seeded-Tree*

O algoritmo *Seeded-Join*, proposto por Lo e Ravishankar (1994, 1995), realiza a indexação do conjunto *B* utilizando uma estrutura específica, a *Seeded-Tree*.

Na primeira fase, o algoritmo copia os níveis superiores da árvore-R já existente para uma nova estrutura, chamada de *Seeded-Tree*. Estes nodos são chamados de nodos sementes.

Na segunda fase, para cada nodo folha da *Seeded-Tree*, é criada uma lista de objetos e os objetos do conjunto *B* são inseridos em uma destas listas. Os nodos sementes não são alterados, servindo como guia para definir a inserção dos objetos. Idealmente, as listas são mantidas em memória, mas, em casos práticos, não há memória disponível suficiente, obrigando a gravação em disco.

Na terceira fase, cada lista é transformada em uma sub-árvore-R, de maneira individual, sem propagar alterações para os nodos sementes, que permanecem fixos. Assim, a árvore final pode não ser balanceada, se a distribuição espacial dos objetos entre os dois conjuntos for diferente.

Obtida a nova árvore, é possível aplicar o algoritmo STT aos dois conjuntos. A principal vantagem deste algoritmo baseia-se na igualdade entre os níveis superiores das duas árvores. Deste modo, haverá menos intersecções entre sub-árvores diferentes, reduzindo o número de operações de E/S e CPU.

O número de operações de disco do algoritmo, segundo Mamoulis e Papadias (2003), deve incluir a cópia (gravação) e leitura dos nodos dos níveis sementes. Para tanto, considera-se que os níveis sementes são a raiz e o nível imediatamente abaixo. Na segunda etapa do algoritmo, os objetos do conjunto *B* são lidos e incluídos em arquivos seqüenciais. Na terceira etapa, cada um destes arquivos é lido e a árvore-R é completada. Então, para a construção da árvore *B*, tem-se

$$(1+n_{h-1}^{A})+3\mathrm{b}^{B}+n_{R}^{B}$$

Para realizar a junção espacial, entre uma árvore-R e uma *Seeded Tree*, pode-se utilizar a mesma estimativa para o caso de junção entre duas árvores-R comuns.

$$disco_{SEEDT}=(1+n_{h-1}^{A})+3\mathrm{b}^{B}+n_{R}^{B}+disco_{STT}$$

A construção da árvore-R requer, para cada objeto do conjunto *B*, percorrer dois nodos (raiz e um nodo intermediário), no mínimo, para identificar em qual das listas incluir o objeto. Após, para cada uma destas listas, é criada uma árvore-R. A criação de cada sub-árvore-R pode ser otimizada, pois todos objetos já estão disponíveis. Se a distribuição dos objetos no espaço for semelhante, o número de objetos em cada lista (sub-árvore) é *n/q*, onde *q* é o número de listas. Somando ao algoritmo a etapa de junção espacial com as duas árvores-R já formadas, apresentará um desempenho da ordem de

$$cpu_{SEEDT}=O(2.\mathrm{n}^{B}+n^{B}\log(\frac{n^{B}}{q})+n_{R}^{B}\times fanout+cpu_{STT}),$$

que pode ser simplificado para

$$cpu_{SEEDT}=O(n^{B}\log(\frac{n^{B}}{q})+cpu_{STT})$$

Ao aproveitar os níveis superiores da árvore-R já existente para construir a *Seeded-Tree*, o número de nodos intermediários que possuem intersecção é reduzido, o que beneficia a junção espacial. Porém, a *Seeded-Tree* pode ser desbalanceada, prejudicando seu aproveitamento para outras operações, como seleção por janela e junção espacial com outros conjuntos de dados.

### A.2.4 *Sort Sweep-Based Spatial Join*

O algoritmo *Sort Sweep-Based Spatial Join*, de Gurret e Rigaux (2000), baseia-se em uma abordagem de ordenação.

Primeiro, os envelopes dos nodos folhas da árvore-R existente são ordenados por uma coordenada, por exemplo, $y_{min}$. Os nodos folha, a maior parte da árvore-R, não precisam ser lidos, pois seus envelopes encontram-se no nível imediatamente superior. Além de reduzir o número de operações de E/S, há grande possibilidade de manter todos envelopes em memória. Após, os objetos do conjunto *B* são ordenados pela mesma coordenada $y_{min}$, utilizando um algoritmo de ordenação interna ou externa.

O terceiro passo consiste em subdividir o eixo em que os objetos não estão ordenados, neste caso, *x*, em faixas. Os limites das faixas são definidos pelos objetos do conjunto *A,* permitindo que cada nodo folha, com seus objetos, seja alocado a uma única faixa, sem replicações. Os objetos do conjunto *B* são alocados posteriormente. Se interseccionarem mais de uma faixa, suas cópias são inseridas em cada faixa. A figura A.2 ilustra a definição de faixas (*strips*) em um espaço com alguns objetos do conjunto *A*, que são a referência para delimitar as faixas.

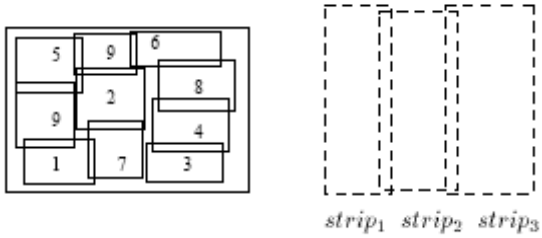



Figura A.2 : Exemplo de faixas no algoritmo *Sort Sweep-Based Spatial Join*. (GURRET e RIGAUX, 2000) .

