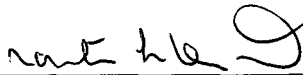


TÉCNICAS PARA ALOCAÇÃO DE FRAGMENTOS EM PROJETO DE
BANCOS DE DADOS DISTRIBUÍDOS

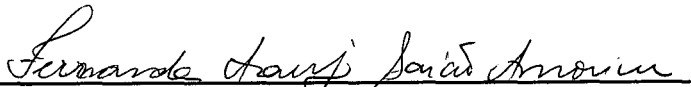
Matheus Wildemberg Souza

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS
PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA
UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS
REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE
EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

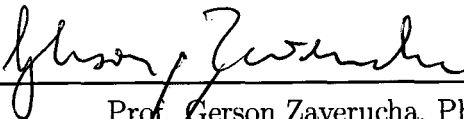
Aprovada por:



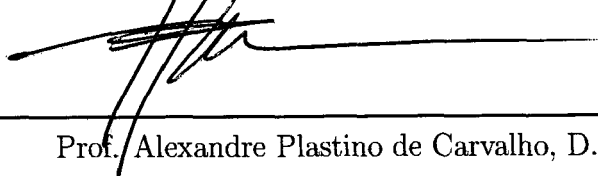
Prof. Marta Lima de Queirós Mattoso, D.Sc.



Prof. Fernanda Araújo Baião Amorim, D.Sc.



Prof. Gerson Zaverucha, Ph.D.



Prof. Alexandre Plastino de Carvalho, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

AGOSTO DE 2004

SOUZA, MATHEUS WILDEMBERG

Técnicas para Alocação de Fragmentos em
Projeto de Bancos de Dados Distribuídos [Rio
de Janeiro] 2004

XIV, 134p. 29,7 cm (COPPE/UFRJ,
M.Sc., Engenharia de Sistemas e Computação,
2004)

Tese - Universidade Federal do Rio de
Janeiro, COPPE

1. Projeto de Bancos de Dados Distribuídos
2. Fragmentação
3. Alocação de dados

I. COPPE/UFRJ II. Título(série)

Dedicatória

Aos meus pais, Eliezer e Suely e tios, Rosangela e Marcos por considerá-los os principais responsáveis por ter alcançado esta conquista.

Agradecimentos

A Deus que tudo nos concedeu.

À minha orientadora Marta Mattoso que com muita paciência repassou muito do seu conhecimento auxiliando o desenvolvimento deste trabalho de forma criteriosa.

À minha co-orientadora Fernanda Baião pela atenção e os conselhos e incentivos que permitiram a conclusão do trabalho. Pelos ensinamentos passados durante as reuniões. Às minhas orientadoras agradeço pela confiança depositada em mim.

À Melise que foi essencial na realização deste trabalho. Pelas ajudas nas idéias que deram origem a esta dissertação. Pelas correções e insistência durante todo o tempo.

A todos os meus colegas de mestrado que foram importantes na minha adaptação à uma nova vida em uma nova cidade.

À "comunidade mineira no Rio de Janeiro". Em especial a Rodrigo, Melba e Lúcio que desde a graduação fazem parte do meu grupo de amigos sempre me apoiando. Também a Bernardo por ter me ajudado com os problemas finais com a dissertação. As ajudas foram muito importantes.

A todas as pessoas que conheci no Rio de Janeiro durante meu mestrado. Em especial a Kate, que me aconselhou muito durante o desenvolvimento dessa dissertação,

ajudando a desvendar os segredos do latex.

Aos meus tios Elder, Eduardo, Karen, Vera e Márcia que sempre se preocupam e nos manter unidos em todos os momentos. Aos primos, com quem sempre pude contar.

Aos meus irmãos Rafael e Júnior que compartilharam comigo as melhores fases da vida. Principalmente a Rafael que é e sempre será meu melhor amigo.

Aos meus padrinhos, Marcos Wildemberg e Rosangela Wildemberg, que me ajudaram durante minha educação. Pelo apoio e incentivo constante em todas as etapas da minha vida. Pelo respeito e amor que tem por mim. Vocês são essenciais na minha vida.

Aos meus pais, Eliezer e Suely que se dedicaram muito para me ajudar a buscar meus objetivos. Ao apoio em todas as etapas da minha vida e a confiança depositada em mim. Por todo o sacrifício passado para que fosse possível a minha formação.

À Capes pelo suporte financeiro.

A todos aqueles que de alguma forma contribuíram para a realização deste trabalho.

Obrigado.

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

TÉCNICAS PARA ALOCAÇÃO DE FRAGMENTOS EM PROJETO DE BANCOS DE DADOS DISTRIBUÍDOS

Matheus Wildemberg Souza

Agosto/2004

Orientadoras: Marta Lima de Queirós Mattoso
Fernanda Araujo Baião Amorim

Programa: Engenharia de Sistemas e Computação

Um projeto de distribuição de bancos de dados visa obter fragmentos da base de dados e alocá-los dentre os nós do sistema distribuído. O objetivo do projeto de alocação é associar fragmentos aos pontos de acesso relevantes e com isso tornar o custo de execução o menor possível. Para isso o projeto de alocação também pode sugerir o uso de réplicas. Devido às dificuldades em modelar o custo de execução e o enorme espaço de soluções, este é um problema caracterizado como NP-Completo. Esta dissertação propõe dois algoritmos, *Aloc* e *GRADA*, para alocação de fragmentos em sistemas de banco de dados distribuídos. *Aloc* é um algoritmo heurístico baseado num algoritmo eficiente da literatura com mesma ordem de complexidade, enquanto *GRADA* é um algoritmo baseado na meta-heurística GRASP. Foram executados diversos experimentos, que se utilizaram de um modelo de custo validado experimentalmente com aplicações de pequeno e grande porte (onde utilizamos a especificação do *benchmark* TPC-C). Nos experimentos realizados identificaram-se situações em que o algoritmo *Aloc* conseguiu atingir a solução ótima ou reduzir o custo final do esquema de alocação em relação ao algoritmo da literatura, especialmente em cenários de atualização intensiva. O algoritmo *GRADA* apresentou resultados melhores que o algoritmo *Aloc* em situações com maior número de nós na rede e/ou uma quantidade menor de transações de atualização, obtendo resultados semelhantes aos resultados alcançados pelo algoritmo *Aloc* nos demais casos.

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

FRAGMENT ALLOCATION TECHNIQUES IN DISTRIBUTED DATABASE DESIGN

Matheus Wildemberg Souza

August/2004

Advisors: Marta Lima de Queirós Mattoso
Fernanda Araujo Baião Amorim

Department: Systems Engineering and Computer Science

The distributed database design aims at obtaining fragments of the database and allocating fragments among the sites of the distributed system. The goal of the distributed database design is to associate fragments to the most adequate network nodes, thus minimizing the application execution cost. The fragment allocation problem also includes decisions about fragment replication. Due to the number of parameters for the problem, and to the number of possible solutions in the search space, the allocation problem is considered in the literature as a NP-Complete problem. In this work, two algorithms are proposed for fragment allocation in distributed database systems: *Aloc* and *GRADA*. *Aloc* is a heuristic algorithm based on another algorithm from the literature that produced good results. *Aloc* maintains the complexity of the original algorithm. *GRADA* is an algorithm based on the GRASP meta-heuristic. Through simulations performed on top of the TPC-C benchmark, it was possible to identify scenarios where the *Aloc* algorithm found the optimal solution and other scenarios where *Aloc* found allocation schema with a reduced cost when compared to the original algorithm. The *GRADA* algorithm obtained better results than the *Aloc* algorithm in scenarios with a large number of sites (where the number of alternative solutions considered is very large) and in scenarios with lower update transaction rates. In other cases, the *GRADA* algorithm obtained the same results as the *Aloc* algorithm.

SUMÁRIO

Dedicatória	iii
Agradecimentos	iv
Lista de Figuras	xi
Lista de Tabelas	xiii
1 Introdução	1
2 Alocação de dados em SBDD	6
2.1 Projeto de Distribuição	7
2.2 Alocação de Fragmentos	11
2.3 Modelagem do problema de alocação de Fragmentos	12
2.3.1 Geração das Informações de Entrada	12
2.3.2 Geração do Conjunto de Restrições	14
2.3.3 Geração do Espaço de Soluções	14
2.3.4 Método de Busca	15
2.3.5 Função de custo [41]	15
2.4 Trabalhos Relacionados	18
3 Proposta do Algoritmo de Alocação Heurístico <i>Aloc</i>	26
3.1 Função de Custo	27
3.2 Algoritmo <i>Huang</i>	30
3.2.1 Primeiro passo do algoritmo <i>Huang</i>	31
3.2.2 Segundo passo do algoritmo <i>Huang</i>	32
3.2.3 Terceiro passo do algoritmo <i>Huang</i>	35

3.3	Algoritmo <i>Aloc</i>	36
3.3.1	Primeiro passo do algoritmo <i>Aloc</i>	38
3.3.2	Segundo passo do algoritmo <i>Aloc</i>	39
3.3.3	Terceiro passo do algoritmo <i>Aloc</i>	40
3.4	Análise comparativa dos algoritmos	42
4	Avaliação do <i>Aloc</i>	46
4.1	Experimentos com alto volume de transações	46
4.2	Experimentos segundo <i>benchmark</i> TPC-C	49
4.3	Experimentos com atualização intensa	55
5	Proposta do Algoritmo de Alocação Meta-Heurístico	57
5.1	Meta-Heurística	57
5.2	GRASP - Procedimento de Busca Adaptativa Aleatória Gulosa	58
5.2.1	Princípios Básicos	58
5.2.2	Fase de Construção do GRASP	59
5.2.3	Fase de Busca Local do GRASP	61
5.3	Algoritmo <i>GRADA</i> - Greedy RAndomized Data Allocation	63
5.3.1	Fase de Construção do <i>GRADA</i>	64
5.3.2	Fase de Busca Local do <i>GRADA</i>	66
6	Avaliação do <i>GRADA</i>	69
6.1	Experimentos com alto volume de transações	71
6.2	Experimentos com baixo volume de transações	72
6.3	Experimentos com atualização intensa	76
7	Conclusão	78
	Referências Bibliográficas	83
A	<i>XAloc</i>: Uma ferramenta para avaliar algoritmos de alocação de dados	88
A.1	Componentes da ferramenta <i>XAloc</i>	88
A.2	Preparação dos Dados	89
A.3	Algoritmos	90
A.4	Função de custo	91
A.5	Funcionalidades	92
B	Implementação das funções do algoritmo de alocação por uma classe abstrata	93

C Implementação do algoritmo Huang pela Classe THuang	110
D Implementação do algoritmo Aloc pela Classe TAlloc	114
E Implementação do algoritmo GRADA pela Classe TGRADA	118
F Implementação da Classe TSol_Otima para o algoritmo de busca exaustiva	127

LISTA DE FIGURAS

2.1	Processo de projeto de um banco de dados [41]	8
2.2	Modelo Geral para o Problema de Alocação de Fragmentos	13
3.1	Algoritmo <i>Huang</i> - Passo 1	32
3.2	Função benefício da retirada de uma réplica	33
3.3	Função custo da retirada de uma réplica	34
3.4	Algoritmo <i>Huang</i> - Passo 2	34
3.5	Pseudo-código do algoritmo <i>Huang</i> [26]	37
3.6	Pseudo-código da função atraso	38
3.7	Algoritmo <i>Huang</i> - Passo 3	38
3.8	Modelo para PAF segundo algoritmo <i>Huang</i>	39
3.9	Algoritmo <i>Aloc</i> - Passo 1	40
3.10	Algoritmo <i>Aloc</i> - Passo 2	41
3.11	Algoritmo <i>Aloc</i> - Passo 3	42
3.12	Pseudo-código do algoritmo <i>Aloc</i>	43
3.13	Pseudo-código da função LRO (Lista de Réplicas Ordenadas)	44
3.14	Modelo para PAF segundo algoritmo <i>Aloc</i>	44
4.1	Resultados dos algoritmos de solução ótima, <i>Aloc</i> e <i>Huang</i> em cenários com alto volume de transações (4 nós, 4 transações e 4 fragmentos)	47
4.2	Resultados dos algoritmos de solução ótima, <i>Aloc</i> e <i>Huang</i> em cenários com alto volume de transações (4 nós, 5 transações e 4 fragmentos)	48
4.3	Resultados dos algoritmos de solução ótima, <i>Aloc</i> e <i>Huang</i> em cenários com alto volume de transações (4 nós, 4 transações e 5 fragmentos)	48
4.4	Resultados dos algoritmos de solução ótima, <i>Aloc</i> e <i>Huang</i> em cenários com alto volume de transações (5 nós, 4 transações e 4 fragmentos)	48
4.5	DER do <i>benchmark</i> TPC-C	50

4.6	Resultados do <i>benchmark</i> TPCC com os algoritmos <i>Aloc</i> e <i>Huang</i> (4 nós, 8 transações e 17 fragmentos)	53
4.7	Resultados do <i>benchmark</i> TPCC com os algoritmos <i>Aloc</i> e <i>Huang</i> (8 nós, 8 transações e 17 fragmentos)	53
4.8	Resultados do <i>benchmark</i> TPCC com os algoritmos <i>Aloc</i> e <i>Huang</i> (12 nós, 8 transações e 17 fragmentos)	54
4.9	Custos de operações de leitura do <i>benchmark</i> TPC-C	54
4.10	Custos de operações de atualização do <i>benchmark</i> TPC-C	54
5.1	Pseudo-código do algoritmo básico GRASP	59
5.2	Pseudo-código para Fase de Construção do GRASP	60
5.3	Pseudo-código para Fase de Busca Local do GRASP	62
5.4	Pseudo-código do algoritmo <i>GRADA</i>	64
5.5	Pseudo-código da Fase de Construção do <i>GRADA</i>	66
5.6	Pseudo-código da função LRCR (Lista de Réplicas Candidatas)	66
5.7	Pseudo-código da Fase de Busca Local do <i>GRADA</i>	67
5.8	Modelo para PAF segundo algoritmo <i>GRADA</i>	67
5.9	Pseudo-código da função de re-alocação da Fase de Busca Local do <i>GRADA</i>	68
6.1	Pseudo-código da função de definição de k	70
6.2	Pseudo-código da função de definição de k fixo	71
6.3	Pseudo-código da função de definição de k baseado na LRCR	71
6.4	Resultado de experimentos com os algoritmos <i>Aloc</i> e <i>GRADA</i> com 10 transações em cenários com baixo volume de transações	73
6.5	Resultado de experimentos com os algoritmos <i>Aloc</i> e <i>GRADA</i> com 20 transações em cenários com baixo volume de transações	73
6.6	Resultado de experimentos com os algoritmos <i>Aloc</i> e <i>GRADA</i> com 30 transações em cenários com baixo volume de transações	74
6.7	Resultado de experimentos com os algoritmos <i>Aloc</i> e <i>GRADA</i> com 40 transações em cenários com baixo volume de transações	74
6.8	Resultado de experimentos com os algoritmos <i>Aloc</i> e <i>GRADA</i> com 60 transações em cenários com baixo volume de transações	75
6.9	Resultado de experimentos com os algoritmos <i>Aloc</i> e <i>GRADA</i> com 80 transações em cenários com baixo volume de transações	75
6.10	Cenário que favorece ao algoritmo <i>GRADA</i>	76
A.1	Dados de entrada	89
A.2	Características do <i>XAloc</i>	90

LISTA DE TABELAS

2.1	Comparação dos trabalhos segundo os itens analisados.	23
4.1	Fragmentação das tabelas do TPC-C.	51
4.2	Resultados de experimentos baseados no <i>benchmark</i> TPC-C.	52
4.3	Desempenho do <i>Aloc</i> com atualização intensa.	56
6.1	Desempenho do <i>GRADA</i> com transações intensas.	72
6.2	Desempenho do <i>GRADA</i> com número fixo de nós.	72
6.3	Desempenho do <i>GRADA</i> com atualização intensa.	76

Capítulo 1

Introdução

Sistemas de bancos de dados que manipulam um grande volume de dados têm se tornado bastante freqüentes fazendo com que aplicações que utilizam esses sistemas se tornem cada vez mais lentas. Para evitar isto, são necessárias técnicas adequadas à manipulação de grandes massas de dados. Dentre essas técnicas surgem os Bancos de Dados Distribuídos (BDD). Um BDD consiste de uma coleção de vários bancos de dados logicamente inter-relacionados e distribuídos por uma rede de computadores. Estes BDD necessitam de um Sistema de Gerenciamento de Bancos de Dados Distribuídos (SGDBD) que permita o gerenciamento do BDD de forma transparente ao usuário. A expressão Sistema de Bancos de Dados Distribuídos (SBDD) é utilizada para representar o conjunto do BDD e o SGDBD [41] que é utilizado como solução para a manipulação de grande volume de dados.

Uma das principais vantagens da utilização de SBDD é o aumento de desempenho das aplicações. Esse aumento de desempenho pode ser alcançado através da distribuição dos dados sobre os nós da rede. Isto possibilita o paralelismo e o aumento de proximidade dos dados ao ponto de acesso, ou seja, fazer com que aumente os acessos locais das aplicações aos dados.

Porém, para alcançar as vantagens do SBDD, é necessário que a distribuição dos dados seja realizada de forma adequada, o que pode ser alcançado através da realização do Projeto de Distribuição de Bancos de Dados (PDBD) [41] que decide como os dados acessados pelas aplicações devem ser distribuídos pelos nós da rede.

Neste processo de decisão, o PDBD define fragmentos da base de dados e decide onde estes fragmentos devem ser alocados. Uma das abordagens de execução do PDBD, denominada abordagem descendente, divide-o em duas fases, que são a fragmentação e a alocação.

A fase de **fragmentação** do projeto de distribuição busca evitar que aplicações acessem dados irrelevantes às suas transações. Estes acessos acontecem porque, muitas vezes, a aplicação acessa toda a tabela para obter um sub-conjunto desta tabela que de fato é seu interesse. Assim, a fase de fragmentação visa encontrar esses sub-conjuntos e definir a forma como cada tabela pode ser dividida em unidades menores, formando os fragmentos.

Uma vez definidos os fragmentos é realizada a fase de **alocação**. Nesta fase é definido o número de réplicas de cada fragmento e os nós da rede onde estas réplicas serão armazenadas, com o objetivo de aumentar a proximidade dos fragmentos aos pontos de acesso e conseqüentemente a eficiência das consultas. Nesta fase, portanto, é decidida a necessidade de replicação de dados, ou seja, se será alocada somente uma réplica de cada fragmento ou se cada fragmento pode aparecer alocado em mais de um nó da rede. A replicação, quando tratada, aumenta a confiabilidade do sistema.

O Problema de Alocação de Fragmentos (PAF) em um SBDD é um aspecto crítico já que uma alocação tem impacto direto no custo de acesso das aplicações aos dados. De uma forma geral, este problema pode ser definido como: dado um conjunto S de nós da rede, um conjunto T de transações que são executadas a partir de S e F um conjunto de fragmentos definidos na fase de fragmentação, busca-se encontrar um esquema de alocação de F em S de forma que o custo de execução de T seja mínimo. Este esquema de alocação deve considerar replicação de cada fragmento nos nós da rede. Além disso, um conjunto de restrições deve ser considerado na definição desse esquema de alocação, como por exemplo a capacidade de armazenamento e processamento dos nós onde os dados são alocados. Este tipo de problema é classificado na literatura como Problema de Otimização Combinatória.

Em [3], o autor demonstra que o problema de encontrar uma alocação de fragmentos ótima em um SBDD é NP-Completo. Segundo Huang, Chen [26], dado

um problema com n fragmentos e m nós, $(2^m - 1)^n$ é o número de soluções possíveis para este problema, incluindo as soluções com replicação de fragmentos. Apesar da existência de um número finito de soluções, a avaliação de cada solução para escolha da melhor é muito custosa. Por conta disso, justifica-se a utilização de métodos heurísticos para solucionar o problema, na maioria dos trabalhos existentes na literatura, pois é inviável solucionar esse tipo de problema com um algoritmo exato em função da sua complexidade.

Existem propostas na literatura [2] para resolver o problema de alocação de dados em sistemas distribuídos de um modo geral, porém estes trabalhos consideram a alocação de arquivos individuais. A alocação de fragmentos em SBDD envolve questões específicas ao problema que dificultam a utilização destas soluções. No problema de alocação de fragmentos em SBDD, estes não podem ser considerados como unidades isoladas, devido à existência de relacionamentos entre os fragmentos. Assim, num SBDD é preciso tirar proveito das transações mais frequentes, alocando os diversos fragmentos acessados pela mesma transação preferencialmente no mesmo nó.

Em alguns trabalhos existentes na literatura, [6], [23], os autores propõem uma simplificação para o problema de alocação desconsiderando a decisão de replicação de fragmentos. Outros trabalhos, [24], [5], propõem uma solução para o problema que trata um modelo de dados específico, tornando a solução restrita. Dentre os trabalhos analisados, somente [26] considera replicação de dados na solução final e não se limita a um modelo de dados específico. Porém a solução apresentada por estes autores limita o espaço de soluções enfatizando transações de leitura. Além disso, nenhum dos trabalhos analisados considera experimentos com dados que represente um problema real. Todos os experimentos propõem cenários de pequenas dimensões.

A proposta desta dissertação é apresentar uma solução para o problema de alocação de fragmentos em SBDD que considere replicação, não seja restrito a um modelo de dados específico, e que seja aplicável em cenários representativos de aplicações reais. Em nossa proposta apresentamos um algoritmo heurístico, o *Aloc*, e um algoritmo meta-heurístico, o *GRADA*.

Aloc é um algoritmo heurístico baseado no algoritmo proposto por Huang e Chen [26]. A principal característica do *Aloc* é ser simples, de fácil compreensão e fácil aplicação, com mesma ordem de complexidade do algoritmo Huang e Chen $O(nm^2q)$, onde n é o número de fragmentos distintos, m é o número de nós e q é o número de transações. Através de experimentos realizados, o algoritmo *Aloc* alcançou resultados próximos da solução ótima e melhores que os algoritmos existentes na literatura.

Já o algoritmo *GRADA* busca aprimorar o algoritmo *Aloc* com a utilização de uma meta-heurística no seu método de solução, visando aumentar o espaço de soluções analisadas. Isso permite que o número de soluções analisadas pelo algoritmo *GRADA* seja maior que no algoritmo *Aloc* permitindo ao algoritmo *GRADA* convergir para soluções melhores.

Os algoritmos propostos, além de um algoritmo de busca exaustiva que encontra a solução ótima para o PAF, foram implementados em uma ferramenta denominada XAloc [33], que permite a comparação entre os resultados dos algoritmos de acordo com três funções de custo distintas. Foram realizados experimentos que simularam a execução dos algoritmos em cenários com um número reduzido de transações, cenários com transações de leitura e atualização intensa, e em cenários construídos a partir da especificação do *benchmark* TPC-C [34]. Estes experimentos comprovam que os resultados alcançados pelos algoritmos *Aloc* e *GRADA* são melhores que os resultados apresentados pelo algoritmo proposto por Huang e Chen [26] em todos os cenários e no pior caso os algoritmos propostos obtêm o mesmo resultado.

O restante desta dissertação está dividido da seguinte maneira. No Capítulo 2 é definido o problema de alocação e suas principais características e é feita uma análise qualitativa de trabalhos existentes na literatura que propõem uma solução para o problema de alocação de dados. No Capítulo 3 é apresentado o algoritmo *Aloc* e o algoritmo proposto por Huang e Chen [26], que possui os fundamentos adotados na criação do *Aloc*. Ainda no Capítulo 3, é descrita a função de custo utilizada pelos algoritmos para definir o custo de execução das aplicações sobre o esquema de alocação resultante. A seguir, no Capítulo 4 são mostrados os resultados

dos experimentos realizados com os algoritmos apresentados no Capítulo 3, onde mostramos que o algoritmo *Aloc* obteve ganhos em relação ao algoritmo proposto por Huang e Chen [2]. No Capítulo 5 é apresentada a proposta da utilização da meta-heurística GRASP [40] através do algoritmo *GRADA*. Neste capítulo é definida a meta-heurística utilizada e a maneira como foi aplicada no método de solução do problema de alocação. No Capítulo 6 são mostrados os resultados de alguns experimentos realizados com os algoritmos *Aloc* e *GRADA* definindo os cenários onde os algoritmos propostos são mais eficientes. E finalmente no Capítulo 7 é mostrada a conclusão desta dissertação e propostas de trabalhos futuros. O Apêndice A descreve as características da ferramenta utilizada na geração dos resultados experimentais com os algoritmos propostos. Os apêndices seguintes descrevem a implementação dos algoritmos.

Capítulo 2

Alocação de dados em SBDD

A utilização do SBDD surge como uma solução para aumento do desempenho de aplicações que manipulam grande volume de dados. Um SBDD utiliza um Banco de Dados Distribuído (BDD) consistindo de vários bancos de dados logicamente integrados em uma rede de computadores. Estes bancos de dados são gerenciados, de forma transparente para o usuário, por um Sistema de Gerenciamento de Bancos de Dados Distribuídos (SGDBD). Os principais objetivos de um (SBDD) são:

1. Aumento do desempenho do sistema distribuído em relação ao centralizado, evitando que as aplicações acessem dados irrelevantes as suas transações. Além disso é possível tirar proveito dos recursos distribuídos na execução das transações. Os fatores que contribuem para isso são:
 - (a) Num ambiente distribuído, os vários pontos de armazenamento de dados podem armazenar partições da base, diminuindo o custo de armazenamento local e otimizando o acesso à memória;
 - (b) Com a base distribuída, a alocação pode aproximar as partições de seus pontos de acesso provendo o que se chama de "localidade de acesso";
 - (c) O impacto do custo de acesso remoto aos dados no custo total das transações pode ser reduzido a partir de uma execução paralela de consultas.
2. Aumento da confiabilidade do sistema, eliminando os pontos únicos de falha;

3. Transparência na gerência dos dados distribuídos e replicados permitindo, por exemplo, que as consultas disparadas pelo usuário sejam executadas sem que este tome consciência da forma como os dados estão distribuídos na rede;
4. Facilidade de expansão facilitando o armazenamento de frações crescentes do banco de dados e economia com equipamentos para armazenamento dos dados. Em SBDD é possível utilizar equipamentos menores com capacidades equivalentes a um único equipamento de grande porte.

Para garantir que estes objetivos sejam alcançados, o Projeto de Distribuição de Bancos de Dados (PDBD) é fundamental pois determina a distribuição adequada dos dados. O PDBD é responsável por definir como a base de dados deve ser fragmentada e onde esses fragmentos devem ser armazenados.

2.1 Projeto de Distribuição

Existem duas abordagens para o projeto de distribuição em banco de dados distribuídos [7]: a abordagem ascendente (*bottom-up*) e a abordagem descendente (*top-down*).

A abordagem ascendente é adequada a sistemas ligados a ambientes onde já existam bancos de dados em produção. Nestes casos, o projeto de distribuição busca a integração destes dados de forma a disponibilizar uma visão única e global dos dados, como ocorre em bancos de dados heterogêneos.

A abordagem descendente é utilizada em sistemas onde o banco de dados está sendo projetado desde o início. Em [41] os autores ilustram a abordagem descendente para o Projeto de Distribuição como uma atividade ao longo do processo no projeto de um banco de dados. A abordagem descendente é mostrada na Figura 2.1.

O projeto de uma base de dados se inicia com a análise de requisitos onde se busca levantar as necessidades para o sistema segundo seus usuários em potencial. O resultado da atividade de análise de requisitos é o documento de requisitos do sistema, que é a entrada para o projeto conceitual e projeto de visões.

Até o momento, o projeto de um sistema distribuído é idêntico a um sistema centralizado. O ponto que diferencia o projeto de um sistema distribuído é a ati-

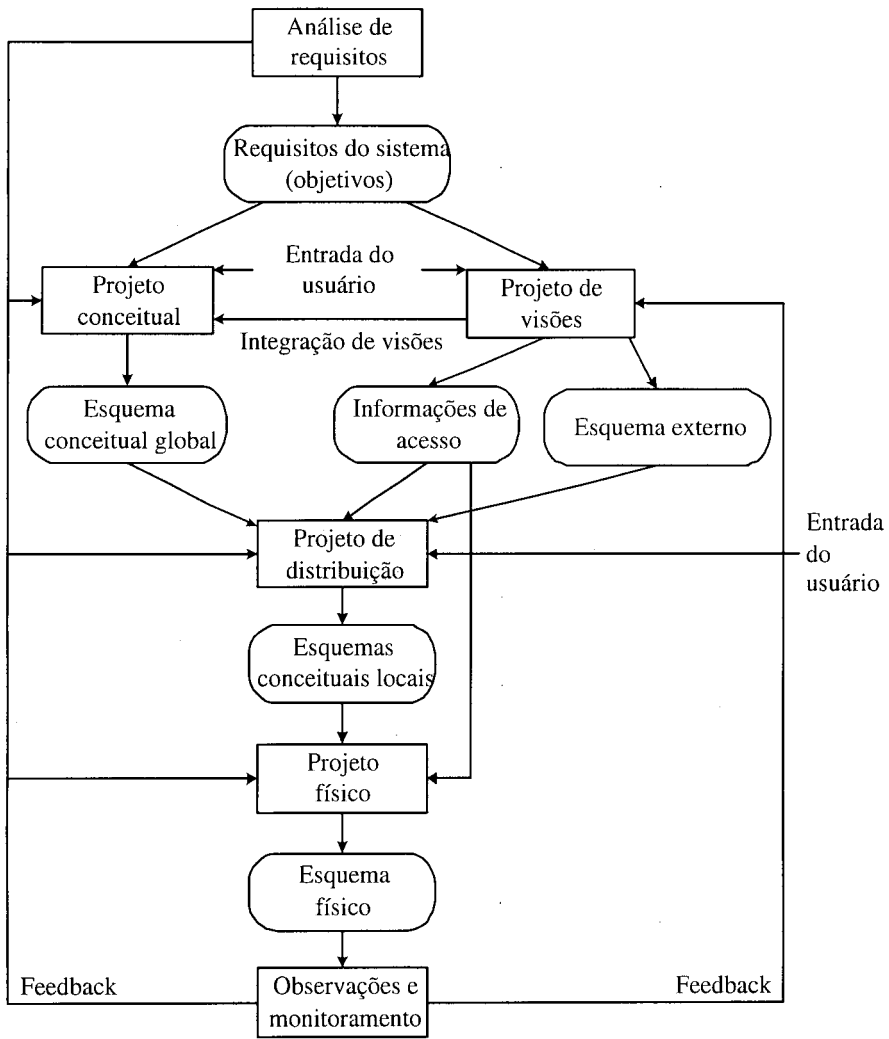


Figura 2.1: Processo de projeto de um banco de dados [41]

vidade do projeto de distribuição. As entradas para essa atividade são o esquema conceitual global, resultante do projeto conceitual, as informações de acesso e esquema externo, resultantes do projeto de visões, e as entradas do usuário. O objetivo desta atividade é produzir os esquemas conceituais locais distribuindo as entidades em fragmentos de acordo com o padrão de acesso dos nós aos dados da base, que serão alocados aos nós do sistema distribuído.

A última atividade do projeto descendente é o projeto físico. Nesta atividade é feito o mapeamento dos esquemas conceituais locais, resultantes do projeto de distribuição, nos dispositivos de armazenamento físico disponíveis em cada nó do sistema. A atividade de monitoramento permite ajustes e aprimoramento do sistema

tanto do ponto de vista da implementação do banco de dados quanto da adequação das visões do usuário.

A atividade de projeto de distribuição, seguindo a abordagem descendente, é dividida em duas fases: fragmentação e alocação. Existem trabalhos que buscam a realização da fragmentação e alocação em um único algoritmo [8], [9] e [10]. Entretanto, devido à sua complexidade, muitas vezes a fase de alocação é tratada de forma separada da fase de fragmentação. Neste caso, a saída da fase de fragmentação é a entrada da fase de alocação. Ainda assim, as duas fases do projeto de distribuição são consideradas problemas NP-Completo, mesmo se tratadas de forma independente [41].

O objetivo da fragmentação é dividir cada tabela de um banco de dados em fragmentos menores de modo a diminuir o volume de dados que é acessado pelas aplicações e que trafegam na rede. A fragmentação busca encontrar a unidade apropriada de distribuição. Uma tabela, na maioria das vezes, não é a unidade apropriada, pois as aplicações acessam visões que são compostas por subconjuntos de tabelas. Assim, o objetivo é que apenas esses subconjuntos sejam acessados. Para isso, na fase de fragmentação, para cada tabela são agrupados os itens de dados em fragmentos de acordo com informações de acesso sobre cada tabela da base de dados.

Existem dois tipos básicos de fragmentação: fragmentação horizontal e fragmentação vertical. A fragmentação horizontal divide as tabelas em subconjuntos de tuplas gerando fragmentos que se diferem por conteúdo, enquanto que a fragmentação vertical divide as tabelas por atributos gerando fragmentos com estruturas diferentes. Além dos tipos básicos, a fragmentação híbrida é a aplicação da fragmentação horizontal e vertical em uma mesma tabela.

A fragmentação horizontal pode ser subdividida em: *fragmentação horizontal primária*, onde a fragmentação é executada através de critérios definidos sobre a própria tabela e *fragmentação horizontal derivada*, onde os fragmentos da tabela são definidos em função do relacionamento da tabela com outra tabela que faça o papel de dono do relacionamento. A fragmentação horizontal primária visa favorecer as operações de seleção, isolando as tuplas que satisfazem os critérios de seleção mais frequentes. Já a fragmentação horizontal derivada visa favorecer a execução das

operações de junção entre uma tabela que tenha sofrido fragmentação horizontal primária (FHP) e uma tabela que tenha sofrido fragmentação horizontal derivada (FHD).

No projeto de distribuição de uma tabela R , a fragmentação deve atender a três critérios de correção:

1. *Compleitude*: garante que todo item de dado de R deve existir em um fragmento dessa tabela, ou seja, para todo dado previsto em R deverá existir um fragmento cuja definição abranja esse dado.
2. *Reconstrução*: garante que se a tabela R é decomposta no conjunto de fragmentos R_1, R_2, \dots, R_n existe um operador relacional ∇ onde $R = \nabla R_i$, $1 \leq i \leq n$, ou seja, ∇ reconstrói R .
3. *Disjunção*: em fragmentação horizontal garante que uma tupla da tabela R pertença a somente um dos fragmentos resultantes, em fragmentação vertical garante que um atributo da tabela original pertença a somente um dos fragmentos resultantes, exceto para chave primária.

A fragmentação também se aplica ao modelo orientado a objetos com regras correspondentes. Em [35] os autores propõem uma definição para fragmentação horizontal, vertical e híbrida para o modelo orientado a objetos.

Existem diversos trabalhos na literatura que abordam o tópico de fragmentação em projeto de distribuição. Para maiores detalhes, informações podem ser encontradas em [12], [20] e [6] onde é abordada a fragmentação em projeto de distribuição em banco de dados relacionais, enquanto que [13], [14], [15], [17], [16], [18], [11] e [35] endereçam o problema de fragmentação em banco de dados orientado a objetos.

A segunda fase do projeto de distribuição, a fase de alocação, define a localização dos fragmentos definidos na fase anterior nos nós da rede e decide quanto à replicação destes fragmentos. Segundo Özsü e Valdúriez [41], existem três alternativas de replicação em banco de dados: um banco de dados *sem replicação* possui cada fragmento alocado em um e somente um nó da rede. Um banco de dados replicado pode estar *totalmente replicado* e possuir todos os fragmentos alocados em cada nó

da rede, ou *parcialmente replicado* com os fragmentos distribuídos nos nós da rede, podendo existir cópias de alguns fragmentos em mais de um nó da rede.

A replicação busca manter os fragmentos próximos ao ponto de acesso. Isto beneficia as transações de leitura, pois reduz o custo de comunicação permitindo que o acesso dos nós aos fragmentos seja local. Por outro lado, o custo da replicação pode ser alto se consideradas as transações de atualização. Com o maior número de cópias de fragmentos, aumenta a dificuldade de garantir a consistência dos dados com as transações de atualização. Assim, a replicação parcial tenta aumentar o desempenho do sistema ao mesmo tempo em que tenta reduzir a sobrecarga de trabalho para a manutenção de consistência entre as réplicas.

2.2 Alocação de Fragmentos

O problema da alocação de fragmentos em um banco de dados distribuídos tem como objetivo encontrar a distribuição dos fragmentos nos nós de forma a minimizar o custo da execução das transações. Assim, seja $S = \{s_1, s_2, \dots, s_m\}$ o conjunto de nós, onde m é o número de nós da rede, $Q = \{q_1, q_2, \dots, q_q\}$ o conjunto de transações que são executados em S , onde q é o número de transações, e $F = \{f_1, f_2, \dots, f_n\}$ o conjunto de fragmentos resultantes da fase de fragmentação do projeto de distribuição, onde n é o número de fragmentos de todas as tabelas, o somatório dos custos para a execução de cada q_k a partir de s_j sobre os diversos fragmentos envolvidos em q_k deve ser o menor possível.

O problema de alocação de fragmentos em sistemas de banco de dados distribuídos não pode ser visto como um simples problema de alocação de dados por apresentar características intrínsecas. Os dados em um sistema distribuído são independentes entre si, enquanto que os fragmentos apresentam relacionamentos uns com os outros, o que torna mais complexas a sua alocação, gerência e recuperação através do processamento de consultas. Assim, a alocação de um fragmento tem impacto na alocação de outros fragmentos. Em sistemas de banco de dados distribuídos o acesso a dados inclui o processamento de consulta (que pode reduzir o volume de dados) e o relacionamento entre os fragmentos (que pode combinar dados). Outra questão que deve ser considerada é a integridade referencial que deve ser mantida

entre os fragmentos e o controle de concorrência que devem ser considerados no custo da alocação de fragmentos.

Além disso, o problema de alocação é um problema NP-Completo [3], ou seja, não é viável a utilização de um algoritmo exato para solução do problema em tempo razoável devido à sua complexidade. Segundo Huang e Chen [26] $(2^m - 1)^n$ é o número de combinações possíveis como solução para um problema com n fragmentos e m nós. Apesar da existência de um número finito de soluções, a avaliação de cada solução para escolha da melhor é muito custosa. Para a viabilidade de uma solução devem ser consideradas informações de frequência da execução de transações e restrições impostas ao problema de alocação.