Finalmente o algoritmo de *plane sweep* é realizado para verificar pares de objetos que atendam ao predicado de junção espacial. Para cada faixa há duas listas de objetos ativos, o que permite processar os objetos de cada faixa de maneira independente. Quando é encontrado um envelope de um nodo folha do conjunto *A*, este é substituído pelos objetos contidos no nodo, que serão incluídos na lista de ativos da sua faixa e verificados com objetos do conjunto *B,* também da sua respectiva faixa.

Segundo os autores, a árvore-R existente é lida uma única vez. No primeiro passo, até o nível superior às folhas, no terceiro passo, as folhas. O custo de ordenação interna ou externa do conjunto *B* já foi apresentado. Na terceira etapa, todos objetos do conjunto *B* devem ser lidos uma única vez. A duplicação de objetos é realizada em memória. Assim, na melhor hipótese, o número de operações de disco será

$$disco_{SSBSJ} = 3\mathrm{b}^B + n_R^A$$

Em termos de processamento em memória, a divisão do espaço em faixas apresenta um grande avanço, pois, na verdade, é como se ocorresse um *plane sweep* para cada faixa, reduzindo o número de pares verificados. Supondo uma distribuição uniforme de objetos no espaço, e *r* o fator de replicação do conjunto *B*, temos que o desempenho da etapa de junção, em cada faixa, será da ordem de

$$O(c_{Reg} + \frac{(n^A + n^B.r)}{l} \log \frac{(n^A + n^B.r)}{l})$$

Incluindo a ordenação das $n_A^R$ folhas da árvore *A,* a ordenação externa do conjunto *B* e o processamento de todas faixas, totaliza um desempenho da ordem de

$$cpu_{SSBSJ} = O(n_h^A \log n_h^A + n_B^R \log M_o . p + c_{Reg} + (n^A + n^B.r) \log \frac{(n^A + n^B.r)}{l}),$$

podendo ser reduzido para

$$cpu_{SSBSJ} = O(c_{Reg} + (n^A + n^B.r) \log \frac{(n^A + n^B.r)}{l}),$$

que é melhor que o mesmo algoritmo sem utilizar as faixas.

O algoritmo é complementado com uma previsão para o caso de não ser possível processar todas as faixas ao mesmo tempo, por limitações de memória. Neste caso, os objetos ativos de uma faixa são colocados em um arquivo auxiliar e o processamento prossegue para as demais faixas. Ao final, será necessário retomar a faixa, recolocando os objetos ativos em memória e relendo a árvore-R e os objetos do conjunto *B*, a partir do ponto de interrupção.

### A.2.5 *Priority Queue-Driven Process*

O algoritmo *Priority Queue-Driven Process* (Arge, 2000) é uma adaptação do algoritmo *Scalable Sweep-based Spatial Join* para o caso de um dos conjuntos possuir uma árvore-R.

Ele também ordena o arquivo de objetos não-indexados por um dos eixos, mas aproveita a árvore-R existente como entrada no algoritmo de *plane sweep*, que verifica pares de objetos que atendem ao predicado de junção espacial. Da mesma forma que o anterior, este também é um algoritmo baseado em ordenação, modificado para o caso de já existir uma árvore-R em um dos conjuntos.

O algoritmo de *plane sweep* é adaptado para realizar um caminhamento sobre a árvore-R e identificar o próximo objeto a ser comparado com um objeto do outro conjunto. No nível da raiz, a *sweep line* identifica todos nodos inferiores ativos, que são incluídos, de maneira ordenada, em uma lista auxiliar. Quando o algoritmo alcança um nodo folha, que contém os envelopes dos objetos, os objetos ativos são colocados em uma lista ordenada. O objeto com a menor coordenada é comparado com o próximo objeto do outro conjunto.

O número de operações de E/S deve incluir a leitura da árvore-R uma única vez, a ordenação dos objetos do conjunto *B* e sua releitura para processamento pelo algoritmo de *plane sweep*, resultando

$$disco_{PQDP} = 3b^{B} + n_R^A$$

Quanto ao processamento, o algoritmo basicamente ordena o conjunto *B* e realiza o *plane sweep*, aproveitando a árvore existente, que é percorrida de maneira específica. Portanto, pode-se definir a complexidade do algoritmo como sendo

$$cpu_{PQDP} = O(n^B \log M_o . p + c + (n^A + n^B)\log(n^A + n^B)) \quad ,$$

ou, apenas,

$$cpu_{PQDP} = O(c + (n^A + n^B)\log(n^A + n^B))$$

Este algoritmo é uma variação do anterior, *Sort Sweep-Based Spatial Join,* alterando apenas a forma de aproveitar a árvore-R existente e eliminando a divisão em faixas. Também não há construção de uma árvore-R para o conjunto *B*.

### *A.2.6 Priority Queue-Driven Process* para arquivos indexados

O algoritmo PQDP também pode ser utilizado se os dois arquivos de dados já possuírem uma árvore-R indexando-os, sendo uma alternativa ao algoritmo STT. Neste caso, as duas árvores são percorridas por uma *sweep-line*, que identifica os nodos filhos a serem incluídos. Quando um nodo folha é lido para memória, seus objetos podem ser inseridos na lista de objetos ativos, permitindo o teste do predicado espacial de pares de objetos.