A seção seguinte define o problema de alocação em projeto de distribuição. Na Seção 2.4 é feita uma comparação de algoritmos da literatura que propõem uma solução para o problema de alocação de dados em SBDD.

2.3 Modelagem do problema de alocação de Fragmentos

A alocação de dados em SBDD define como os dados deverão ser alocados nos nós da rede considerada. Como todo problema de otimização combinatória, o problema de alocação de dados possui uma função de custo que é a função que define o custo de execução de transações com a alocação obtida, um conjunto de restrições que define o conjunto de limitações impostas pelo ambiente da alocação e um método de solução, que é o método usado para encontrar a melhor alocação dos dados. Esta alocação define como os dados deverão ser alocados nos nós da rede considerada.

De forma geral um Problema de Alocação de Fragmentos em SBDD pode ser descrito como na Figura 2.2.

2.3.1 Geração das Informações de Entrada

Para a aplicação de um método de solução para o problema de alocação são necessárias informações que podem ser agrupadas em informações sobre o banco de dados, aplicações, nós da rede e da rede considerada [41].

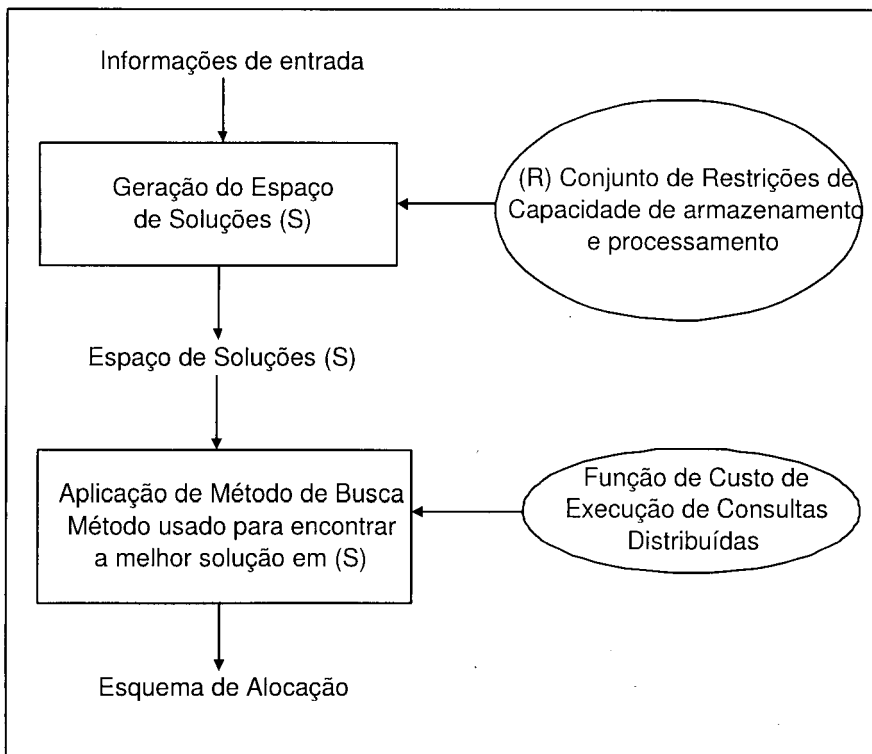


Figura 2.2: Modelo Geral para o Problema de Alocação de Fragmentos

Informações sobre o banco de dados

Envolvem informações sobre seletividade, ou seja, o percentual de dados de um fragmento f_i necessário para a execução da transação q_k . Além disso, é necessária informação sobre o tamanho dos fragmentos dada pelo produto do comprimento em bytes de uma tupla e o número de tuplas deste fragmento.

Informações sobre aplicações

Incluem informações de frequência de acessos de leitura ou atualização de uma determinada transação q_k a um fragmento f_i durante sua execução.

Também é definido o vetor O , onde $o(k)$ especifica o nó de origem da transação q_k e, finalmente, deve ser definido o tempo de resposta máximo aceitável de cada aplicação.

Informações sobre a rede

Englobam informações sobre custo de comunicação entre os nós da rede.

2.3.2 Geração do Conjunto de Restrições

Restrições se referem a tempo de resposta, tempo de processamento e capacidade de armazenamento. De forma simplificada, estas restrições podem ser definidas, segundo Özsu e Valduriez [41], como:

- Tempo de resposta

tempo de execução da transação \leq tempo máximo de resposta desta
transação

- Capacidade de armazenamento

necessidade de armazenamento de um fragmento em um nó \leq capacidade de
armazenamento daquele nó

- Capacidade de processamento

carga de processamento de uma transação em um nó \leq capacidade de
processamento daquele nó

Respeitando o conjunto de restrições é o espaço de soluções (S) sobre o qual será aplicado um método de busca.

2.3.3 Geração do Espaço de Soluções

Dado o conjunto definido na Seção 2.2 com: $S = \{s_1, s_2, \dots, s_m\}$, $Q = \{q_1, q_2, \dots, q_q\}$ e $F = \{f_1, f_2, \dots, f_n\}$, onde F é o conjunto de fragmentos resultantes da fase de fragmentação do projeto de distribuição, e n é o número de fragmentos de todas as tabelas, o custo total para a execução de cada q_k a partir de s_j sobre os diversos fragmentos envolvidos em q_k deve ser o menor possível. A variável de decisão é x_{ij} onde:

$$x_{ij} = \begin{cases} 1 & \text{se o fragmento } f_i \text{ estiver armazenado no nó } s_j; \\ 0 & \text{em qualquer outro caso.} \end{cases}$$

para todo $f_i \in F$ e $s_j \in S$.

Portanto, o espaço de soluções para o PAF é gerado como todas as possíveis combinações de valores x_{ij} que atendem ao conjunto de restrições definido na Seção 2.3.2.

2.3.4 Método de Busca

Uma vez definido o espaço de soluções inicial para o problema de alocação de dados, deve-se então definir um método para realizar a busca de uma solução para o problema.

Uma forma de resolver problemas de otimização combinatória seria simplesmente enumerar todas as soluções possíveis e escolher aquela de menor custo. Esta abordagem é denominada busca exaustiva e representa um algoritmo exato para o problema, já que garante a obtenção da solução ótima. Entretanto, para qualquer problema de um tamanho minimamente interessante ou útil, a busca exaustiva torna-se impraticável, já que o número de soluções possíveis é muito grande. Portanto, técnicas mais apuradas são necessárias para serem utilizadas como método de busca.

Os métodos heurísticos são algoritmos baseados nas propriedades estruturais ou características das soluções dos problemas que não garantem encontrar a solução ótima de um problema, mas são capazes de encontrar uma solução de qualidade em um tempo razoável, apresentando complexidade reduzida em relação aos algoritmos exatos. Estes métodos heurísticos utilizam uma função de custo que define o custo de um esquema de alocação, e permite encontrar o esquema de alocação que represente a melhor solução dentro do conjunto de soluções inicial.

A seção seguinte apresenta uma função de custo simplificada proposta em [41] para o problema de alocação em projeto de distribuição.

2.3.5 Função de custo [41]

O modelo de alocação de fragmentos em SBDD busca minimizar o custo total de execução das aplicações sobre os fragmentos. Assim a função de custo possui a seguinte forma:

$$\textit{Min}(\textit{CustoTotal})$$

A função de custo total está definida na Equação 2.1 e se apresenta com dois componentes: custo de processamento de transação sobre fragmentos distribuídos e

custo de armazenamento dos fragmentos,

$$CustoTotal = \sum_{\forall q_k \in Q} QPC_k + \sum_{\forall s_j \in S} \sum_{\forall f_i \in F} TSC_{ij} \quad (2.1)$$

onde QPC_k (*Query Processing Cost*) é o custo de processamento da transação q_k e TSC_{ij} (*Total Storage Cost*) é o custo de armazenamento do fragmento f_i no nó s_j . O primeiro somatório representa custo de processamento de cada transação e os dois últimos somatórios representam o custo de armazenamento para todos os fragmentos para todos os nós.

Considerando USC_j (*Unit Storage Cost*) o custo de armazenamento de uma unidade de dados em s_j , o custo de armazenamento do fragmento f_i no nó s_j pode ser definido de forma simples através da Equação 2.2.

$$TSC_{ij} = USC_j * tamanho(f_i) * x_{ij} \quad (2.2)$$

O custo de processamento de transações possui dois componentes: o custo de processamento (PC - *Processing Cost*) e custo de transmissão (TC - *Transmission Cost*). Assim, o custo de processamento de transação (QPC - *Query Processing cost*) da transação q_k é definido na Equação 2.3.

$$QPC_k = PC_k + TC_k \quad (2.3)$$

O componente de processamento PC , representado na Equação 2.4, consiste de três fatores de custo: (i) custo de acesso (AC - *Access Cost*), (ii) o custo de manter a integridade (IE - *Integrity Enforcement cost*) e (iii) o custo de controle de concorrência (CC - *Concurrency Control cost*):

$$PC_k = AC_k + IE_k + CC_k \quad (2.4)$$

O fator de custo de manter a integridade pode ser especificado de forma semelhante ao do controle de concorrência e dependem do algoritmo usado para realizar essas tarefas. O fator de custo de acesso é detalhado na Equação 2.5.

$$AC_k = \sum_{\forall s_j \in S} \sum_{\forall f_i \in F} (u_{ki} * UR_{ki} + r_{ki} * RR_{ki}) * x_{ij} * LPC_j \quad (2.5)$$

Os dois primeiros termos da Equação 2.5 calculam o número de acessos de leitura RR e atualização UR da transação q_k ao fragmento f_i . Assim o somatório

representa o número total de acessos para todos os fragmentos relativos a q_k . O fator LPC_j (*Local Processing Cost*) representa o custo desse acesso ao nó s_j e a variável x_{ij} garante que o custo seja calculado somente para os nós nos quais os fragmentos estão armazenados.

O componente de transmissão (TC) pode ser formulado de acordo com a função de custo de acesso. Contudo, o custo de transmissão de dados da atualização e a de leitura são diferentes. Nas transações de atualização, é necessário enviar a atualização a todos os nós onde estão as réplicas e não é necessária nenhuma transmissão de dados senão a mensagem de confirmação, enquanto que nas transações de leitura basta acessar apenas uma das cópias.

O componente de atualização de TC está definido na Equação 2.6.

$$TCU_k = \sum_{\forall s_j \in S} \sum_{\forall f_i \in F} \text{custo da msg de atualizar} + \sum_{\forall s_j \in S} \sum_{\forall f_i \in F} \text{custo de confirmação} \quad (2.6)$$

O primeiro termo se refere à transmissão da mensagem de atualização do nó de origem para todas as réplicas de fragmentos que precisam ser atualizadas. O segundo termo se refere à confirmação.

O componente de leitura de TC está definido na Equação 2.7

$$TCR_k = \sum_{\forall f_i \in F} (\min_{s_j \in S} (\text{mensagem de leitura} + \text{custo do envio da resposta})) \quad (2.7)$$

O primeiro termo representa o custo de transmissão da solicitação de leitura aos nós que possuem cópias dos fragmentos. O segundo termo representa a transmissão dos resultados desses nós para o nó de origem. O custo do envio dos dados dos fragmentos é considerado somente para os nós com menor custo de comunicação com o nó origem. Assim o componente de transmissão (TC) pode ser definido como na Equação 2.8.

$$TC_k = TCU_k + TCR_k \quad (2.8)$$

A estratégia de execução de operações de atualização considerada nesta função de custo é denominada, na literatura, como *transporte de função* [31]. Neste tipo de abordagem, o nó que dispara as operações de atualização envia uma mensagem de atualização para os nós que possuem os fragmentos envolvidos na operação. Todos

os nós executam as mesmas operações de atualização e nenhuma transmissão de dados é necessária senão a mensagem de atualização e a mensagem de confirmação que é retornada ao nó que coordena a atualização. Nesta abordagem, o custo de comunicação de atualização é baixo, uma vez que o custo de transmissão de dados é menor, pois não envolve a transmissão de dados senão mensagens de pedido e de confirmação. Entretanto, o custo de processamento pode ser alto, já que o processamento das operações de atualização será realizado por todos os nós que alocam os fragmentos envolvidos na operação.

Outra estratégia de execução de operações de atualização é o *transporte de dados*. Nesta abordagem, as operações de atualização não são executadas em todas as réplicas dos fragmentos envolvidos na operação. Em vez disso, a atualização é executada em uma das réplicas do fragmento e posteriormente os dados já atualizados são propagados para todas as outras réplicas do fragmento.

A escolha da estratégia de execução de operações de atualização apresenta influência direta na função de custo e, conseqüentemente, no método de solução no problema de alocação do projeto de distribuição. Mais detalhes sobre as estratégias de execução de operações de atualização podem ser obtidos em [28], [29] e [30].

2.4 Trabalhos Relacionados

Existem na literatura diversas propostas para o problema de alocação em sistemas de bancos de dados distribuídos. Em 1982, Chang e Liu [4] projetaram um algoritmo de fluxo em rede para resolver o problema de alocação em banco de dados. Existem alguns trabalhos que propõem uma simulação para o problema de alocação de arquivos [2], porém não são facilmente aplicáveis ao problema de alocação de fragmentos em SBDD. Várias outras formulações para o problema surgiram durante os anos [6] e [1] mas se mostraram bastante difíceis de serem aplicados à alocação de fragmentos em SBDD.

Existem várias propostas de simplificar o problema de alocação de fragmentos através de métodos heurísticos para encontrar uma solução próxima de ótima para a distribuição dos fragmentos em SBDD.

Uma função de custo para o problema de alocação de fragmentos é proposta

em [41]. Os autores fazem uma análise comparando o problema de alocação de fragmentos com o problema de alocação de dados. Segundo Özsü e Valdúriez [41], alocação de dados não é facilmente estendível para alocação de fragmentos, pois este possui características particulares. Na função de custo proposta, Özsü e Valdúriez [41] consideram replicação de fragmentos e o transporte de função como plano de execução de transações. Porém nenhum método de solução é proposto pelos autores.

Em [23] os autores propõem um método de alocação de fragmentos de tabelas de banco de dados relacional em duas fases. Na primeira fase, é realizado um agrupamento dos fragmentos baseado no conjunto de transações mais freqüentes e nas suas respectivas freqüências de acesso aos fragmentos. Na segunda fase, é determinada uma alocação eficiente dos grupamentos de fragmentos nos nós da rede, de maneira que o custo total de processamento da consulta determinado pelo otimizador de consulta distribuído seja minimizado. Neste trabalho o custo total de processamento é uma combinação do custo de transmissão com o custo de processamento local nos nós da rede. Esta abordagem busca manter próximos os fragmentos que são usados juntos em resposta a consultas. Esta solução se limita ao problema de alocação de fragmentos no modelo relacional por considerar a afinidade entre fragmentos. Portanto não seria facilmente aplicável à alocação de dados em outros ambientes distribuídos. A replicação de dados também não é prevista pelo método proposto pelos autores.

No trabalho apresentado em [6], é proposto um algoritmo heurístico para alocação de fragmentos de tabelas de banco de dados relacionais. O algoritmo apresentado combina um método guloso com um algoritmo First-Fit [21], [22]. Na função de custo utilizada neste trabalho, são consideradas restrições de processamento e capacidade de armazenamento dos nós. O resultado final do algoritmo proposto em [6] não prevê replicação. Os resultados do algoritmo não são comparados com nenhum outro algoritmo.

Em [5] os autores propõem uma abordagem heurística para alocação próxima de ótima para um sistema de objetos distribuídos, com uma função de custo estática. Por considerar uma análise em tempo de projeto, ele assume que um conjunto de aplicações executando em nós diferentes do sistema distribuído não muda com o

tempo, assim como aplicações não mudam de nós e a frequência de execução de qualquer aplicação em qualquer nó é constante. Objetos não migram de uma classe para outra, assim mudanças de tipos dinâmicos não são consideradas. O método de solução não trata replicação de classes ou fragmentos de classe.

Em [24], é proposto um método de solução para o problema de alocação de fragmentos sem restrição quanto ao modelo de dados, ou seja, o método pode ser aplicado tanto ao modelo relacional quanto ao modelo orientado a objeto. Os autores afirmam que o componente mais custoso para realização de uma transação em um sistema de banco de dados distribuídos é custo de transferência dos dados, ou seja, o custo de mover o dado do nó onde está localizado para o nó onde a transação for executada. Segundo os autores, a maior dificuldade deste problema advém da dependência mútua que existe entre a estratégia de execução das consultas (decidida pelo otimizador de consulta) e a alocação dos fragmentos. A alocação ótima dos fragmentos depende da estratégia de execução das consultas e uma estratégia de execução de consulta ótima depende da localização dos fragmentos acessados pelas consultas. Deste modo, o problema principal na decisão de uma alocação ótima é a necessidade de um modelo que represente a dependência entre os fragmentos. Para isso os autores propõem uma árvore de operadores de consultas nos fragmentos para representar esta dependência entre os fragmentos. Esta árvore é construída através de um algoritmo de decomposição de consulta. Isto restringe esta solução para alocação de fragmentos em SBDD. A replicação também não é considerada pelos autores.

Os autores propõem em [26] um método de solução simples para o problema de alocação de dados que considera a replicação de fragmentos nos nós da rede. O objetivo do trabalho foi encontrar o número de replicações de cada fragmento e uma alocação ótima, ou próxima da ótima, de todos os fragmentos, inclusive de suas replicações, em uma WAN (*Wide Area Network*), de maneira que o custo total de comunicação seja minimizado. Em [26], são feitas considerações quanto aos algoritmos existentes na literatura, onde ou a complexidade dos algoritmos é muito grande, tornando-os de difícil aplicabilidade, ou o problema é muito simplificado, como é o caso da desconsideração da replicação. Além disso, os resultados são com-

parados com a solução ótima alcançada através de um algoritmo de busca exaustiva. A função de custo apresentada é validada com a comparação de testes realizados e resultados obtidos em um ambiente real. Porém, uma falha foi identificada no algoritmo no que diz respeito à alocação completa de fragmentos. Nos casos onde existam fragmentos que não são acessados por nenhuma transação considerada no projeto de distribuição, estes fragmentos não são alocados em nenhuma das fases da solução dos autores. Isto viola a restrição de alocação completa de dados do projeto de distribuição. Outro ponto negativo é o fato dos autores não considerarem problemas reais nos experimentos realizados. Todos os casos de testes foram com problemas de pequenas dimensões. Apesar disso o algoritmo se destacou dentre os demais por considerar replicação de dados, desconsiderado pelos demais trabalhos e por apresentar uma função de custo validada através de testes reais. Isso comprova a validade dos resultados alcançados pelo algoritmo.

A Tabela 2.1 relaciona as características mais relevantes dos trabalhos apresentados acima. Os itens analisados de cada artigo foram:

Replicação: este item indica se um determinado trabalho considera ou não a replicação de fragmentos.

Modelo de dados: este item indica o modelo de dados considerado no problema: se relacional, ou orientado a objeto ou ambos.

Informações de Fragmentos: descreve quais as informações relacionadas aos fragmentos que são consideradas como parâmetro de entrada do algoritmo de alocação. Por exemplo, o tamanho do fragmento.

Capacidade de Processamento: Indica se o trabalho considera ou não a restrição de capacidade de processamento dos nós da rede.

Capacidade de Armazenamento: Indica se o trabalho considera ou não a restrição de capacidade de armazenamento dos nós da rede.

Função Custo: representa os custos considerados na função dentre custo de comunicação entre os nós da rede (Com), custo de processamento nos nós da rede (CPU) e custo de entrada e saída (I/O).

Abordagem de atualização: Indica a abordagem de atualização adotada: Transporte de dados ou Transporte de função como descrito na Seção 2.3.5.

Comparação dos Resultados: indica se os autores apresentaram uma comparação entre os resultados obtidos com o algoritmo proposto e os resultados obtidos com outros trabalhos encontrados na literatura. Além disso, esse item indica se os autores comparam os resultados com a busca exaustiva.

Ordem de Complexidade: representa a ordem de complexidade do algoritmo.

Os trabalhos são identificados da seguinte forma:

SaWi: [6]

Huang: [26]

ShPh: [23]

KaPu: [24]

BaBh: [5]

Tabela 2.1: Comparação dos trabalhos segundo os itens analisados.

Item analisado	SaWi	Huang	ShPh	KaPu	BaBh
Replicação	Não	Sim	Não	Não	Não
Modelo de dados	Relacional	Não especificado	Relacional	Ambos	Orientado a Objetos
Informações de Fragmentos	Não Informa	Tamanho dos fragmentos	Não Informa	Informações de Dependência entre Fragmentos	Relação Método-Atributo e Método-Método
Capacidade de Processamento	Sim	Não	Não informado	Não	Não
Capacidade de Armazenamento	Não	Não	Sim	Sim	Não
Função de Custo	CPU + Com	CPU + Com	CPU + Com	Com	Com
Abordagem de atualização	Função	Dados	Híbrido	Dados	Dados
Comparação dos resultados	Resultados experimentais	Resultados experimentais comparados com trabalhos da literatura e busca exaustiva	Resultados comparados com algoritmos da literatura e com a busca exaustiva	Resultados comparados com busca exaustiva	Resultados experimentais
Ordem de Complexidade: n : n° de fragmentos distintos, m : n° de nós e q : n° de transações	Não definido	$O(nm^2q)$	Não definido	Não definido	$O(n^2)$

A Tabela 2.1 mostra as principais características dos algoritmos analisados. Além das informações gerais como replicação, ordem de complexidade e modelo de dados considerado, a comparação é realizada segundo características da função de custo dos algoritmos.

Alguns trabalhos tratam o problema de alocação de fragmentos de forma simplificada tentando reduzir a complexidade do problema desconsiderando a característica de replicação dos fragmentos, que é uma característica importante do problema de alocação, ou se limitando a um modelo de dados de modo muito específico. Somente o algoritmo proposto em [26] trata a questão de replicação em problemas para modelos de dados relacionais ou orientado a objetos. Em [42] os autores apresentam uma proposta de fragmentação e alocação das regras em um banco de dados dedutivo, considerando replicação. Por não considerar a fragmentação e alocação dos dados da base de dados, este trabalho não é aplicável a bancos de dados relacionais ou orientados a objeto. Além deste, Özsü e Valdúriez [41] apresentam uma função de custo genérica que trata a replicação, porém os autores não apresentam um método de solução para o problema de alocação.

Outra simplificação adotada por alguns trabalhos é quanto à função de custo. É o caso, por exemplo, dos trabalhos propostos em [26] e [25] que não consideram o custo de processamento e armazenamento de cada nó da rede. Isto porque eles visam a WAN o que torna o custo local desprezível. Apesar disso, Huang e Chen [26] validam a função de custo com resultados obtidos em ambiente real.

Os trabalhos comparam os resultados dos seus algoritmos com resultados experimentais e em alguns casos com a solução ótima encontrada por um algoritmo de busca exaustiva em problemas com pequenas dimensões. Nenhum deles realiza experimentos em problemas reais.

O custo de um algoritmo que atenda às principais características do problema de alocação de fragmentos da melhor maneira possível pode ser muito alto. O algoritmo proposto em [26] se apresenta com uma proposta simples, com uma ordem de complexidade relativamente baixa e que atende a quesitos importantes do problema de alocação. Além disso, como dito anteriormente, apesar de propor uma função de custo que desconsidera custos de processamento e armazenamento dos

nós, apresenta comparação com resultados reais. Através dessa função de custo, os experimentos realizados comprovam a qualidade dos resultados comparando-os com a solução ótima.

A proposta desta tese é apresentar dois algoritmos, *Aloc* e *GRADA*, para o problema de alocação de fragmentos em SBDD. Estes algoritmos são baseados no algoritmo proposto em [26] mantendo a característica de considerar a replicação na solução final e de não se restringir a um modelo de dados específico. Os experimentos realizados comprovam que os algoritmos *Aloc* e *GRADA* obtêm resultados próximos de ótimos. Além disso, estes algoritmos conseguem um esquema de alocação com um custo inferior ao algoritmo proposto em [26] em ambientes reais segundo experimentos realizados com o *benchmark* TPC-C [34]. Este tipo de experimento não é considerado por nenhum dos trabalhos analisados neste capítulo.

Capítulo 3

Proposta do Algoritmo de Alocação Heurístico *Aloc*

Aloc é um dos algoritmos de alocação de fragmentos propostos nesta dissertação. Baseado em um algoritmo heurístico definido na literatura [26], *Aloc* busca encontrar um esquema de alocação de fragmentos próxima de ótima. A escolha do algoritmo proposto em [26] se justifica pela simplicidade do algoritmo apresentado e principalmente pelos resultados satisfatórios obtidos pelos autores. Outro fator determinante é o fato do algoritmo proposto em [26] levar em conta a replicação de fragmentos e refletir o comportamento real das transações em um banco de dados distribuído. Outra característica é sua independência do modelo de dados, assim é possível estendê-lo para problemas de alocação de dados em outros ambientes que possuam seus dados distribuídos. Em [26], os autores propõem uma alocação de fragmentos partindo do princípio que a fase de fragmentação já tenha sido realizada.

O algoritmo *Aloc* [32] apresenta uma variação importante do algoritmo apresentado em [26], mantendo algumas características como a independência do modelo de dados adotado e o fato de considerar a replicação de fragmentos na solução final. De forma simplificada, a heurística implementada pelo algoritmo de [26] faz uma alocação inicial gulosa de réplicas dos fragmentos privilegiando as operações de leitura, e em uma fase seguinte reduz o número de réplicas de acordo com transações de atualização. No algoritmo *Aloc*, a alocação inicial é estendida contemplando também as operações de atualização e a fase seguinte é realizada como em [26].

Assim, é mantida a simplicidade do algoritmo, embora o espaço de soluções seja aumentado por considerar, na alocação inicial, nós que executem somente transações de atualização. Considerando que um dos principais objetivos desta dissertação é apresentar uma proposta de alocação que considera replicação, o tratamento diferenciado de transações de atualização torna-se essencial. Este aumento do espaço de soluções acarretou em um aumento do tempo de execução do algoritmo (9.26% na média, 16% no máximo em problemas grandes) em relação ao de [26], sem, no entanto, alterar a ordem de complexidade, desta forma mantendo a viabilidade de sua utilização em problemas reais. O algoritmo *Aloc*, em casos que possuem transações apenas de atualização, conseguiu alcançar resultados melhores que o algoritmo de [26] com ganhos de até 16% em aplicações como o TPC-C. Nos demais casos, ele conseguiu alcançar os mesmos resultados de [26], como será mostrado nos resultados experimentais no Capítulo 4.

Na Seção 3.1 é descrita a função de custo utilizada no algoritmo *Aloc* para avaliar o custo das soluções consideradas durante a busca. Esta função de custo foi proposta em [26] e utilizada em seu algoritmo. Assim, para viabilizar as comparações entre os algoritmos adotamos a mesma função de custo. Em seguida, na Seção 3.2 será descrito o algoritmo original proposto em [26] em seguida na Seção 3.3 será descrito o algoritmo *Aloc*.

Para facilitar o entendimento, a partir daqui o algoritmo proposto em [26] será denominando *Huang*.

3.1 Função de Custo

Como dito na Seção 2.3.5, alocação de fragmentos em um banco de dados distribuídos tem como objetivo encontrar a melhor distribuição dos fragmentos nos nós de forma a minimizar o custo da execução das transações sobre um SBDD. Na função de custo proposta em [26], os autores consideram $S = \{s_1, s_2, \dots, s_m\}$ o conjunto de nós, onde m é o número de nós da rede, $T = \{t_1, t_2, \dots, t_q\}$ o conjunto de transações que são executadas em S , onde q é o número de transações, e $F = \{f_1, f_2, \dots, f_n\}$ o conjunto de fragmentos resultantes da fase de fragmentação do projeto de distribuição, onde n é o número de fragmentos. Desta forma, o custo

total para a execução de cada t_k disparada de cada s_j sobre os diversos fragmentos envolvidos em t_k deve ser o menor possível. A variável de decisão é fat_{ij} onde:

$$fat_{ij} = \begin{cases} 1 & \text{se o fragmento } f_i \text{ estiver armazenado no nó } s_j; \\ 0 & \text{em qualquer outro caso.} \end{cases}$$

assim, FAT (*Fragment Allocation Table*) é a matriz de alocação que representa a distribuição dos fragmentos (F) nos nós (S) da rede, ou seja, o esquema de alocação.

A modelagem do problema considera um conjunto de parâmetros baseados nas informações de entrada definidas na Seção 2.3.1. Na função de custo utilizada, estes parâmetros são classificados em:

1. Parâmetros do banco de dados: os tamanhos dos fragmentos, representados por um vetor TAM , onde tam_i é o tamanho do fragmento f_i .
2. Parâmetros das transações: incluem quatro matrizes:
 - (i) a matriz RM representa os pedidos de leitura, onde rm_{ki} representa o número de vezes que a transação t_k faz acesso de leitura ao fragmento f_i , a cada vez que é executada. Este valor é inteiro e positivo podendo ser 0 caso a transação t_k não acesse o fragmento f_i para leitura;
 - (ii) a matriz UM representa os pedidos de atualização, onde um_{ki} representa o número de vezes que a transação t_k atualiza o fragmento f_i , a cada vez que é executada. Como no caso anterior, um_{ki} é um valor inteiro e positivo podendo ser igual a 0;
 - (iii) a matriz SEL representa a seletividade das transações, onde sel_{ki} representa o percentual do volume de dados de f_i que é acessado durante a execução da transação t_k ;
 - (iv) a matriz $FREQ$ representa a frequência de execução de cada transação em cada nó, onde $freq_{kj}$ é o número de vezes que o nó s_j dispara a transação t_k .
3. Parâmetros da rede: incluem o custo de transferir uma unidade de dado entre os nós da rede e o custo de construção do circuito virtual entre dois nós:
 - (i) A matriz CTR representa o primeiro parâmetro, onde ctr_{jl} é o custo de transferir um dado do nó s_j para o nó s_l . Para simplificar o problema, CTR é uma matriz simétrica onde ctr_{ii} é igual a 0 para $1 < i < m$;

- (ii) $VCini$ representa o segundo parâmetro e diz respeito ao circuito virtual que é criado entre o nó que dispara a transação e o nó que possui um fragmento acessado por esta transação. Com o fim da transação este circuito é fechado;
- (iii) $Cini$ é um custo constante de iniciar a transmissão de um pacote de dados de tamanho p_size , onde p_size é a capacidade de transmissão da rede.

De posse de tais parâmetros, define-se uma função de custo a partir da qual pode-se estimar o custo da execução de um conjunto de transações sobre uma base de dados distribuída.

Na função de custo apresentada em [26], os autores, por privilegiarem a rede WAN, desprezam o custo de processamento local e concentram apenas no custo de comunicação, já que este último é o custo de maior impacto em um ambiente distribuído. Além disso, os autores assumem a estratégia de transporte de dados, descrita na Seção 2.3.5, para a execução das transações de atualização. A estratégia de transporte de dados considera o custo de envio dos dados atualizados para os nós que contenham uma cópia destes dados e despreza o custo de envio de uma mensagem de atualização. Resultados de testes realizados pelos autores comprovaram a validade desta função de custo na comparação dos seus resultados com os obtidos em um ambiente real. Desta forma, a função para cálculo do custo de comunicação definida em [26] possui dois componentes, que são o custo de carga dos fragmentos ($CCload$) e o custo de processamento das transações ($CCproc$). $CCload$ representa o custo para enviar todos os fragmentos de um nó inicial para os nós definidos no esquema de alocação e $CCproc$ representa o custo para execução das transações de T sobre o ambiente distribuído.

Nesta dissertação, pressupomos uma alocação estática dos fragmentos. Desta forma, utilizamos a função de custo definida em [26] desconsiderando o custo de carga dos fragmentos (cujo impacto, bastante relevante em problemas que tratam da alocação dinâmica de fragmentos, é bastante reduzido ao longo do tempo no caso de problemas de alocação estática). A função de minimização do custo que utilizamos nesta dissertação pode ser então definida como na Equação 3.1,

$$\min(CC_{proc} = \sum_{j=1}^m \sum_{k=1}^q freq_{kj} * (TR_k + TU_k + VCini)) \quad (3.1)$$

onde TR e TU são os custos relacionados às transações de leitura (consulta) e de atualização respectivamente. Estes custos são detalhados a seguir. Para o cálculo de TR , supondo uma transação t_k que realiza um acesso de leitura ao fragmento f_i e é disparada a partir do nó s_j , TR_k (definido na Equação 3.2) representa o custo de executar a transação t_k , ou seja, o custo de leitura dos dados de f_i consultados por t_k todas as vezes que t_k é disparada. O nó que será lido durante a execução da transação t_k (s_s) é o que apresenta o menor custo de comunicação (CC_{com}) para transferir os dados consultados por t_k para o nó s_j , dentre todos os nós que armazenam uma réplica do fragmento f_i ,

$$TR_k = rm_{ki} * \min(CC_{com}(ctr_{js} \text{ where } fat_{is}=1, sel_{ki} * tam_i)) \quad (3.2)$$

onde fat_{is} é igual a 1 se o fragmento f_i está alocado no nó s_s , senão é igual a 0 e CC_{com} é uma função definida na Equação 3.3,

$$CC_{com}(ctr_{jl}, m_size) = C_{ini} * \frac{m_size}{p_size} + ctr_{jl} * m_size \quad (3.3)$$

onde p_size é a capacidade de transmissão da rede.

Para o cálculo do custo relacionado às transações de atualização (TU), supondo uma transação t_k que atualiza o fragmento f_i e é disparada a partir do nó s_j , TU_k (definido na Equação 3.4) representa o custo de executar a transação t_k , ou seja, a soma dos custos de comunicação para transferir os dados atualizados pela transação t_k a partir de s_j para todos os nós que alocam o fragmento f_i , de forma a garantir a consistência de todas as réplicas de f_i no sistema distribuído.

$$TU_k = \sum_{i=1}^n um_{ki} * \left(\sum_{l=1}^m fat_{il} * CC_{com}(ctr_{jl}, sel_{ki} * tam_i) \right) \quad (3.4)$$

Como dito anteriormente, diferente da função de custo apresentada em [41], a estratégia de execução de operações de atualização considerada em [26] é o transporte de dados.

3.2 Algoritmo Huang

O algoritmo *Huang* tem como parâmetros de entrada as matrizes de transações de leitura (RM), de transações de atualização (UM), de seletividade (SEL), de

freqüência (*FREQ*) e a matriz de custo de comunicação (*CTR*) como definidos na Seção 3.1. O esquema de alocação resultante deste algoritmo é representado na matriz de alocação (*FAT*). O algoritmo *Huang* está dividido em três passos. No primeiro passo é feita uma alocação inicial de réplicas dos fragmentos de modo guloso privilegiando exclusivamente os acessos de leitura feitos pelos nós a cada fragmento. No passo seguinte as réplicas de cada fragmento são reduzidas de acordo com o benefício e o custo de sua retirada. O segundo passo tem o objetivo de favorecer as transações de atualização. Finalmente, no terceiro passo são alocados alguns fragmentos não considerados no primeiro passo do algoritmo. Os três passos do algoritmo são apresentados com mais detalhes nas sub-seções seguintes.

3.2.1 Primeiro passo do algoritmo *Huang*

Para cada fragmento, uma réplica é alocada em cada nó que dispore ao menos uma transação de leitura a este fragmento. Formalmente, dado o fragmento $f_i \in F$ e a transação $t_k \in T$, então:

$$fat_{ij} = 1 \forall s_j \in S \text{ onde } freq_{kj} > 0 \text{ e } rm_{ki} > 0.$$

Neste passo não é feita nenhuma análise de custo para a alocação dos fragmentos e réplicas, portanto não existe um limite de replicações para estes fragmentos. Ao final desse passo, o esquema de alocação está de forma que o custo de execução das transações de leitura é mínimo e o custo de execução das transações de atualização pode não ser o ideal, devido ao possível grande número de réplicas de cada fragmento.

A Figura 3.1 exemplifica a alocação inicial proposta pelo primeiro passo do algoritmo *Huang* considerando os parâmetros definidos na Seção 3.1. Neste exemplo que considera um cenário com quatro nós (S_1, S_2, S_3, S_4) e quatro fragmentos (F_1, F_2, F_3, F_4), estes fragmentos são alocados segundo as transações de leitura representadas nas matrizes de transações associadas a cada nó. Assim, a matriz de transação associada ao nó S_1 denota que S_1 executa uma transação de leitura do fragmento F_2 e uma transação de atualização dos fragmentos F_1 e F_3 . De forma análoga para os demais nós. Considerando este cenário descrito, a alocação inicial do

algoritmo *Huang* consiste em alocar réplicas do fragmento F_1 nos nós S_2 e S_3 , assim como as réplicas do fragmento F_2 nos nós S_1 , S_2 e S_3 de acordo com as transações de leitura.

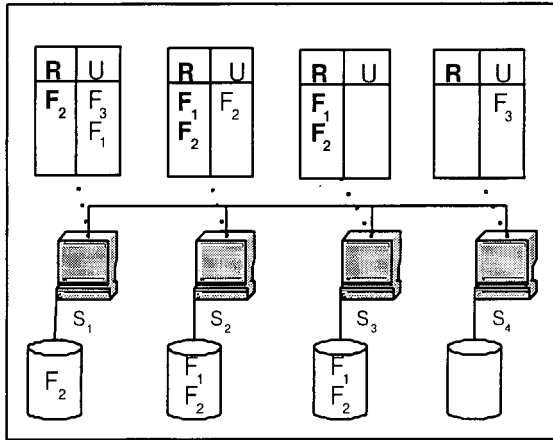


Figura 3.1: Algoritmo *Huang* - Passo 1

3.2.2 Segundo passo do algoritmo *Huang*

No segundo passo, são consideradas as atualizações com o objetivo de reduzir o custo de execução das transações de atualização, através da diminuição do número de réplicas de fragmentos alocadas no primeiro passo. O critério para a retirada da réplica de um fragmento de um determinado nó é analisar o custo adicional (gerado pelo aumento dos custos de leitura) e o benefício alcançado (com a redução dos custos de atualização) gerados pela retirada. Desta forma, para cada fragmento, é retirada a réplica com a maior diferença entre o benefício alcançado e o custo gerado com a retirada da réplica, onde este valor seja maior que 0. Isto é repetido até que o número de réplicas do fragmento seja igual a um ou não existam mais réplicas cuja diferença entre o benefício e o custo da retirada seja maior que 0. Portanto, ao final desse passo, fica garantido que existe ao menos uma réplica de cada fragmento já alocado no passo anterior em um dos nós. Este passo representa a fase de busca da melhor solução na Figura 2.2.