Desta maneira, cada árvore-R é lida uma única vez, garantindo um número mínimo de operações de E/S

$$cpu_{PQDP2} = n_R^A + n_R^B$$

Embora o modo de localizar pares de nodos que se interseccionem seja diferente do STT, para funcionar corretamente, os mesmos nodos devem ser verificados. Portanto, o

desempenho do algoritmo é da mesma ordem, dependente do número e tamanho médio dos nodos em cada nível.

$$cpu_{PQDP2} = cpu_{STT} = O(c_{STT} + CompH \times 2\,Fanout \log(2\,Fanout))$$

Como os autores destacam, é necessário manter vários nodos em memória, além dos objetos ativos, requerendo um espaço de memória maior que o STT. Se não houver suficiente memória, é possível adaptar o algoritmo, mantendo os objetos em arquivos temporários, o que aumentaria as operações de E/S.

## A.3 Algoritmos Baseados em Subdivisão do Espaço

Algoritmos de junção espacial baseados em subdivisão do espaço dividem o espaço de objetos (universo) em subespaços e procuram resolver o problema da junção para os objetos pertencentes a cada um dos subespaços, podendo ser classificado como um método de divisão e conquista. Em princípio, a divisão é realizada durante a junção, mas se já houver uma anterior, esta pode ser aproveitada.

### A.3.1 Spatial Hash Join

Lo e Ravishankar (1996) propuseram outro algoritmo, chamado *Spatial Hash Join* (SHJ) para realizar a operação de junção espacial. Inicialmente, o algoritmo calcula o número de partições, considerando a cardinalidade dos conjuntos. O método de cálculo utilizado é semelhante ao que determina o número de níveis que devem ser copiados para uma *Seeded-Tree*, também proposta pelos autores (LO e RAVISHANKAR, 1995). O resultado do método é um número de partições bastante alto, por exemplo, para dois conjuntos de 100.000 objetos, mais de 3.000 partições, dependendo da memória disponível.

Após, o algoritmo realiza uma amostragem dos objetos pertencentes ao conjunto *A* Com base nesta amostra, determina o divisão inicial do espaço em células retangulares, uma célula para cada partição. Esta divisão é irregular, pois depende da distribuição dos objetos no universo. Para cada objeto do conjunto *A*, o centro do objeto é calculado. O objeto é inserido na célula com menor distância entre o centro da célula e o centro do objeto. Se necessário, os limites da célula são expandidos para incorporar todo o objeto. O número de células se mantém constante, mas há intersecção entre elas. A figura A.3 mostra um grupo de objetos do conjunto *A* e as partições resultantes.

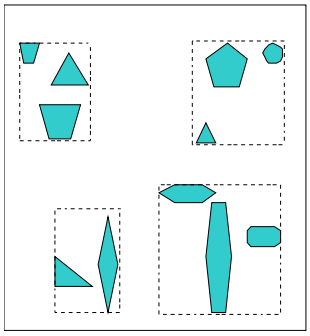


Figura A.3 : Partições resultantes após a alocação de objetos do conjunto *A*.

Concluída a alocação do conjunto *A*, procede-se a alocação dos elementos do conjunto *B*. As células já definidas para *A* são mantidas, inclusive as suas geometrias, e cada objeto de *B* é inserido em todas as células que interseccionar. O resultado da utilização das mesmas partições sobre o conjunto *B* pode ser um significativo

desbalanceamento do número de objetos por partição, se a distribuição espacial dos objetos for diferente.

Na etapa de junção, cada partição, contendo objetos dos dois conjuntos, é trazida para memória. Os elementos de cada célula são verificados de acordo com o predicado da junção. Pode ocorrer do número de objetos exceder a disponibilidade de memória, obrigando a um reparticionamento, mas esta é uma situação improvável, devido ao grande número de partições que é utilizado. O algoritmo completo pode ser visto na figura A.4

```
Procedure SHJ(A,B: conjunto de objetos espaciais)
i, npart: integer
a, b: objeto espacial
c_a: conjunto de objetos espaciais
c_cel: conjunto de células
c: célula
Begin
npart = calculo do número de células
c_a = Amostra o conjunto A
Define npart células com base no conjunto c_a
For each a in A do
    C = célula mais próxima do centro de a
    Insere (a,c)
    If  a.mbr esta fora de c.mbr
        Then  reajusta c.mbr
    End-If
End-for
For each b in B do
    c_cel = conjunto de células que o objeto b intersecciona
    For each c in c_cel
        insere (c,b)
    End-for
End-for
For i=1 to ncel do
    Ler objetos de A pertencentes a célula i
    Ler objetos de B pertencentes a célula i
    Aplicar predicado de junção aos objetos
    Gravar pares de objetos que atendam ao predicado
Next i
End-proc
```

Figura A.4 : Algoritmo *Spatial Hash Join*.

O algoritmo, na primeira etapa, lê todos objetos do conjunto *A* e os grava em partições, sem réplicas. Objetos do conjunto *B* também são lidos, mas a gravação pode incluir réplicas. Na etapa de junção, cada partição é lida para memória. Se todo conjunto de objetos da partição couber em memória, não são necessárias leituras adicionais. Se, no entanto, o número de objetos for maior que a memória disponível, o algoritmo realiza a junção por laços aninhados. Como o número de partições é maior que no PBSM, o número de partições com mais objetos que a memória disponível é menor. Assim, consideramos, de maneira otimista, que não ocorre *overflow* em nenhuma partição. O total de operações de E/S, pode ser expresso da seguinte forma

$$disco_{SHJ} = 3b^{A} + (1 + 2\,r^{B})\,b^{B}$$

O desempenho do algoritmo, em termos de processamento, é dependente do método de alocação dos objetos à suas partições. Como as partições não são regulares, podendo