Supondo a retirada de uma réplica do fragmento f_r alocada ao nó s_t , temos que:

- Benefício da retirada da réplica: é definido como a eliminação do custo de

execução de todas as transações de atualização de f_r em s_t .

A função de cálculo do benefício da retirada da réplica está detalhada na Figura 3.2.

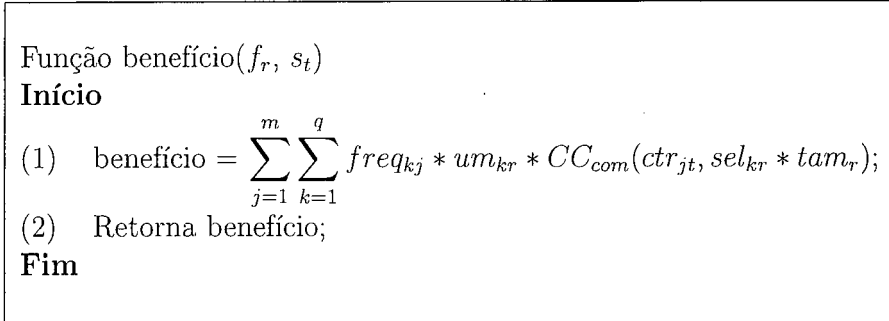


Figura 3.2: Função benefício da retirada de uma réplica

- Custo da retirada da réplica: é definido como o custo adicional de execução de todas as transações de consulta sobre f_r , e é calculado somando o custo adicional em relação a todos os nós da rede. Para cada $s_j \in S$, o custo adicional em relação a s_j é calculado da seguinte forma: suponha que s_{prox1} seja o nó mais próximo de s_j , ou seja, s_{prox1} é o nó onde ctr_{prox1j} seja mínimo, e s_{prox2} o segundo nó mais próximo de s_j . Se $s_{prox1} = s_t$, então o custo adicional em relação a s_j é a diferença entre o custo de consultar f_r em s_{prox2} a partir de s_j (T2) e o custo de consultar f_r em s_{prox1} a partir de s_j (T1). A função é definida na Equação 3.5:

$$Custo da Retirada = \sum_{j=1}^m (T2 - T1) \quad (3.5)$$

onde

$$T1 = \sum_{k=1}^q freq_{kj} * rm_{kr} * CC_{com}(ctr_{jprox1}, sel_{kr} * tam_r) \quad (3.6)$$

$$T2 = \sum_{k=1}^q freq_{kj} * rm_{kr} * CC_{com}(ctr_{jprox2}, sel_{kr} * tam_r) \quad (3.7)$$

O segundo passo do algoritmo é repetido para todo $f_i \in F$.

A função de cálculo do custo da retirada da réplica está detalhada na Figura 3.3.

Função custo(f_r, s_t)

Início

- (1) para todo s_j em S faça
- (2) Seja s_{prox1} o nó mais próximo de s_j quando s_j recupera f_r
- (3) Se ($s_{prox1} = s_t$) então
- (4) $T1 = \sum_{k=1}^q freq_{kj} * rm_{kr} * CC_{com}(ctr_{jprox1}, sel_{kr} * tam_r)$;
- (5) Seja s_{prox2} o segundo nó mais próximo de s_j quando s_j recupera f_r
- (6) $T2 = \sum_{k=1}^q freq_{kj} * rm_{kr} * CC_{com}(ctr_{jprox2}, sel_{kr} * tam_r)$;
- (7) custo = custo + (T2 - T1);
- (8) Fim_se;
- (9) Fim_para;
- (10) Retorna custo;

Fim

Figura 3.3: Função custo da retirada de uma réplica

Um exemplo do segundo passo do algoritmo *Huang* é ilustrado na Figura 3.4 que tem como entrada a matriz de alocação definida no primeiro passo do algoritmo. Pode-se observar que as réplicas dos fragmentos F_1 e F_2 que haviam sido alocadas a S_3 na Figura 3.1 foram removidas neste passo como mostrado na Figura 3.4. De acordo com o segundo passo do algoritmo, o custo de manter estas réplicas nestes nós (definido a partir das transações de atualização) é maior que o custo de retirá-las (definido a partir das transações de leitura).

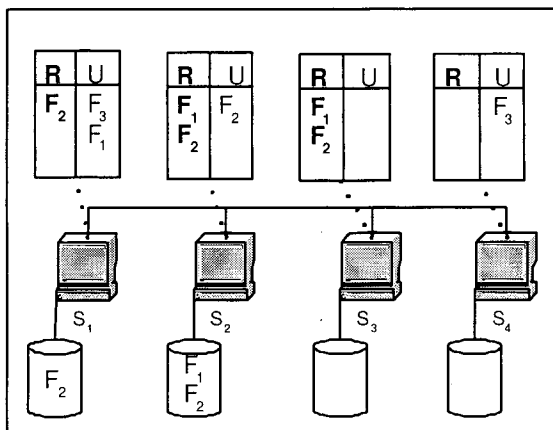


Figura 3.4: Algoritmo *Huang* - Passo 2

3.2.3 Terceiro passo do algoritmo *Huang*

No terceiro passo do algoritmo, é feita a alocação dos fragmentos que não são acessados por nenhuma transação de leitura (portanto não foram considerados no primeiro passo), mas são atualizados por alguma transação. Este passo se justifica pelo fato de que a alocação realizada nos passos anteriores considera somente a matriz de acesso de leitura (RM) para distribuir os fragmentos nos nós da rede, e a matriz de atualização é considerada somente para eliminação das possíveis replicações. Deste modo, os fragmentos que são acessados somente para atualização não são considerados na solução obtida nos passos 1 e 2.

Neste passo, são consideradas as matrizes de atualização UM e a de frequência $FREQ$. Para cada fragmento ainda não alocado e que seja atualizado por alguma transação, é encontrado o nó que possui o menor tempo de execução das transações de atualização. O cálculo do tempo de execução é dado como: para cada nó que atualiza o fragmento é calculado o custo de executar todas as transações de atualização sobre o fragmento, estando ele alocado neste nó. Esse custo é o produto da frequência de execução da transação pelo custo de comunicação entre o nó que dispara a transação e o nó candidato à alocação do fragmento. Assim, considere S_u o conjunto dos nós que atualizam f_r , o nó onde f_r será alocado é $s_v \in S_u$ onde o custo expresso na Equação 3.8 seja mínimo. O pseudo-código da função do cálculo do nó com menor tempo de execução está detalhado na Figura 3.6. Portanto neste passo os fragmentos são alocados em um e somente um nó. O algoritmo *Huang* está detalhado na Figura 3.5.

$$\sum_{s_j \in S} \sum_{T_k \in T} um_{kr} * freq_{kj} * ctr_{vj} \quad (3.8)$$

Em [26] os autores mostram que os resultados obtidos por seu algoritmo são melhores que o algoritmo proposto em [27]. Segundo Huang e Chen [26], o problema considerado em ambos os trabalhos possui a mesma configuração. Segundo Huang e Chen [26], a ordem de complexidade do algoritmo proposto no seu trabalho é $O(nm^2q)$, onde n é o número de fragmentos distintos, m é o número de nós e q é o número de transações.

O algoritmo *Huang* apresenta uma falha quanto ao princípio da alocação com-

pleta dos fragmentos. Em um projeto de alocação cada fragmento considerado deve se apresentar alocado em ao menos um nó da rede. Considerando um problema onde exista ao menos um fragmento que não seja acessado por nenhuma das transações consideradas, o algoritmo *Huang* falha desconsiderando este(s) fragmento(s) na solução final. Essa situação é bastante comum já que o objetivo da fragmentação é evitar que toda a tabela seja acessada.

Na Figura 3.5, o primeiro passo do algoritmo representa a geração do espaço de busca com a distribuição de réplicas do fragmento de acordo com as transações de leitura e o segundo passo representa o método de busca com análise da retirada de cada réplica alocada no primeiro passo. O terceiro passo faz a distribuição dos fragmentos que são acessados somente por transações de atualização.

As linhas (1)-(5) representam a solução inicial com alocação dos fragmentos segundo as transações de leitura. As linhas (6)-(15) representam a heurística para a redução das réplicas na rede e as linhas (16)-(21) representam a alocação dos fragmentos que não são acessados por transações de leitura.

As iterações, descritas nas linhas (2)-(6) da Figura 3.6, representam uma busca por cada nó da rede. Na linha (4) é escolhido o nó que atende a condição definida na linha (3). Este nó aloca o fragmento f_r e apresenta o menor custo de executar todas as transações de atualização em f_r segundo função definida na linha (2).

O último passo do algoritmo *Huang* é ilustrado na Figura 3.7 que aloca ao nó S_4 o fragmento F_3 até então não alocado nos passos 1 e 2 (Figura 3.1 e 3.4). A falha do algoritmo *Huang* é representada pelo fragmento F_4 . Por não ser considerado por nenhuma das transações do projeto de alocação, este fragmento não é alocado.

Segundo o modelo apresentado na Figura 2.2, o algoritmo *Huang* pode ser visto como representa a Figura 3.8.

3.3 Algoritmo *Aloc*

Um algoritmo proposto nesta dissertação é o algoritmo *Aloc* que propõe uma alteração do algoritmo *Huang* cobrindo as falhas encontradas. O algoritmo *Aloc* altera o primeiro passo do algoritmo *Huang*, ou seja, a fase de geração do espaço de soluções de maneira que a alocação inicial dos fragmentos considere também as

Algoritmo *Huang*

Início

Passo 1

- (1) para todo t_k de T , f_i de F e s_j de S faça
- (2) se $(RM_{ki} * FREQ_{kj} > 0)$ então
- (3) $FAT_{ij} = 1$;
- (4) Fim_se;
- (5) Fim_para;

Passo 2

- (6) para todo f_i de F faça
- (7) enquanto **CopiasFrag**(f_i) > 1 então
- (8) Seja s_j o nó onde $FAT_{ij} = 1$ e
(**benefício**(f_i, s_j) - **custo**(f_i, s_j)) é máximo
- (9) se ((**benefício**(f_i, s_j) - **custo**(f_i, s_j)) > 0) então
- (10) $FAT_{ij} = 0$
- (11) senão
- (12) pare; {Volta para a linha 6}
- (13) Fim_se;
- (14) Fim_enquanto;
- (15) Fim_para

Passo 3

- (16) para todo f_i de F faça
- (17) se **CopiasFrag**(f_i) = 0 então
- (18) $s_j = \text{atraso}(f_i)$;
- (19) $FAT_{ij} = 1$;
- (20) Fim_se;
- (21) Fim_para;

Fim

Figura 3.5: Pseudo-código do algoritmo *Huang* [26]

transações de atualização, aumentando o espaço de soluções considerado pelo método de busca no segundo passo. Deste modo, uma característica que difere o algoritmo *Aloc* do algoritmo *Huang* é a distribuição inicial dos fragmentos. Além disso, o algoritmo *Aloc* também modifica o terceiro passo do algoritmo *Huang*, para garantir a alocação completa dos fragmentos no esquema de alocação final encontrado.

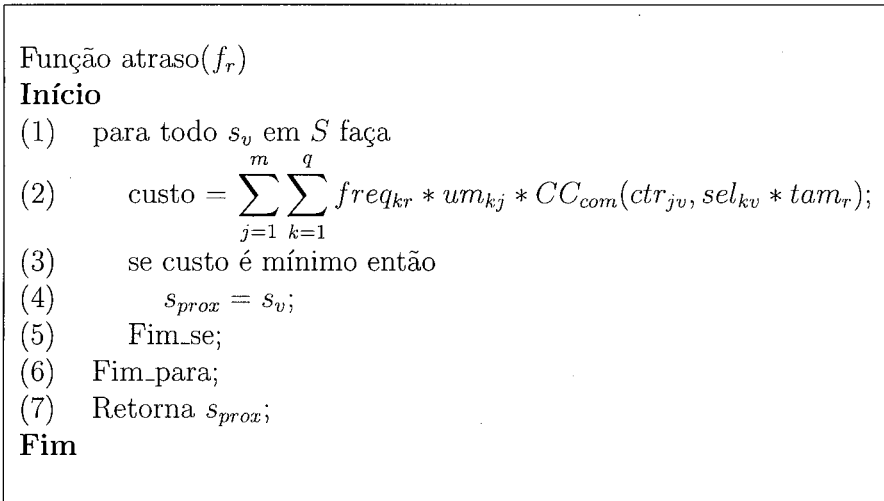


Figura 3.6: Pseudo-código da função atraso

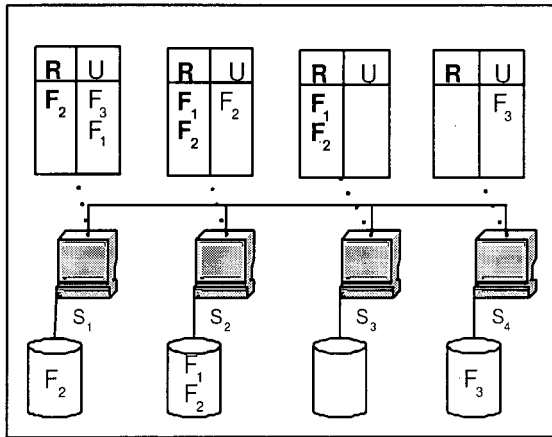


Figura 3.7: Algoritmo *Huang* - Passo 3

3.3.1 Primeiro passo do algoritmo *Aloc*

No primeiro passo do algoritmo *Aloc*, uma réplica de cada fragmento é alocada em cada nó que dispare ao menos uma transação de leitura ou de atualização a este fragmento, ou seja, dado o fragmento $f_i \in F$ e a transação $t_k \in T$, onde $rm_{ki} > 0$ ou $um_{ki} > 0$, então

$$fat_{ij} = 1 \text{ para todo nó } s_j \in S \text{ onde } freq_{kj} > 0 \text{ e } (rm_{ki} > 0 \text{ ou } um_{ki} > 0)$$

Como no algoritmo anterior, não é feita nenhuma análise de custo para a distribuição inicial dos fragmentos, portanto não existe um limite de replicações para estes fragmentos. A solução apresentada no primeiro passo possui o custo mínimo de execução das transações de leitura. Ao final desse passo cada fragmento

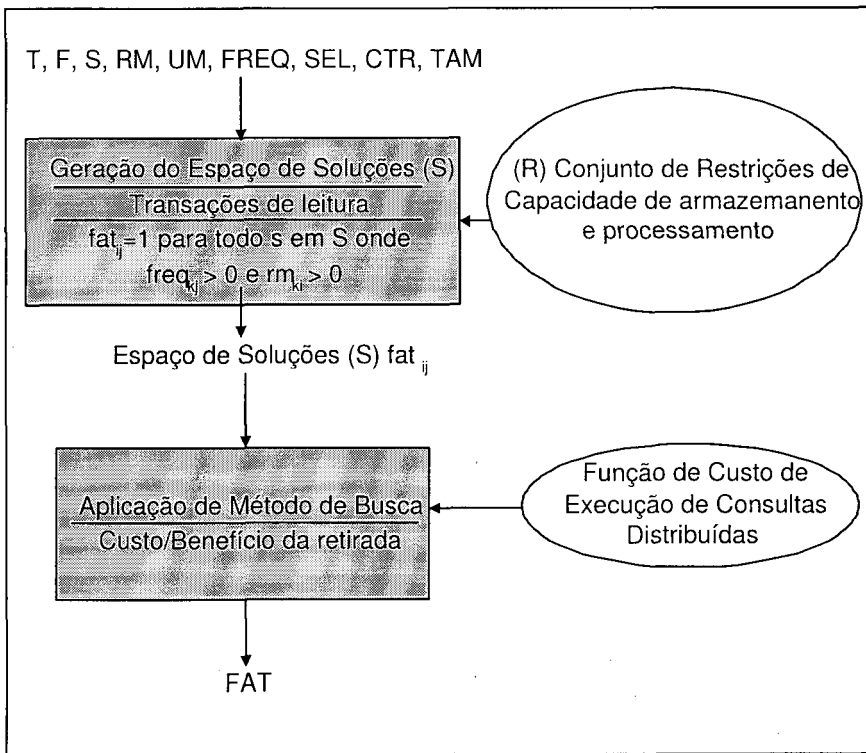


Figura 3.8: Modelo para PAF segundo algoritmo *Huang*

que é acessado por alguma transação de leitura ou atualização está alocado em ao menos um nó da rede, fato que não acontecia no algoritmo *Huang*. Assim o espaço de soluções considerado no segundo passo é maior que no algoritmo *Huang*.

O primeiro passo do algoritmo *Aloc* é ilustrado na Figura 3.9, que considera um cenário com os mesmos parâmetros de entrada que a Figura 3.1, ou seja, quatro nós (S_1, S_2, S_3, S_4) e quatro fragmentos (F_1, F_2, F_3, F_4), e as mesmas matrizes de transação de cada nó. Observa-se que, no algoritmo *Aloc* (Figura 3.9), o número de alocações é maior que no algoritmo *Huang* (Figura 3.1). Na Figura 3.9, o fragmento F_3 é alocado nos nós S_1 e S_4 , e o fragmento F_1 é alocado no nó S_1 , o que não ocorre na Figura 3.1 pois o fragmento F_3 não é acessado por nenhuma transação de leitura.

3.3.2 Segundo passo do algoritmo *Aloc*

O segundo passo do algoritmo *Aloc* é idêntico ao segundo passo do algoritmo *Huang*, porém no algoritmo *Aloc* é criada uma lista LRO (Lista de Réplicas Ordenada) para cada fragmento, contendo todas as réplicas do fragmento onde a diferença

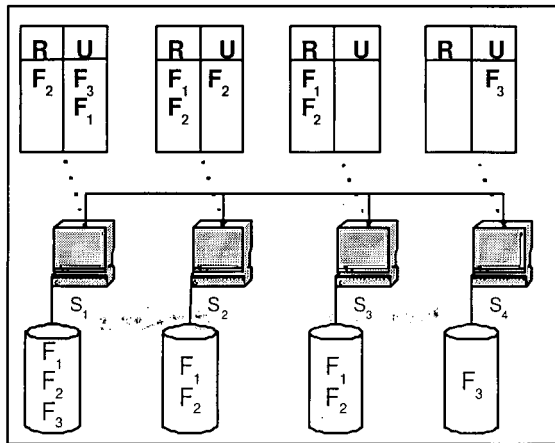


Figura 3.9: Algoritmo *Aloc* - Passo 1

entre o benefício e o custo de sua retirada seja maior que zero, ordenada de forma decrescente. Desta forma, para um determinado fragmento, retira-se o primeiro elemento da sua LRO, e isto é repetido até que o número de réplicas deste fragmento seja igual a um ou não existam mais elementos na LRO. Esta análise é feita para todos os fragmentos. A cada retirada de uma réplica de um determinado fragmento, a lista LRO é refeita para este fragmento. As funções para definir o benefício e o custo da retirada de uma determinada cópia de um fragmento seguem as definições apresentadas anteriormente (Figuras 3.2 e 3.3, respectivamente). O algoritmo *Huang* não define explicitamente a utilização de uma estrutura de lista no segundo passo de sua execução, da mesma forma que não deixa claro se a ordem das réplicas de um determinado fragmento é refeita a cada retirada de uma de suas réplicas.