estar distribuídas de maneira não-uniforme no espaço, uma opção é organizá-las em uma árvore-R, que envolve relativamente poucos objetos e pode ser mantida em memória. Nesta etapa, para cada objeto, é realizada uma consulta em janela sobre a árvore, sendo o objeto a ser alocado à "janela". O desempenho pode ser expresso por $ALOC = O(n \times (\frac{Fanout}{2}) \sum_{i=1}^{h_A - 1} P(s_{x,i}^{Part} + s_x^{obj})(s_{y,i}^{Part} + s_y^{obj}))$, sendo $s_{x,i}^{Part}$ e $s_{y,i}^{Part}$ o tamanho médio dos nodos da árvore-R de partições no nível *i*. Com a inserção de objetos do conjunto *A*, o tamanho médio dos nodos tende a crescer progressivamente, uma vez que o número de partições não é alterado, tornando a inserção progressivamente mais custosa. Durante a inserção de objetos do conjunto B, as células não são alteradas, ou seja, o custo de inserção varia apenas com relação ao tamanho do objeto. Toda vez que um objeto não esta contido em uma célula de partição, a célula mais próxima é aumentada.

Na etapa de junção, quando pares de objetos pertencentes às partições são verificados pelo algoritmo de *plane sweep*, o desempenho é determinado pelo número de objetos em cada partição. O número de pares comparados, *c*, é influenciado pela divisão irregular do espaço. No geral, o desempenho do algoritmo completo é da ordem de

$$cpu_{SHJ} = O(ALOC^A + ALOC^B + c_{Irreg} + (n^A + r^B n^B) \log \frac{(n^A + r^B n^B)}{P})$$

Da mesma forma que no PBSM, um número maior de partições favorece o algoritmo, porém pode produzir um número de réplicas muito alto, resultando uma questão de otimização difícil, pois esta vinculada ao tamanho médio dos objetos, a qualidade da amostra que define inicialmente as partições e o tamanho final de cada célula de partição.

### A.3.2 *Size Separation Spatial Join*

O algoritmo denominado *Size Separation Spatial Join* (S3J) foi proposto por Koudas e Sevcik (1997) e está baseado em *Filter-Trees* (KOUDAS, 1998). O algoritmo subdivide o espaço através de uma *Filter-Tree*, sem construir uma árvore de índice completa.

O nível *j* de uma *Filter-Tree* divide o espaço através de $2^{(j+1)} - 1$ linhas igualmente espaçadas em cada dimensão. Por exemplo, para um espaço bidimensional, no nível 0 (raiz) da árvore, há uma linha vertical e outra horizontal subdividindo o espaço. No nível 1, há 3 linhas horizontais e 3 verticais.

Cada objeto é inserido no nível mais alto da árvore, no qual uma linha interseccionar seu envelope. Desta maneira, objetos grandes são alocados nos níveis mais altos da árvore, e objetos menores, nos níveis mais próximos às folhas, embora objetos pequenos possam ser alocados nos níveis mais altos da árvore por cruzarem uma linha de divisão.

A figura A.5 ilustra três objetos e a seqüência de partições para formar uma *Filter-Tree*. O objeto $a_1$, apesar de pequeno, é atravessado por uma linha no primeiro processo de partição, ficando alocado no nível 0 da árvore. O objeto $a_3$ é atravessado por uma linha na segunda partição, sendo alocado no nível 1 da árvore. Já o objeto $a_2$ somente no quarto processo de partição é atravessado por uma linha, sendo alocado no nível 3 da árvore.

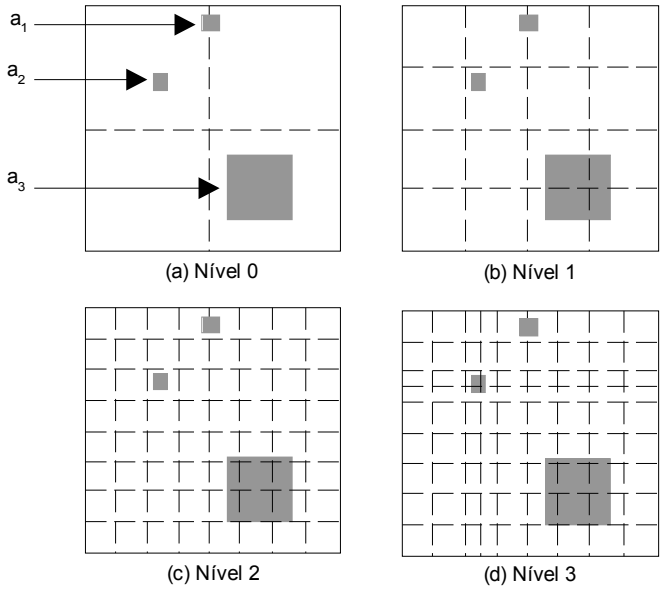



Figura A.5 : Exemplo do processo de partição para formar uma *Filter-Tree*.

O algoritmo inicia calculando dois valores para cada objeto pertencente aos dois conjuntos:

- Valor Hilbert do ponto central do envelope.
- Nível da árvore onde o objeto será inserido.

O envelope do objeto é definido pelas coordenadas $(x_{min}, y_{min})$ e $(x_{max}, y_{max})$, que representam cantos opostos da célula, os extremos de uma de suas diagonais. Pode-se utilizar o fato de que a representação computacional das coordenadas é mantida em números binários e definir o nível da árvore como sendo o menor valor do par composto pelo número de bits que $x_{min}$ e $x_{max}$ possuem iguais e pelo número de bits iguais em $y_{min}$ e $y_{max}$, sempre a partir do mais significativo. Para esta regra ser aplicada, os valores das coordenadas devem ser normalizados para o intervalo (0,1). A tabela A.1 mostra esta alternativa, utilizando os objetos da figura A.5.