A Figura 3.10 ilustra a simulação do segundo passo do algoritmo *Aloc* onde é reduzido o número de réplicas de cada fragmento. A ilustração da lista LRO utilizada no segundo passo do algoritmo *Aloc* para o fragmento F_2 neste exemplo poderia ser inicialmente $[S_3, S_1, S_2]$. No final da retirada das réplicas de F_2 esta lista LRO estaria na forma $[S_2]$ representando a alocação final do fragmento F_2 somente no nó S_2 .

3.3.3 Terceiro passo do algoritmo *Aloc*

O terceiro passo do algoritmo *Aloc* considera todos os fragmentos que não são acessados por nenhuma transação, desta forma garantindo a alocação completa dos

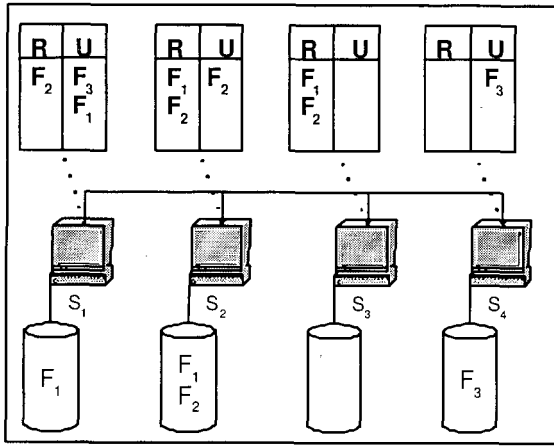


Figura 3.10: Algoritmo *Aloc* - Passo 2

fragmentos. A alocação é realizada de forma circular, ou seja, dado o conjunto de fragmentos ainda não alocados nos passos anteriores $Fr = \{fr_1, fr_2, \dots, fr_n\}$ e o conjunto de nós $S = \{s_1, s_2, \dots, s_m\}$, então a alocação ocorre da seguinte forma:

$$fat_{r_1,1} = 1, fat_{r_2,2} = 1, \dots, fat_{r_m,m} = 1, fat_{r_{m+1},1} = 1, \dots$$

Esta consideração, que não é feita no algoritmo Huang, permite garantir a completude da alocação, ou seja, garantir que todo fragmento que pertença ao conjunto de fragmentos resultantes da fase de fragmentação do projeto de distribuição seja alocado em ao menos um nó.

A Figura 3.11 ilustra o último passo do algoritmo *Aloc*. Aqui é possível verificar que o fragmento F_4 , desconsiderado pelo algoritmo *Huang*, foi alocado no nó S_1 , garantindo a alocação completa dos fragmentos.

O algoritmo *Aloc* está detalhado na Figura 3.12. As linhas (1)-(5) representam a solução inicial com alocação beneficiando tanto as transações de leitura quanto atualização (passo 1). As linhas (6)-(12) representam a heurística para a redução das réplicas na rede (passo 2) e as linhas (13)-(23) representam a alocação dos fragmentos que não são acessados por nenhuma transação considerada no projeto de alocação (passo 3).

As iterações para a construção da lista LRO, descritas nas linhas (1)-(8) da Figura 3.13, são realizadas para cada nó $s_j \in S$. Na linha (4), o nó s_j é armazenado na lista LRO somente se o nó s_j armazena o fragmento f_r , segundo a linha (2), e o

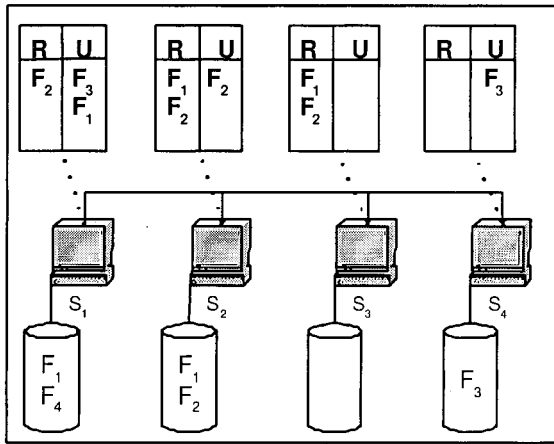


Figura 3.11: Algoritmo *Aloc* - Passo 3

benefício da retirada do fragmento f_r do nó s_j for maior que o custo dessa retirada, como definido na linha (3).

A adaptação do modelo apresentado na Figura 2.2 para o algoritmo *Aloc* pode ser representado pela Figura 3.14. O algoritmo *Aloc* propõe uma alteração na geração do espaço de soluções inicial, privilegiando também transações de atualização. Além disso, no algoritmo *Aloc* o terceiro passo é alterado visando garantir a alocação completa dos fragmentos.

3.4 Análise comparativa dos algoritmos

O *Aloc* propõe uma variação da geração do espaço de busca com a distribuição de cópias do fragmento de acordo com as transações de leitura e atualização no primeiro passo do algoritmo e mantém o método de busca no segundo passo alterando o terceiro passo para garantir a alocação completa dos fragmentos.

Esta alteração proposta no algoritmo *Aloc* aumenta o espaço de soluções do algoritmo em relação ao algoritmo *Huang*. Além disso, essa alteração influencia a ordem de retirada das réplicas do fragmento. Considere um fragmento f_r e um nó s_t de onde se quer retirar o fragmento f_r , considere ainda um s_u que não executa nenhuma transação de leitura em f_r mas executa alguma transação de atualização neste fragmento. Como o algoritmo *Huang* não considera transações de atualização na alocação inicial, somente o algoritmo *Aloc* realizará a alocação de f_r em s_u no primeiro passo do algoritmo. Esta alocação influenciará o cálculo do benefício e do


```

Algoritmo Aloc
Início
Passo 1
(1) para todo  $t_k$  de  $T$ ,  $f_i$  de  $F$  e  $s_j$  de  $S$  faça
(2)   se  $((RM_{ki} * FREQ_{kj} > 0)$  ou  $(UM_{ki} * FREQ_{kj} > 0))$  então
(3)      $FAT_{ij} = 1$ ;
(4)   Fim_se;
(5) Fim_para;

Passo 2
(6) para todo  $f_i$  de  $F$  faça
(7)   Monta LRO( $f_i$ );
(8)   enquanto  $((\text{CopiasFrag}(f_i) > 1)$  e  $| \text{LRO}(f_i) | > 1)$  faça
(9)     Desaloca primeira réplica de LRO( $f_i$ );
(10)    Monta LRO( $f_i$ );
(11)   Fim_enquanto;
(12) Fim_para;

Passo 3
(13)  $s_j = s_1$ ;
(14) para todo  $f_i$  de  $F$  faça
(15)   se  $(\text{CopiasFrag}(f_i) = 0)$  então
(16)      $FAT_{ij} = 1$ ;
(17)     se  $(s_j = s_m)$  então
(18)        $s_j = s_1$ 
(19)     senão
(20)       incrementa( $j$ );
(21)     Fim_se;
(22)   Fim_se;
(23) Fim_para;
Fim

```

Figura 3.12: Pseudo-código do algoritmo *Aloc*

custo da retirada de f_r de s_u .

O benefício da retirada da réplica não será afetado por esta alocação uma vez que no seu cálculo não são consideradas as demais alocações de f_r (Figura 3.2). Assim o benefício da retirada de cada réplica é igual para os dois algoritmos, ou seja, não depende do número de réplicas.

Para o custo da retirada da réplica, considere s_j um nó que executa transação de leitura em f_r e que s_t seja o nó que aloca f_r e apresente o menor custo de

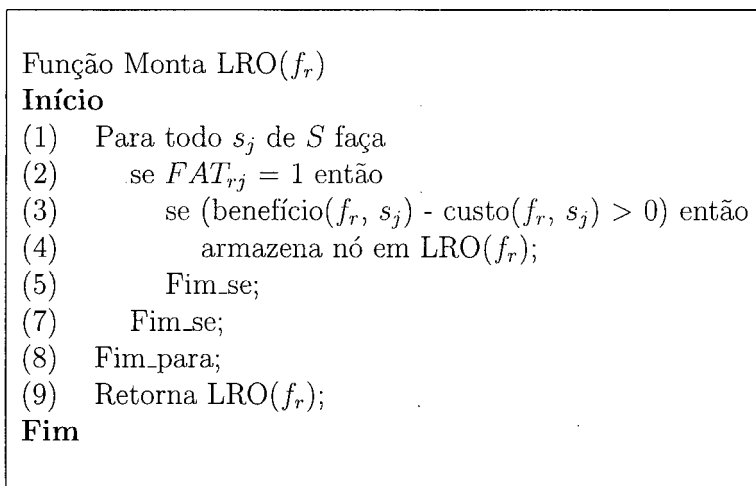


Figura 3.13: Pseudo-código da função LRO (Lista de Réplicas Ordenadas)

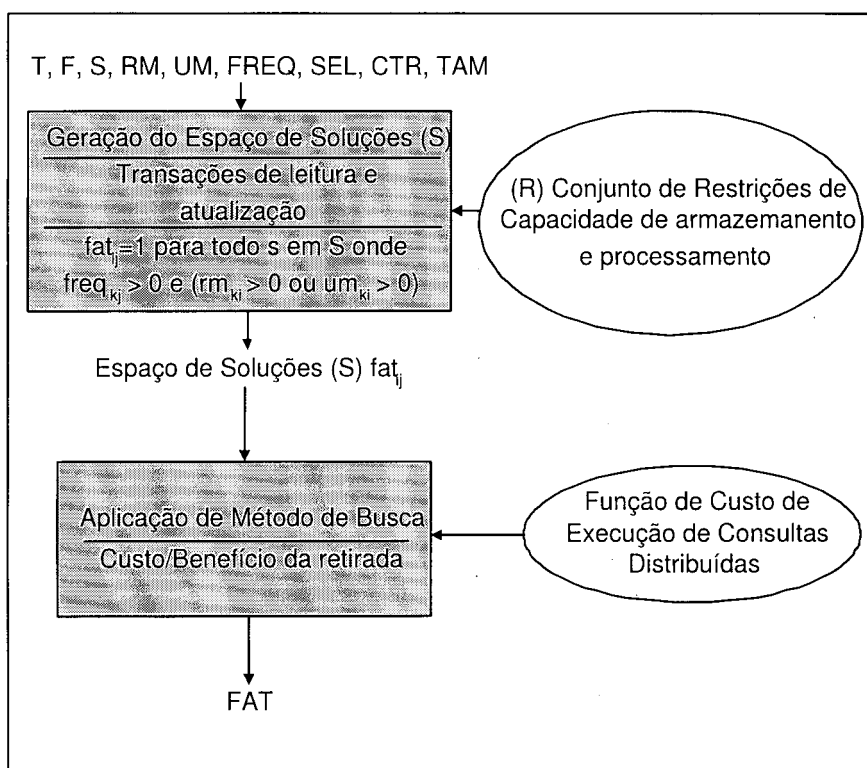


Figura 3.14: Modelo para PAF segundo algoritmo *Aloc*

comunicação com s_j . A alocação de f_r em s_u , que acontece apenas no algoritmo *Aloc*, somente influenciará no cálculo do custo da retirada se s_u for o segundo nó mais próximo de s_t (Equação 3.5). O custo de s_j executar as transações de leitura em f_r será menor se este estiver alocado em s_u do que em qualquer outro nó que o algoritmo *Huang* possa ter alocado. Com isso, o custo da retirada é reduzido no algoritmo

Aloc. Por isso, conclui-se que a inclusão de réplicas em nós que executam somente transação de atualização sobre um determinado fragmento influenciará somente com a redução da diferença entre o custo e o benefício da retirada de outras réplicas.

Capítulo 4

Avaliação do *Aloc*

Este capítulo apresenta os resultados experimentais que foram realizados executando o algoritmo *Aloc*, o algoritmo *Huang* e, em alguns casos, um algoritmo de busca exaustiva que encontra a solução ótima. Foram realizados três grupos de experimentos: experimentos com dados aleatórios para problemas com grande intensidade de transações de leitura e atualização, experimentos com dados segundo especificação do *benchmark* TPC-C [34] e experimentos com dados aleatórios para problemas com atualização intensa.

As Seções 4.1 e 4.2 e 4.3 descrevem cada ambiente utilizado em cada experimento e os resultados alcançados pelos algoritmos.

4.1 Experimentos com alto volume de transações

Neste primeiro grupo de experimentos, os algoritmos foram executados para um conjunto de problemas pequenos (em torno de 4 nós, 4 fragmentos e 4 transações) gerados aleatoriamente, utilizando a mesma metodologia utilizada em [26]. Para um número fixo de nós, transações e fragmentos, são geradas - aleatoriamente - variações nos dados de entrada envolvendo a frequência de transações, custo de comunicação, fatores de seletividade e tamanho dos fragmentos. Durante a geração dos experimentos foram considerados cenários com grande frequência de transações. O objetivo deste primeiro grupo de experimentos é comparar o custo das soluções obtidas pelo algoritmo *Aloc* com os resultados obtidos através do algoritmo *Huang* e com a solução

ótima em ambientes com grande intensidade de transações de leitura e atualização. A importância destes testes, mesmo considerando problemas de tamanho reduzido, se deve à impossibilidade de executar o algoritmo de busca exaustiva para problemas maiores, com um número de soluções possíveis muito grande.

Nestes experimentos foi utilizada a função de custo apresentada na Seção 3.1. Esta função de custo possui três parâmetros constantes: (i) $Cini$: custo de iniciar a transferência de um pacote de dados; (ii) p_size : tamanho do pacote de dados e (iii) $VCini$: custo de construção do circuito virtual. Nos experimentos realizados foram considerados os mesmos valores adotados em [26]: $Cini = 0,032$ ms/byte, $p_size = 6250$ bytes e $VCini = 65,5676$ ms.

As Figuras 4.1, 4.2, 4.3 e 4.4 mostram a relação entre os resultados obtidos com o algoritmo de busca exaustiva (solução ótima) e os algoritmos *Aloc* e *Huang* para os casos aleatórios. As variações 1-5 apresentadas no eixo X foram geradas variando-se as matrizes de entrada (RM , UM , SEL , $FREQ$, CTR e TAM) e mantendo o número de nós, transações e fragmentos. No eixo Y está representado o custo de execução das transações no esquema de alocação.

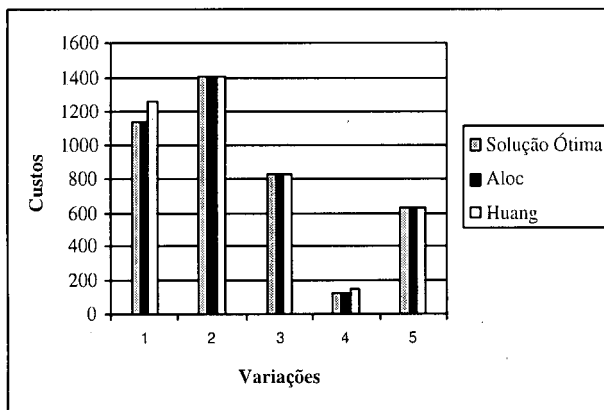


Figura 4.1: Resultados dos algoritmos de solução ótima, *Aloc* e *Huang* em cenários com alto volume de transações (4 nós, 4 transações e 4 fragmentos)

Foram geradas 5 variações com dados aleatórios. Para cada variação foram executados os algoritmos de busca exaustiva, *Aloc* e *Huang*. Sobre o conjunto de problemas gerados inicialmente na Figura 4.1, foram adicionados uma transação, um fragmento e um nó nas Figuras 4.2, 4.3 e 4.4 respectivamente. Os dados necessários para complementar as matrizes com a adição destes valores foram gerados

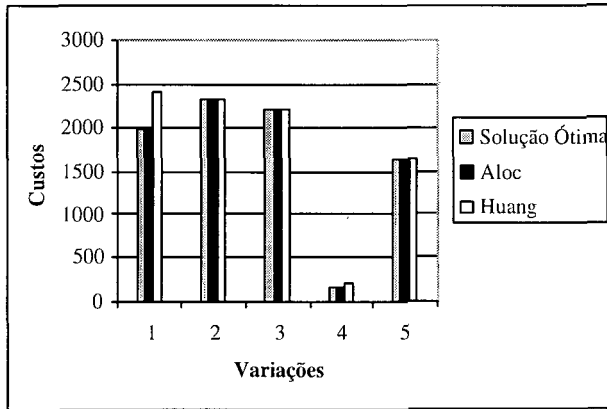


Figura 4.2: Resultados dos algoritmos de solução ótima, *Aloc* e *Huang* em cenários com alto volume de transações (4 nós, 5 transações e 4 fragmentos)

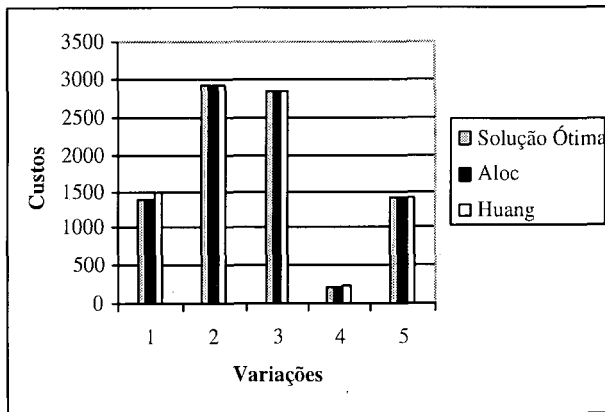


Figura 4.3: Resultados dos algoritmos de solução ótima, *Aloc* e *Huang* em cenários com alto volume de transações (4 nós, 4 transações e 5 fragmentos)

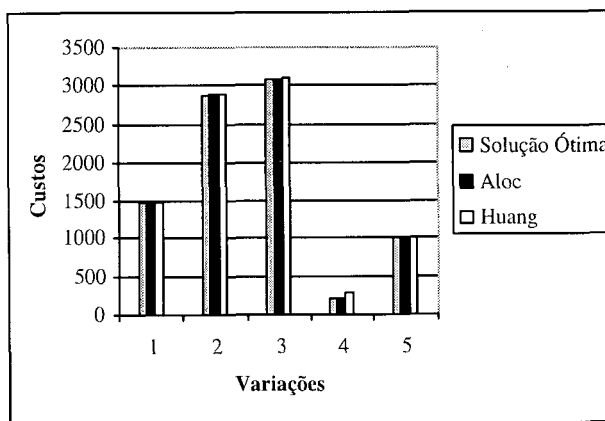


Figura 4.4: Resultados dos algoritmos de solução ótima, *Aloc* e *Huang* em cenários com alto volume de transações (5 nós, 4 transações e 4 fragmentos)

aleatoriamente.

As Figuras de 4.1 a 4.4 mostram que os resultados dos algoritmos *Aloc* e *Huang* se aproximam da solução ótima e em alguns casos o algoritmo *Aloc* apresenta uma redução no custo de execução das transações em relação ao resultado apresentado pelo algoritmo *Huang*.

A diferença entre os resultados apresentados pelos algoritmos *Aloc* e o *Huang* se acentua nas variações 1 e 4 em todas as figuras anteriores. Isto se justifica pelo fato de que, nestes casos, na solução final apresentada, existem fragmentos que foram alocados em nós que somente acessam estes fragmentos por transações de atualização. Especificamente para a variação 1 da Figura 4.2, onde esta diferença é ainda maior, verificou-se que ela reflete exatamente o custo de atualização de um fragmento que foi alocado pelo algoritmo *Aloc* a um nó que dispara somente transações de atualização a este fragmento.

Se um determinado nó que executa um maior número de transações somente de atualização possuir um custo de comunicação relativamente baixo com os demais nós da rede, o projeto de alocação do algoritmo *Aloc* será mais eficiente que o projeto apresentado pelo algoritmo *Huang*. Caso contrário os resultados dos algoritmos *Aloc* e *Huang* tendem a ser iguais.

Por se tratarem de problemas pequenos, os custos são muito próximos da solução ótima. O objetivo dessa etapa foi mostrar a proximidade dos custos em relação à solução ótima.

4.2 Experimentos segundo *benchmark* TPC-C

Neste grupo de experimentos, foram feitas simulações com os algoritmos *Aloc* e *Huang* para um caso real baseado no *benchmark* TPC-C [34]. Os experimentos foram divididos em três grupos variando o número de nós. Foram utilizadas todas as transações especificadas no *benchmark*. A fase de fragmentação do projeto de distribuição foi realizada utilizando-se os algoritmos propostos em [35] para escolha das técnicas de fragmentação aplicadas às tabelas e [36] para definição dos fragmentos de cada tabela, resultando em um total de 17 fragmentos onde a tabela *Order_Line* sofreu fragmentação vertical gerando dois fragmentos, a tabela *Item* sofreu frag-

mentação híbrida gerando cinco fragmentos, a tabela *Stock* sofreu fragmentação horizontal derivada da tabela *Item* gerando quatro fragmentos e as tabelas *Warehouse*, *District*, *History*, *New_Order*, *Customer* e *Order* não sofreram fragmentação. A Figura 4.5 representa o diagrama de entidades e relacionamentos do TPC-C.

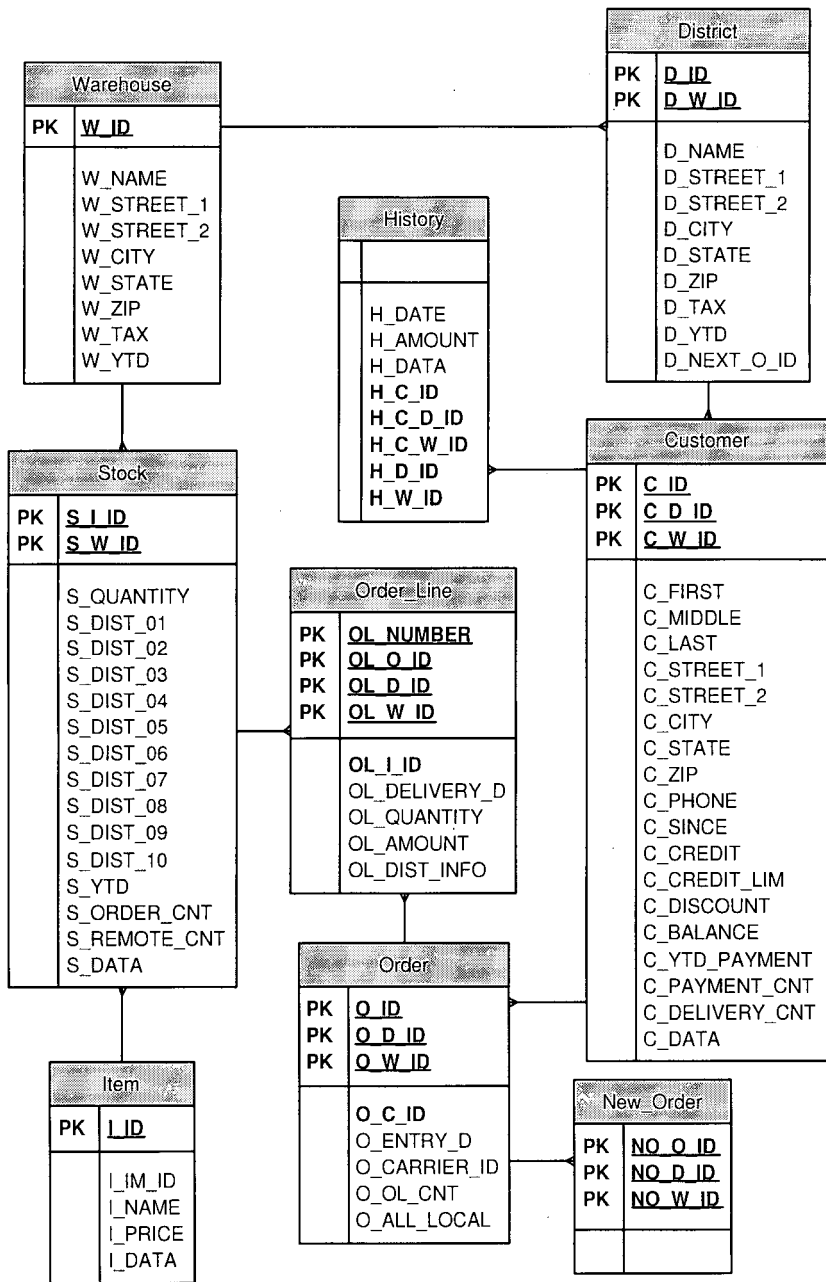


Figura 4.5: DER do benchmark TPC-C

Tabela 4.1: Fragmentação das tabelas do TPC-C.

Tabelas	Fragmentos
WAREHOUSE	$F_1 = \text{WAREHOUSE}$
DISTRICT	$F_2 = \text{DISTRICT}$
HISTORY	$F_3 = \text{HISTORY}$
NEW_ORDER	$F_4 = \text{NEW_ORDER}$
CUSTOMER	$F_5 = \text{CUSTOMER}$
ORDER	$F_6 = \text{ORDER}$
STOCK	$F_7 = \text{STOCK} \times_{I_ID} (F_{12})$
STOCK	$F_8 = \text{STOCK} \times_{I_ID} (F_{13})$
STOCK	$F_9 = \text{STOCK} \times_{I_ID} (F_{14})$
STOCK	$F_{10} = \text{STOCK} \times_{I_ID} (F_{15})$
ITEM	$ITEM_1 = \Pi_{I_ID, I_NAME, I_PRICE, I_DATA}(\text{ITEM})$
ITEM	$F_{11} = \Pi_{I_IM_ID}(\text{ITEM})$
ITEM_1	$F_{12} = \sigma_{\min_id \leq id < \max_id/4}(\text{ITEM}_1)$
ITEM_1	$F_{13} = \sigma_{\min_id/4 \leq id < \max_id/2}(\text{ITEM}_1)$
ITEM_1	$F_{14} = \sigma_{\min_id/2 \leq id < 3\max_id/4}(\text{ITEM}_1)$
ITEM_1	$F_{15} = \sigma_{3\min_id/4 \leq id < \max_id}(\text{ITEM}_1)$
ORDER_LINE	$F_{16} = \Pi_{OL_I_ID, OL_SUPPLY_W_ID, OL_QUANTITY, OL_AMOUNT, OL_DELIVERY_ID}(\text{ORDER_LINE})$
ORDER_LINE	$F_{17} = \Pi_{OL_O_ID, OL_D_ID, OL_W_ID, OL_NUMBER, OL_DIST_INFO}(\text{ORDER_LINE})$

A Tabela 4.1 representa as fragmentações realizadas sobre as tabelas originais do TPC-C.

Depois de realizada a fragmentação, alguns casos foram gerados variando a quantidade de nós e o custo de comunicação que não constam na especificação do *benchmark* e fazendo uma pequena variação da matriz de frequência permitida pela especificação. Em função do tamanho do problema, e conseqüentemente do grande número de soluções viáveis, não foram realizados experimentos com o algoritmo de busca exaustiva (cujo tempo de processamento inviabilizaria a sua execução na prática).

Como nos testes anteriores, foi utilizada a função de custo apresentada na Seção 3.1 e as constantes definidas nos modelos possuem os seguintes valores: $Cini = 0,032$ ms/byte, $p_size = 6250$ bytes e $VCini = 65,5676$ ms.

Tabela 4.2: Resultados de experimentos baseados no *benchmark* TPC-C.

Variações	Nº de Nós	Custo Huang (H)	Custo Aloc (A)	Ganho % ((H-A)/H)
1	4	40.855.813	40.855.813	0
2	4	40.858.742	40.849.526	0,02
3	8	46.622.635	42.734.635	8,33
4	8	46.586.584	42.687.064	8,37
5	8	46.608.514	42.715.906	8,35
6	8	44.674.516	40.781.908	8,71
7	12	83.094.830	72.726.830	12,47
8	12	82.944.482	72.571.874	12,50
9	12	82.408.342	78.518.038	4,72
10	12	91.858.218	87.970.218	4,23

A Tabela 4.2 mostra a relação entre os custos nos dois algoritmos. As variações podem ser agrupadas por quantidade de nós em: grupo 1 (4 nós, variações 1 e 2), grupo 2 (8 nós, variações 3 a 6) e grupo 3 (12 nós, variações 7 a 10) onde a diferença entre os elementos de cada grupo é a matriz *FREQ*, exceto para as variações 9 e 10 onde foram alterados também dados da matriz *CTR*.

Na última coluna da Tabela 4.2 é mostrada a porcentagem de ganho do algoritmo *Aloc* em relação ao algoritmo *Huang*. Verifica-se que em todas as variações realizadas *Aloc* foi mais eficiente ou idêntico ao algoritmo *Huang*.

As Figuras 4.6, 4.7 e 4.8 mostram os custos dos algoritmos para cada variação de cada grupo. Para facilitar a visualização, em todas as figuras o eixo de custo

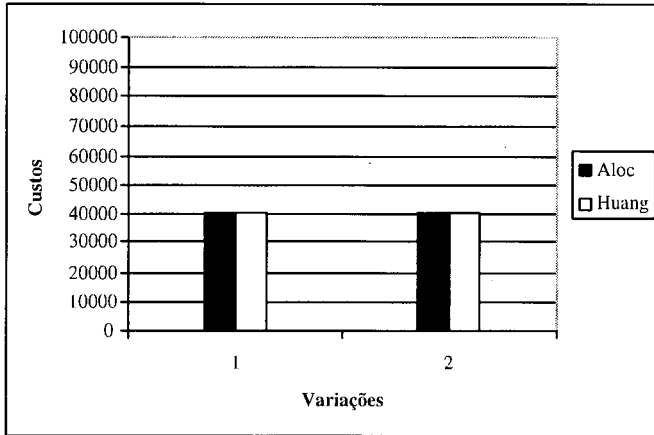


Figura 4.6: Resultados do *benchmark* TPCC com os algoritmos *Aloc* e *Huang* (4 nós, 8 transações e 17 fragmentos)

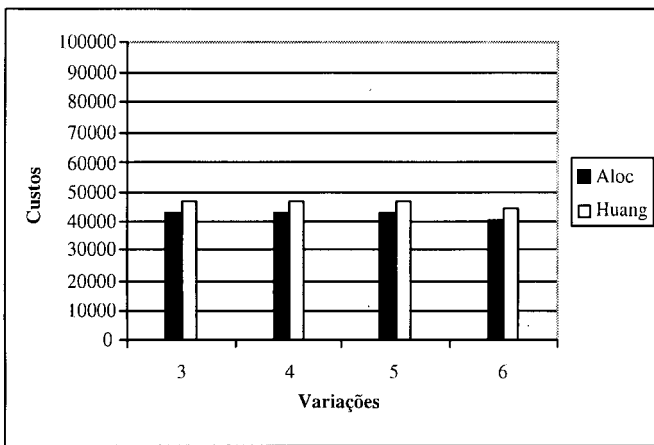


Figura 4.7: Resultados do *benchmark* TPCC com os algoritmos *Aloc* e *Huang* (8 nós, 8 transações e 17 fragmentos)

está na escala $1:10^3$. Segundo estas figuras e a Tabela 4.2 é possível verificar que a maior porcentagem de ganho ocorre nos casos onde o número de nós é maior. Nestes casos o espaço de soluções considerado no algoritmo *Aloc* é maior que no algoritmo *Huang*. Para todos os testes executados o custo do algoritmo *Aloc* foi menor ou igual segundo a função de custo definida na Seção 3.1. A Figura 4.6, onde o número de nós é somente quatro, mostra que o custo de alocação resultante dos algoritmos *Huang* e *Aloc* é muito próximo. Isso porque o número de soluções, resultantes do primeiro passo nos dois algoritmos, é muito semelhante, limitando os benefícios do algoritmo *Aloc*.

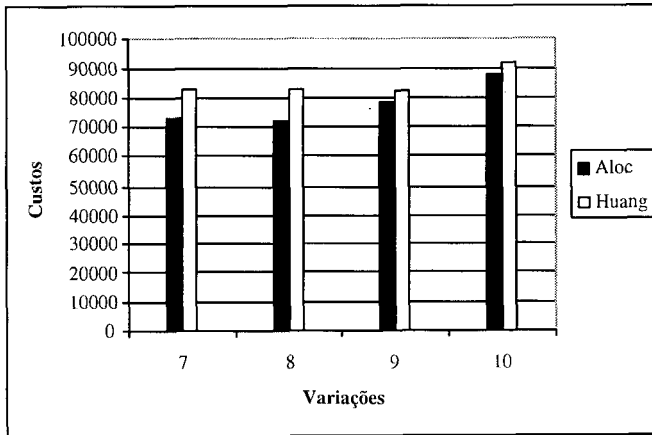


Figura 4.8: Resultados do *benchmark* TPC-C com os algoritmos *Aloc* e *Huang* (12 nós, 8 transações e 17 fragmentos)

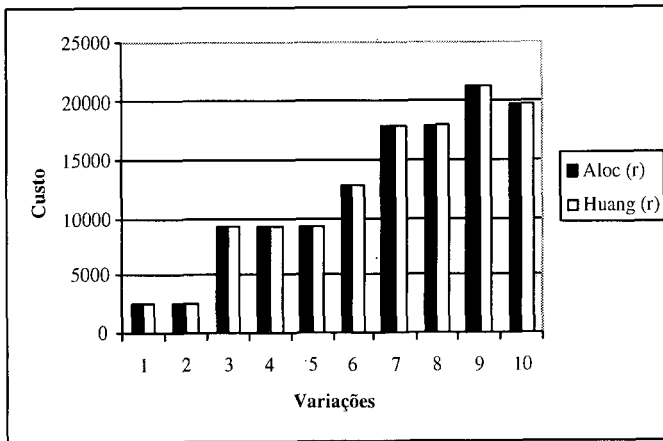


Figura 4.9: Custos de operações de leitura do *benchmark* TPC-C

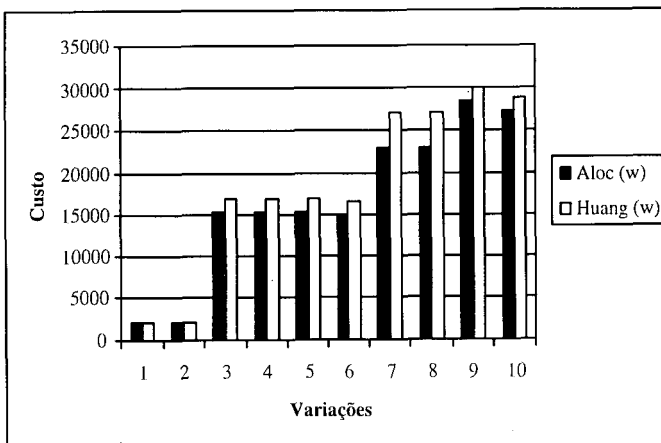


Figura 4.10: Custos de operações de atualização do *benchmark* TPC-C

O maior benefício que o algoritmo *Aloc* apresenta em relação ao *Huang* é privilegiar as transações de atualização. As Figuras 4.9 e 4.10 representam os custos das operações de leitura e de atualização de cada algoritmo respectivamente. Estes custos consideram os experimentos realizados sobre o TPC-C utilizando a mesma função de custo definida na Seção 3.1. É possível verificar nestas figuras que a redução do custo de operações de atualização é mais notável que a redução do custo de leitura. Isto fica mais visível nas variações 3 a 10 das figuras. Assim, a alteração realizada no primeiro passo do algoritmo *Huang* para alocar fragmentos conforme transações de atualização apresenta um benefício maior para as operações de atualização, permitindo que estas possam ser executadas de forma local, reduzindo o custo de transferência de dados na rede.

Os experimentos realizados sobre o TPC-C mostram o comportamento dos algoritmos em um caso real. O TPC-C pode ser visto como um ambiente com pouca intensidade de transações onde o número de transações de atualização não supera o número de transações de leitura. Para mostrar o comportamentos dos algoritmos neste caso onde o número de transações de atualização é maior que o número de transações de leitura, a Seção 4.3 relaciona o custo dos algoritmos em experimentos que representam este cenário.

4.3 Experimentos com atualização intensa

A geração dos casos utilizados nestes experimentos foi semelhante à Seção 4.1. Assim, são geradas, aleatoriamente, as matrizes de RM (leitura), UM (atualização), FREQ (frequência), SEL (seletividade), TAM (tamanho dos fragmentos) e CTR (custo de comunicação entre os nós) para um determinado número de nós, transações e fragmentos. O objetivo deste grupo de experimentos é mostrar o comportamento do algoritmo *Aloc* em ambientes onde o volume de atualização seja maior que leitura. Nestes experimentos, o número de transações de atualização chega ao dobro de transações de leitura disparadas pelo nó. Os resultados destes experimentos são mostrados na Tabela 4.3.

Como dito anteriormente, o resultado do *Aloc* tende a ser melhor em situações onde o número de transações de atualização seja maior que leitura. Isto se justifica

Tabela 4.3: Desempenho do *Aloc* com atualização intensa.

Transações	Fragmentos	Huang (H)	Aloc (A)	Ganho % $((H-A)/H)$
80	80	36.526.181	33.149.440	9,24
80	90	35.049.357	32.752.995	6,55
80	100	47.015.444	42.949.030	8,64
60	60	25.891.524	23.479.190	9,31
60	70	25.140.708	23.166.997	7,85
60	80	34.262.325	30.492.727	11,00
50	50	23.439.371	20.633.763	11,96
50	60	25.026.302	23.137.071	7,54
40	40	10.285.220	9.694.974	5,73
40	50	16.193.751	14.486.761	10,54
40	60	17.024.184	15.848.411	6,90
30	30	8.170.585	6.821.662	16,50
30	40	10.654.021	9.801.203	8,00
30	50	6.444.021	6.268.469	2,72

por, nestes casos, aumentar o espaço de soluções do algoritmo *Aloc*. A Tabela 4.3 representa esta afirmação. Em todos os casos, o algoritmo *Aloc* superou o algoritmo *Huang* considerando o custo final da matriz de alocação.

Outro fator importante que pode ser destacado nos resultados é que o ganho em relação ao custo das matrizes de alocação dos dois algoritmos não é proporcional ao número de transações ou de fragmentos. Na Tabela 4.3 a redução no custo da matriz de alocação apresentado pelo algoritmo *Aloc* é maior no caso com 30 transações e 30 fragmentos do que em situações com 80 transações e 80 fragmentos por exemplo. Este ganho está relacionado principalmente ao ambiente onde os algoritmos são executados (volume de transações, relação entre transações de leitura ou atualização) e as informações de rede (custo de comunicação entre os nós). Portanto, o crescimento do custo não é linear, conseqüentemente, o crescimento do ganho do algoritmo *Aloc* em relação ao *Huang* também não é linear.