Tabela A.1 : Exemplo de definição do nível em uma *Filter-Tree* a partir da representação em binário das coordenadas de envelopes de objetos.

| Objeto | $x_{min}$ | $x_{max}$ | $y_{min}$ | $y_{max}$ | Nível |
|---|---|---|---|---|---|
| $a_1$ | $.01111100_B$ ($.48_D$) | $.10000011_B$ ($.52_D$) | **.1110**$0110_B$ ($.90_D$) | **.1110**$1010_B$ ($.92_D$) | 0 |
| $a_2$ | **.1**$0001100_B$ ($.55_D$) | **.1**$1001100_B$ ($.80_D$) | **.0**$0100110_B$ ($.15_D$) | **.0**$1100110_B$ ($.40_D$) | 1 |
| $a_3$ | **.0100**$0010_B$ ($.26_D$) | **.0100**$1100_B$ ($.30_D$) | **.101**$00110_B$ ($.70_D$) | **.101**$10011_B$ ($.40_D$) | 3 |

Para cada nível identificado, dois arquivos são criados, um para objetos de *A*, outro para objetos de *B*. Para cada objeto é montado um registro contendo as coordenadas de seu envelope, o valor Hilbert associado a ele e um ponteiro para o objeto. Este registro é gravado no arquivo correspondente. A seguir, cada arquivo é ordenado pelo valor Hilbert.

A junção é realizada percorrendo, seqüencialmente, todos os arquivos de todos os níveis. Cada bloco de disco de cada arquivo é lido uma única vez e a memória deverá ser suficiente para conter um conjunto destes blocos em um mesmo instante de tempo.

Cada bloco contém registros que abrangem um certo intervalo de valores Hilbert, sendo $H_{min}$ o limite inferior e $H_{max}$ o limite superior. O bloco do conjunto *A*, nível *j*, é denotado como $A^j(H_{min}, H_{max})$. Para os arquivos de níveis $l = 0, 1, \ldots, L$, sendo *L* o maior nível existente, faz-se:

- combinar as entradas existentes em $A^l(H_{min}, H_{max})$ com as contidas em $B^{l-i}(H_{min}, H_{max})$, para $i = 0, \ldots, l$. Ou seja, um bloco de *A* será combinado com um bloco do mesmo nível em *B* e todos os blocos de níveis superiores em *B*.

- combinar as entradas existentes em $B^l(H_{min}, H_{max})$ com as contidas em $A^{l-i}(H_{min}, H_{max})$, para $i = 1, \ldots, l$. Ou seja, um bloco de *B* será combinado com todos blocos de níveis superiores em *A*.

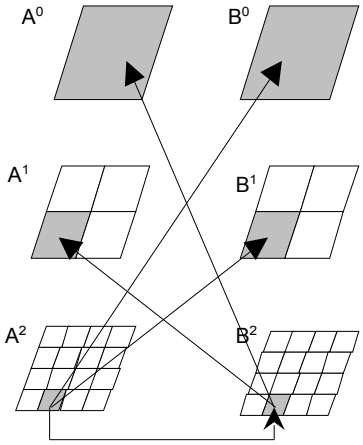



Figura A.6 : Exemplo de junção com o algoritmo S3J.

A figura A.6 mostra o espaço dividido em três níveis, o que resultaria em 6 arquivos de níveis. A página destacada em $A^2$, é combinada com as páginas indicadas em $B^2, B^1$ e $B^0$, somente. A página correspondente em $B^2$ é combinada com as página em $A^1$ e $A^0$. Desta maneira, todas intersecções entre objetos naquela área são detectadas. Nenhum par de objetos é testado mais que uma vez. O algoritmo pode ser visto completo na figura A.7.

```
Procedure SSSJ(A,B: conjunto de objetos espaciais)
i, ncel: integer
a, b: objeto espacial
/* cada objeto possui um ponto central,
   nível da árvore e  valor Hilbert */
f: arquivo de objetos espaciais
c_arqA: conjunto de arquivos para objetos de A
c_arqB: conjunto de arquivos para objetos de A
Begin
For each a in A do
    a.Hilbert = Hilbert(a.center)
    a.level = nível do objeto na Filter-Tree.
    Insere (a, c_arqA[a.level])
End-for
For each b in B do
    b.Hilbert = Hilbert(b.center)
    b.level = nível do objeto na Filter-Tree.
    Insere (b, c_arqB[a.level])
End-for
For each f in c_arqA e c_arqB do
     Ordena f pelo valor Hilbert de seus objetos
End-for
Percorre sincronizadamente todos arquivo de c_arqA e c_arqB,
fazendo a combinação de entradas.
End-Proc
```

Figura A.7 : Algoritmo *Size Separation Spatial Join*.

O algoritmo, na primeira etapa, lê todos objetos, calcula o valor Hilbert e os grava em um arquivo de nível, sem ocorrência de réplicas. Depois, cada arquivo de nível é ordenado. De maneira otimista, pode-se considerar que é possível ordenar a todos em memória. Finalmente, os arquivos de níveis são lidos de maneira sincronizada, para verificar pares de objetos, totalizando um número de operações de E/S igual à

$$disco_{SizeJ} = 5\, b^A + b^B$$

O desempenho do algoritmo é afetado pela complexidade do cálculo de valor Hilbert, que pode ser realizado por um algoritmo da ordem *O(Nbits)*, sendo *Nbits* o número de bits no qual um valor será representado. Os autores recomendam um valor entre 20 e 24 para *Nbits*, visando obter uma representação adequada. Este cálculo é repetido para todos objetos, resultando, apenas nesta etapa, um desempenho da ordem de $O((n^A + n^B)\, Nbits)$.

A etapa de ordenação é menos previsível, pois se baseia no número de objetos em cada arquivo de nível, que é dependente do tamanho dos objetos e sua distribuição espacial. Uma previsão otimista indicaria um número de objetos aproximadamente igual em cada nível, ou seja $n / Niveis$, que é repetido para cada arquivo de nível.

Finalmente, na última etapa é realizado um algoritmo de *plane sweep*, envolvendo todos objetos. Porém, o número de pares verificados é reduzido pela divisão do espaço em células, resultando em um desempenho da ordem de

$$cpu_{SizeJ} = O((n^A + n^B)(NBits) + n^A \log(\frac{n^A}{Niveis}) + n^B \log(\frac{n^B}{Niveis}) + \\ + \frac{c}{Niveis^2} + (n^A + n^B)(\log(n^A + n^B)))$$

Esta expressão pode ser simplificada, eliminando as parcelas pouco significativas, para

$$cpu_{SizeJ} = O\left(\frac{c}{Niveis^2} + (n^A + n^B)\log((n^A + n^B) \times Nbits)\right)$$

Um aperfeiçoamento deste algoritmo é realizado acrescentando-se um mapa de bits para cada nível do conjunto *A*. O mapa de bits do nível *j* conterá $4^j$ bits, que inicialmente possuem valor 0. Cada bit representa uma determinada célula naquele nível. Se um objeto de *A* é inserido em um arquivo de nível, ele ocupa uma certa célula naquele nível. O valor do respectivo bit é, então, passado para 1.

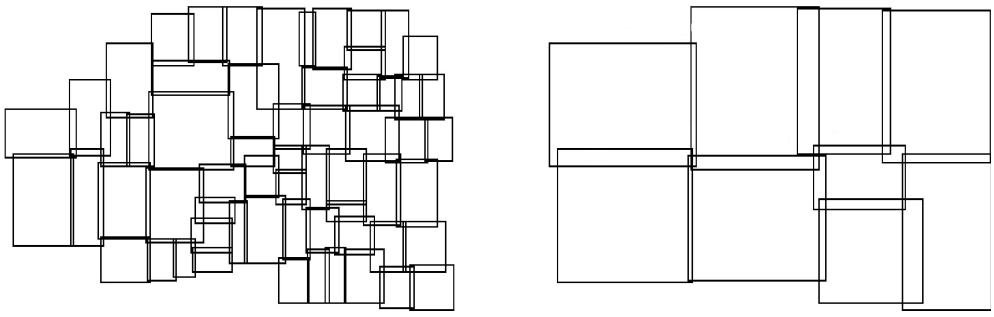
Ao percorrer o conjunto *B*, antes de inserir uma entrada em um certo nível (célula), os bits daquela célula em *A*, no mesmo nível e nas células correspondentes de nível superior são testados. Se todos forem iguais a zero, o objeto de *B* não é inserido, pois não há possibilidade se haver um objeto em *A* que atenda ao predicado de junção.

Este recurso reduziria os objetos de *B* manipulados, porém a complexidade do cálculo do valor Hilbert, o número de operações de E/S e a utilização de um algoritmo baseado em estruturas não usuais, tornam a implementação deste algoritmo menos interessante.

### A.3.3 *Slot Index Spatial Join*

O algoritmo *Slot Index Spatial Join* (SISJ) (MAMOULIS, 2003) é um método baseado em subdivisão do espaço, porém, deve ser aplicado quando apenas um dos conjuntos já possui uma árvore-R. A idéia principal do algoritmo é utilizar o envelope dos nodos dos níveis superiores da árvore-R para definir o particionamento do espaço. As extensões de cada partição são otimizadas pelo algoritmo de inserção na árvore-R, e podem ser utilizadas para dividir os objetos do conjunto *B*.

Um problema importante para o algoritmo é definir o nível adequado da árvore para utilizar suas entradas como partições. Um nível muito baixo tende a possuir muitas entradas, o que resultará muitas réplicas de objetos do conjunto *B*. Um nível muito alto pode resultar em poucas partições, com número de objetos maior que a capacidade de memória, obrigando a realizar reparticionamentos. Esta questão é mais crítica se considerar que o *Fanout* da árvore-R pode ser grande. Assim, a diferença do número de entradas entre um nível e outro é significativa, impedindo uma escolha adequada.



(a) todas entradas do nodo raiz (b) slots definidos

Figura A.8 : Exemplo de uma árvore-R e *slots* com base nela (MAMOULIS, 2003).

Para encontrar uma solução ótima, o algoritmo calcula um número ideal de partições e agrupa diversas entradas em um grupo, chamado *slot*. Assim, um *slot* contém várias

entradas da árvore-R, com suas respectivas sub-árvores, até o nível das folhas. Cada partição é definida por um *slot*. A figura A.8 mostra as entradas existentes no nível da raiz de uma árvore-R e o conjunto de *slots* definido com base nestas entradas.

Para realizar a junção espacial, o conteúdo de cada *slot*, objetos do conjunto *A*, é combinado com objetos do conjunto *B* alocados na respectiva partição. Neste momento, quatro situações podem ocorrer:

1. é possível manter todos objetos em memória, bastando realizar um algoritmo de *plane sweep*.
2. apenas o conteúdo do *slot,* objetos do conjunto *A,* pode ser mantido em memória, realizando a junção pelo algoritmo de *scan-index*.
3. apenas o conteúdo da partição, objetos do conjunto *B*, pode ser mantido em memória. Nesta situação é construída uma árvore-R, em memória, e objetos do conjunto *A* são verificados contra esta árvore, também pelo algoritmo de *scan-index*.
4. nenhum dos conteúdos pode ser mantido totalmente em memória. O algoritmo é reaplicado recursivamente.

A definição dos *slots* requer apenas a leitura dos níveis mais altos da árvore-R. Depois, cada objeto do conjunto *B* é alocado a uma ou mais partições, necessitando ser lido e gravado. Finalmente, cada partição (*slot*) é processado. Neste ponto, o restante da árvore-R é lido. O número de operações de disco pode ser expresso por

$$disco_{SISJ} = (1+2r)b^B + n_R^B$$

Quanto ao processamento, a etapa de definição dos *slots* pode ser executada utilizando uma das quatro alternativas propostas pelos autores, mas como envolve poucos objetos, não representa um custo relevante. A alocação dos objetos do conjunto B às partições requer identificar os *slots* que interseccionam cada um dos objetos. Com o formato irregular dos *slots*, é interessante organizá-los em memória de alguma forma, por exemplo, uma árvore binária de busca, para agilizar o processamento, reduzindo o custo para $O(n^B \times \log slots)$.

Na etapa de junção, para cada partição é utilizada uma das quatro alternativas listadas. Como é possível ajustar o cálculo de partições para compensar a distribuição irregular de objetos no espaço, mantendo o número de objetos em cada partição dentro da disponibilidade de memória, pode-se estimar que cada partição possui $n^A + r^B n^B / P$ objetos. Ao final, o desempenho do algoritmo é da ordem de

$$cpu_{SISJ} = O(c_{STT} + n^B \log slots + (n^A + r\,n^B)\log \frac{n^A + r^B n^B}{P}),$$

que pode ser reduzido à

$$cpu_{SISJ} = O(c_{STT} + (n^A + r\,n^B)\log \frac{n^A + r^B n^B}{P})$$

Este algoritmo combina a existência de uma árvore-R para um dos conjuntos com a idéia de particionamento do espaço para dividir a junção em tarefas mais simples. Também não constrói uma árvore-R para o conjunto não-indexado, mas é eficiente quanto ao número de operações de E/S e processamento.

# ANEXO B DESEMPENHO DO ALGORITMO DE LAÇOS ANINHADOS

O algoritmo de Laços Aninhados foi implementado de modo a alocar *M-1* blocos para o conjunto *A* e apenas um bloco para o conjunto *B*. Na primeira leitura de um bloco do conjunto *B*, este é ordenado e gravado. Nas leituras seguintes, o bloco já esta ordenado, reduzindo o processamento em memória. O número previsto de operações de E/S é função da cardinalidade dos conjunto e memória disponível.

$$disco_{LA} = |b^A| + \frac{|b^A|}{(M_b - 1)} \times |b^B|$$

O desempenho do algoritmo, quanto a processamento, é da ordem de

$$cpu_{LA} = O(c + \frac{((n^A + n^B).M_b)}{(M_b - 1)} \log f\, M_b)$$

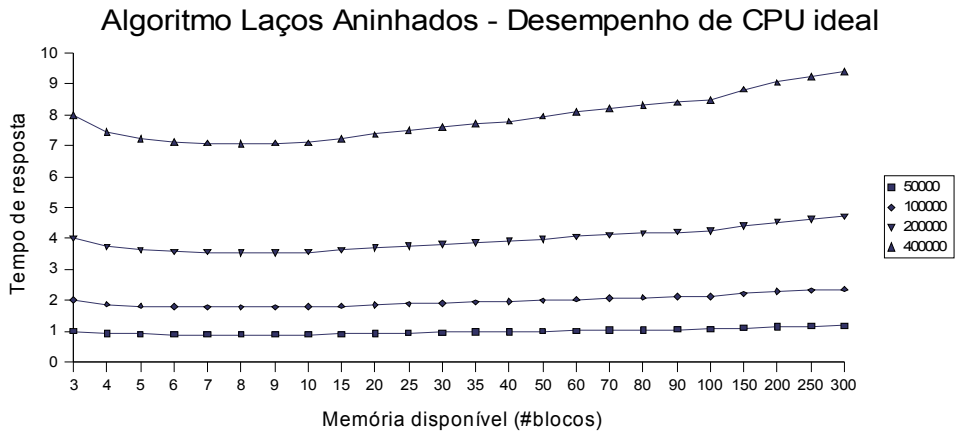



Figura B.1 : Gráfico de tempo de processamento previsto para o algoritmo de Laços Aninhados.

Um ponto interessante é que aumentar a memória disponível para *buffer* deve reduzir o número de operações de E/S, porém, o tempo de processamento apresenta uma curva com um ponto de inflexão, como mostra o gráfico na figura B.1. Nele está representada a estimativa de desempenho da junção espacial para quatro diferentes cardinalidades de pares de conjuntos, variando a memória disponível. Os resultados

estão normalizados, sendo 1 o valor obtido para a situação com um total de 50.000 objetos, dois conjuntos de 25.000 objetos, e memória disponível para apenas 3 blocos.

Conforme mostra o gráfico na figura B.2, este comportamento é confirmado com conjuntos reais de dados. Os menores tempo de resposta são obtidos utilizando a menor quantidade possível de memória. Com pouca memória, o tempo de processamento é mínimo, porém, o número de operações de E/S realizadas é muito alto, contribuindo com mais de 50% do tempo de resposta, como mostra a tabela B.1. Aumentando o tamanho do *buffer*, reduz o número de operações de E/S, até significarem menos de 1% do tempo de resposta.

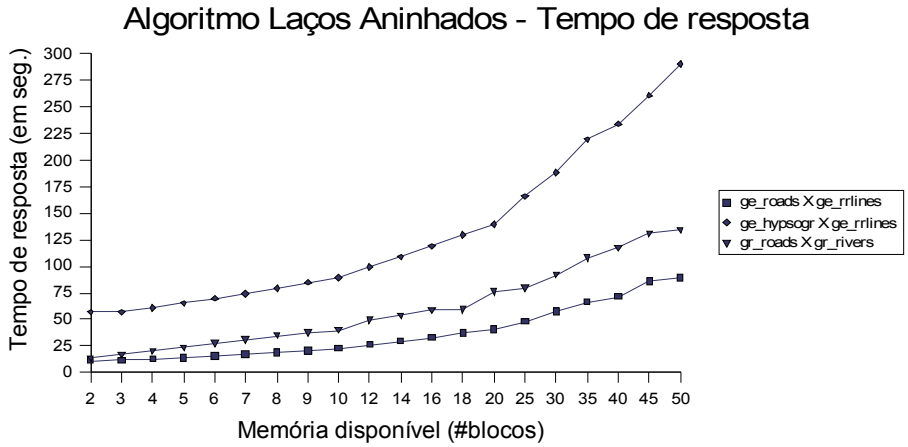



Figura B.2 : Gráfico do tempo de resposta do algoritmo de Laços Aninhados para três junções espaciais diferentes, variando a memória disponível.

Tabela B.1 : Proporção entre o tempo de operações de E/S e o tempo total de resposta para o algoritmo de Laços Aninhados.

| | | ***M=2*** | ***M=5*** | ***M=10*** | ***M=20*** | ***M=50*** |
|---|---|---|---|---|---|---|
| ge_roads X ge_rrlines | Tempo E/S | 6,88 | 2,84 | 1,49 | 0,81 | 0,36 |
| | Tempo total | 11,49 | 14,14 | 22,76 | 40,93 | 89,20 |
| | % tempo I/O | 59,87% | 20,05% | 6,53% | 1,99% | 0,41% |
| ge_hypsogr X ge_rrlines | Tempo E/S | 17,16 | 6,97 | 3,6 | 1,88 | 0,91 |
| | Tempo total | 56,77 | 65,22 | 89,22 | 139,3 | 290,12 |
| | % tempo I/O | 30,23% | 10,69% | 4,04% | 1,35% | 0,31% |

A figura B.3 mostra o tempo de resposta do algoritmo para duas junções diferentes, aplicando o predicado de distância entre objetos, e variando esta distância, em proporção ao universo. A memória disponível foi mantida constante. Assim, a variação no número de acessos a disco deveu-se exclusivamente ao aumento no conjunto resposta, que deve ser gravado em disco. O tempo de processamento também aumenta, pois aumenta o número de pares de objetos que devem ser testados para verificar o predicado espacial.

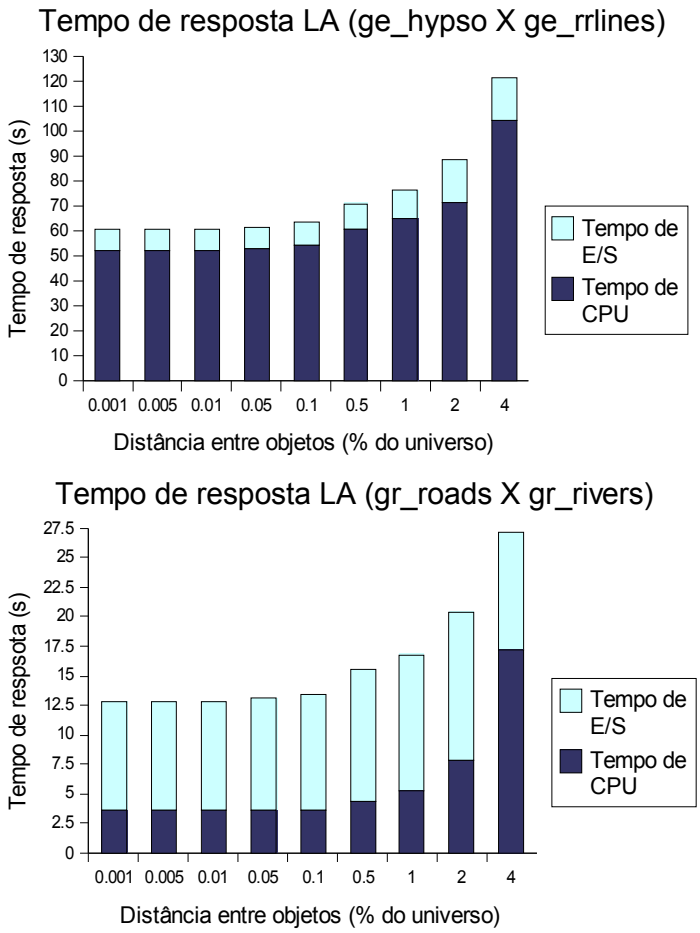



Figura B.3 : Gráfico do tempo de resposta, variando a distância mínima entre objetos como predicado espacial.

O problema do algoritmo é que o tempo de resposta é muito alto, o que dificultou testes com conjuntos reais de maior cardinalidade. Em outros algoritmos, os objetos são, de alguma forma, organizados por um critério espacial, enquanto neste algoritmo, não há nenhuma previsão, resultando um número de comparações entre objetos muito alto.