Capítulo 5

Proposta do Algoritmo de Alocação Meta-Heurístico

Neste capítulo será apresentado o *GRADA* (Greedy RANdomized Data Allocation) que propõe uma alocação de fragmentos através de um método que agrega uma meta-heurística com característica aleatória ao algoritmo *Aloc* apresentado na Seção 3.3.

Antes de descrever o *GRADA*, alguns conceitos da meta-heurística adotada no algoritmo proposto devem ser definidos.

5.1 Meta-Heurística

A utilização de heurísticas para a solução de problemas de otimização combinatória geralmente consegue alcançar somente uma solução ótima local. Esta solução não garante ser a solução ótima global para o problema e não se consegue gerar nenhuma melhora para esta solução uma vez atingida.

Uma possibilidade de evitar que se chegue a uma solução muito aquém do ótimo global é a variação na busca das soluções que admita, em passos intermediários, soluções com valores acima do mínimo já encontrado até aquele momento, aumentando assim a chance de se encontrar, no espaço de soluções, caminhos que levem a soluções melhores e também evitando um ótimo local. O ingrediente fundamental nessas abordagens é a introdução de alguma aleatoriedade no procedimento.

Nas abordagens que permitem soluções intermediárias em que não há uma me-

lhora explícita na solução, há o risco do algoritmo entrar num ciclo (eventualmente infinito) se não houver algum esquema para eliminação de soluções repetidas. Uma forma de eliminar este problema é o uso de listas que impeça a repetição na solução. Essas listas são denominadas de listas de tabus e algoritmos baseados nessa técnica são denominados de *tabu search* [37], [38].

Todas estas abordagens são denominadas meta-heurísticas e são aplicadas a problemas de otimização.

5.2 GRASP - Procedimento de Busca Adaptativa Aleatória Gulosa

GRASP (Greedy Randomized Adaptive Search Procedures), desenvolvida por T. Feo e M. Resende [39], é uma meta-heurística que consiste em um procedimento iterativo, combinando um método construtivo com uma busca local. Cada iteração do GRASP é constituída de duas fases: uma fase de construção em que se constrói, elemento por elemento, uma solução viável para o problema, e uma fase de busca local, quando se aplica uma técnica de busca na solução construída na fase anterior, com o objetivo de encontrar um ótimo local. A melhor solução obtida na fase de busca local é comparada às soluções das iterações anteriores e a melhor solução é mantida como resultado.

Em [39] e [40], os autores apresentam o método descrevendo seus componentes básicos. Uma revisão detalhada da literatura de GRASP pode ser vista em [40], onde o autor afirma que o GRASP foi aplicado obtendo boas soluções em diversas classes de problemas de otimização combinatória, tais como: Problemas de Escalonamento, Problemas de Planarização em Grafos, Problemas de Partição em Grafos, Problema de Steiner, Problema do Clique Máximo em Grafos. Além disso, o autor cita trabalhos onde o GRASP foi implementado em paralelo, dividindo as iterações entre diversos processadores.

5.2.1 Princípios Básicos

Seja P um problema de otimização combinatória a ser resolvido aplicando a meta-heurística GRASP. O pseudocódigo abaixo descreve um algoritmo básico para

o GRASP:

```
Algoritmo GRASP()  
Início  
(1)  DadosEntrada(P);  
(2)  Inicialização da melhor solução encontrada: S*;  
(3)  Enquanto critério de parada não satisfeito faça  
(4)    Fase de Construção;  
(5)    Fase de Busca Local;  
(6)    Atualização de S*;  
(7)  Fim_enquanto;  
Fim
```

Figura 5.1: Pseudo-código do algoritmo básico GRASP

As linhas (1) e (2), da Figura 5.1, correspondem à inicialização dos dados de entrada do problema e da variável que representa a melhor solução encontrada, respectivamente. Ainda na Figura 5.1, linhas (3)-(7), estão descritas as iterações do GRASP que serão executadas até que o critério de parada definido seja atendido. Critérios de parada comumente utilizados em GRASP são: definição de um número máximo de iterações ou quando for encontrada uma solução desejada. Na linha (4) da Figura 5.1, está representada a fase de construção do método, onde, a cada iteração, uma solução inicial é construída. Já a linha (5), Figura 5.1, corresponde à fase de busca local, na qual tenta-se obter um ótimo local a partir de uma solução obtida na fase de construção. A cada iteração GRASP, S^* é atualizada, como descrito na Figura 5.1, linha (6).

5.2.2 Fase de Construção do GRASP

Na fase de construção, uma solução viável (S), para o problema P , é construída também iterativamente, elemento por elemento. A cada iteração, os elementos que podem ser inseridos em S são avaliados por uma função gulosa definida de acordo com o problema P , segundo o benefício alcançado pela inclusão do elemento na solução S . Os elementos avaliados são incluídos em uma lista, denominada Lista C , que é ordenada de forma crescente em relação ao benefício dos elementos na solução. Uma lista, denominada Lista Restrita de Candidatos (RCL - Restricted

Candidate List) ou Lista de Melhores Candidatos, é formada com os elementos de melhor avaliação. O componente probabilístico de GRASP está na escolha aleatória de um elemento da RCL para ser inserido em S. O elemento escolhido pode, ou não, ser o de melhor avaliação. Além disso, a cada iteração, a escolha do próximo elemento é realizada considerando a escolha anterior, este é o fator adaptativo de GRASP. Na Figura 5.2 está descrito o pseudocódigo para a fase de construção.

<p>Algoritmo FaseConstruçãoGRASP(S,g(C))</p> <p>Início</p> <p>(1) $S = \emptyset$;</p> <p>(2) Lista de candidatos (C) ordenada de acordo com a função gulosa $g: C \rightarrow \mathbb{R}$;</p> <p>(3) enquanto ($C \neq \emptyset$) faça</p> <p>(4) Elabore RCL;</p> <p>(5) Selecione aleatoriamente um elemento de RCL;</p> <p>(6) Inclua o elemento em S;</p> <p>(7) Atualize a Lista C;</p> <p>(8) fim_enquanto;</p> <p>Fim</p>
--

Figura 5.2: Pseudo-código para Fase de Construção do GRASP

Na linha (1) da Figura 5.2, a solução S é inicializada. Enquanto que, na linha (2) da mesma figura, a Lista C é inicializada ordenando, de forma crescente, os elementos candidatos a serem inseridos em S de acordo com a função gulosa $g: C \rightarrow \mathbb{R}$ sendo \mathbb{R} o conjunto dos números reais. As iterações, descritas nas linhas (3)-(8) da Figura 5.2, são realizadas até que uma solução viável seja construída. Ainda na Figura 5.2, linha 4, a lista restrita de candidatos, RCL, é elaborada. Na linha (5), um elemento da lista RCL é escolhido aleatoriamente e adicionado à solução S na linha (6) da Figura 5.2. Na linha (7), a lista C é atualizada. A primeira fase termina quando todos os elementos forem incluídos em S, ou seja, quando a Lista C estiver vazia.

Um fator importante em GRASP é o tamanho da Lista Restrita de Candidatos (RCL). A diversidade das soluções encontradas depende do tamanho da lista. Se $|RCL| = 1$, a cada iteração da fase de construção, a escolha do elemento que vai compor S é uma escolha gulosa e as soluções, obtidas na fase de construção, serão

iguais em todas as iterações do GRASP. Por outro lado, quanto maior o tamanho da lista RCL, maior a diversidade das soluções tornando mais claro o caráter aleatório da fase de construção.

A restrição dos elementos na RCL pode ser feita determinando um número máximo de elementos na lista e/ou definindo um intervalo, no qual os elementos devem pertencer para que possam ser incluídos na RCL.

Um outro fator importante a ser discutido em relação à definição da lista RCL é a qualidade da solução obtida que depende da qualidade dos elementos que compõem a RCL e do tamanho da lista. Em relação à qualidade dos elementos da RCL, o critério guloso deve ser elaborado de forma cuidadosa, pois os elementos que irão compor a RCL são definidos de acordo com esse critério. Portanto, caso a função gulosa não seja bem definida, podem ser geradas soluções ruins na fase de construção. Além disso, o tamanho da RCL deve ser limitado para impedir que sejam escolhidos, a cada iteração da fase de construção, elementos que determinam um pequeno benefício à solução.

5.2.3 Fase de Busca Local do GRASP

Antes de descrever esta fase do método GRASP, é preciso definir o conceito de vizinhança. Uma vizinhança $N(S)$, para o problema P , é o elemento que introduz a noção de proximidade entre uma solução S e as demais soluções de P . Uma solução S é um ótimo local se não existe nenhuma solução melhor que S em $N(S)$.

A solução gerada na fase de construção do GRASP não é, necessariamente, um ótimo local em relação a uma determinada vizinhança. Deste modo, é conveniente aplicar um método de busca local para tentar melhorar cada solução obtida na fase de construção. Assim, o dado de entrada para a fase de busca local do GRASP corresponde à solução inicial S obtida na fase de construção. Em cada iteração da fase de busca local, são realizadas tentativas de melhorias sucessivas na solução S através de uma busca na sua vizinhança. A segunda fase termina quando não é possível obter melhoras na solução S em $N(S)$.

Dado uma vizinhança N , para uma solução S , o pseudocódigo apresentado na Figura 5.3 descreve a fase de busca local:

```

Algoritmo_FaseBuscaLocal(N(S),S)
Início
(1) enquanto critério de parada não satisfeito faça
(2)     Encontre solução  $S' \in N(S)$  ;
(3)     Se  $S'$  for melhor que  $S$  então  $S = S'$ ;
(4)     Fim_enquanto;
Fim

```

Figura 5.3: Pseudo-código para Fase de Busca Local do GRASP

As linhas (1)-(4), da Figura 5.3, correspondem às iterações da fase de busca local que são executadas até que o critério de parada, definido para a fase de busca local, não seja atendido.

A eficiência do procedimento de busca local depende da definição correta da vizinhança, da estratégia de busca escolhida e da qualidade da solução inicial. Além disso, a utilização de estruturas de dados eficientes pode também contribuir para eficiência do procedimento de busca acelerando sua execução.

Uma das vantagens na utilização do GRASP se refere à qualidade da solução inicial, pois, caso os componentes da fase de construção sejam bem definidos, as soluções iniciais geradas, pela própria definição do procedimento de construção, são consideradas de boa qualidade. Outro benefício intrínseco nesta meta-heurística é a facilidade de implementação, sendo necessária a definição de poucos parâmetros como, por exemplo, a restritividade da lista de candidatos e o número de iterações. Além disso, normalmente, heurísticas gulosas são simples de se projetar e implementar.

Outra vantagem encontrada em GRASP é a variedade das soluções obtidas quando a realização de várias iterações é permitida. Para os problemas de otimização combinatória que são considerados difíceis, muitas vezes, as soluções ótimas não são conhecidas e, sendo assim, a crítica para distinguir entre soluções ótimas e próximas do ótimo não existe. Deste modo, a flexibilidade alcançada pela rapidez com que GRASP gera um grande número de soluções pode ser bastante satisfatória à avaliação destas soluções.

5.3 Algoritmo *GRADA* - Greedy RANdomized Data Allocation

O objetivo do *GRADA* (Greedy RANdomized Data Allocation) é agregar características da meta-heurística GRASP ao algoritmo *Aloc* com a finalidade de alcançar uma solução com um custo de alocação menor. Com a adição de um fator aleatório, definido pela meta-heurística GRASP, ao método guloso estabelecido pelo algoritmo *Aloc*, espera-se encontrar resultados melhores seguindo caminhos alternativos no espaço de soluções. O algoritmo *GRADA* possui os mesmos princípios do algoritmo *Aloc*, ou seja, em sua fase inicial, os fragmentos são distribuídos nos nós e na fase seguinte é feita a remoção das réplicas dos fragmentos. Porém foram utilizados princípios do GRASP no método de solução. A principal característica que difere o *Aloc* do *GRADA* é a forma com que suas réplicas são reduzidas no segundo passo, fase de busca. A função de custo e as funções para o cálculo do benefício e o custo de retirar uma réplica de um fragmento do nó são as mesmas utilizadas no algoritmo *Huang* definidas nas Figuras 3.2 e 3.3 respectivamente.

O algoritmo *GRADA*, diferente do algoritmo *Aloc*, está dividido em duas fases: fase de construção e fase de busca local. A fase de construção é semelhante ao algoritmo *Aloc*, ou seja, em um passo inicial os fragmentos são distribuídos e as réplicas são removidas em um passo seguinte. Portanto, o algoritmo *GRADA* se apresenta como uma alteração da fase de refinamento e busca da melhor solução segundo modelo proposto na Figura 2.2. Além disso *GRADA* apresenta uma fase de busca local que procura uma solução com o custo de execução das aplicações menor que a solução encontrada na fase de construção. O pseudo-código do algoritmo *GRADA* está definido na Figura 5.4.

Como dito anteriormente, a estrutura do algoritmo *GRADA* é baseada na proposta do GRASP. Assim, as linhas (2)-(4) são executadas até que uma condição de parada seja definida. No algoritmo *GRADA*, a condição de parada foi definida como um número fixo de iterações. Portanto, a fase de construção, linha (2), seguida da fase de busca local, linha (3), é executada várias vezes e se uma melhor solução for encontrada, S^* é atualizada.

Algoritmo GRADA

Início

- (1) Enquanto critério de parada não satisfeito faça
- (2) Fase de Construção;
- (3) Fase de Busca Local;
- (4) Atualização de S^* ;
- (5) Fim_enquanto;

Fim

Figura 5.4: Pseudo-código do algoritmo *GRADA*

5.3.1 Fase de Construção do *GRADA*

A fase de construção do *GRADA* é dividida em dois passos. O primeiro passo possui o mesmo objetivo do primeiro passo do algoritmo *Aloc*, ou seja, buscar uma solução inicial que beneficie tanto as transações de leitura quanto as transações de atualização. Porém, visando aumentar o espaço de soluções do algoritmo *GRADA*, cada fragmento será alocado em todos os nós da rede, independente das transações disparadas por este nó. Assim,

$$fat_{ij} = 1 \text{ para todo nó } s_j \in S \text{ e } f_i \in F$$

Como nos algoritmos anteriores, não é feita nenhuma análise de custo para a distribuição dos fragmentos e ao final todos os fragmentos estarão alocados em todos os nós, ou seja, a matriz de alocação estará completa. O objetivo dessa alteração do primeiro passo é facilitar a alocação dos fragmentos em nós que apresentem baixo custo de comunicação com os demais nós da rede. Ainda como no algoritmo *Aloc*, ao final desse passo o esquema de alocação garante uma solução ótima para a execução das transações de leitura porém o custo de execução das transações de atualização sob a solução definida pode não ser o ideal devido ao número de réplicas dos fragmentos.

O segundo passo também tem o objetivo de reduzir o número de réplicas dos fragmentos nos nós definido no primeiro passo. O critério para retirada das réplicas é o mesmo do algoritmo *Huang*, ou seja, para cada réplica de fragmento é analisado o efeito de sua retirada. Para isso é calculada a diferença do benefício e o custo

da retirada da réplica deste fragmento do nó. O cálculo do benefício e do custo é feito segundo as funções representadas nas Figuras 3.2 e 3.3 definidas na Seção 3.2, respectivamente. Porém neste algoritmo, para cada fragmento $f_i \in F$, é elaborada uma lista LRO (Lista de Réplicas Ordenadas) com todas as réplicas cuja diferença do benefício e custo da retirada seja um valor positivo e maior que 0. Esta lista está ordenada em ordem crescente da diferença. Em um passo seguinte são escolhidos os k primeiros elementos de LRO para a elaboração da lista LRDR (Lista de Réplicas Candidatas a Retirada). Em seguida é removida uma réplica aleatória de LRDR. Isto é repetido até que o número de réplicas de f_i seja igual a 1 ou não exista elementos em LRDR. Este procedimento é realizado para todo fragmento $f_i \in F$.

Como descrito na Seção 5.2.2, o valor definido para k tem influência direta no resultado. Se $|k|=1$, o algoritmo *GRADA* apresentará um comportamento semelhante ao *Aloc* com uma escolha gulosa para a retirada de uma réplica, apresentando assim o mesmo resultado alcançado pelo algoritmo *Aloc* ao final de sua execução. Por outro lado, quanto maior o valor de k , maior a diversidade das soluções tornando mais claro o caráter aleatório permitindo a escolha de caminhos do espaço de soluções que levem a resultados não satisfatórios.

O pseudo-código da fase de construção do algoritmo *GRADA* está definido na Figura 5.5. As iterações, descritas nas linhas (4)-(11) da Figura 5.5, são realizadas para cada $f_i \in F$. Na linha (5) a lista LRDR é iniciada segundo função definida na Figura 5.6. As iterações descritas nas linhas (6)-(10), são realizadas enquanto o número de réplicas de f_i seja maior que 1 e exista elementos em LRDR. Na linha (7) é definida uma réplica aleatória de LRDR para, na linha (8), ser desfeita a alocação desta réplica. Na linha (9) a lista LRDR é refeita. O primeiro passo do algoritmo *GRADA*, linhas (1)-(3) é realizada a alocação completa dos fragmentos, ou seja, cada fragmento é alocado em todos os nós da rede. Devido essa alteração no primeiro passo do algoritmo *GRADA*, torna-se desnecessário a execução do terceiro passo do algoritmo, pois ao final do primeiro passo todo fragmento está alocado em ao menos um nó da rede.

A Figura 5.6 define o pseudo-código para a função para a criação da lista LRDR. O pseudo-código para a função de criação da lista LRO está definida na

```

Início
Passo 1
(1) para todo  $f_i$  de  $F$  e  $s_j$  de  $S$  faça
(2)    $FAT_{ij} = 1$ ;
(3)   Fim_para;

Passo 2
(4) para todo  $f_i$  de  $F$  faça
(5)   Monta LRCCR( $f_i$ );
(6)   enquanto CopiasFrag( $f_i$ ) > 1 e tamanho(LRCCR) > 0 faça
(7)     define( $1 \leq t \leq$  tamanho(LRCCR));
(8)      $FAT(f_i, s_t) = 0$ ;
(9)     Monta LRCCR( $f_i$ );
(10)  Fim_enquanto;
(11) Fim_para;
Fim

```

Figura 5.5: Pseudo-código da Fase de Construção do *GRADA*

Figura 3.13 no Capítulo 3.

```

Função Monta LRCCR( $f_r$ )
Início
(1) Monta LRO( $f_r$ );
(2) Define  $k$ ;
(3) LRCCR = LRO[1.. $k$ ];
(4) Retorna LRCCR;
Fim

```

Figura 5.6: Pseudo-código da função LRCCR (Lista de Réplicas Candidatas)

Na função LRCCR definida na Figura 5.6, a linha (1) inicia a lista LRO para o fragmento f_r . Na linha (2) é definido o valor de k para, na linha (3), construir a lista LRCCR com os k primeiros elementos de LRO.

5.3.2 Fase de Busca Local do *GRADA*

A fase de busca local tem como dado de entrada a solução encontrada na fase de construção. O objetivo desta fase é encontrar uma solução dentro da vizinhança de S com um custo de alocação inferior ao custo de S . O pseudo-código da fase de

busca local está representado na Figura 5.7.

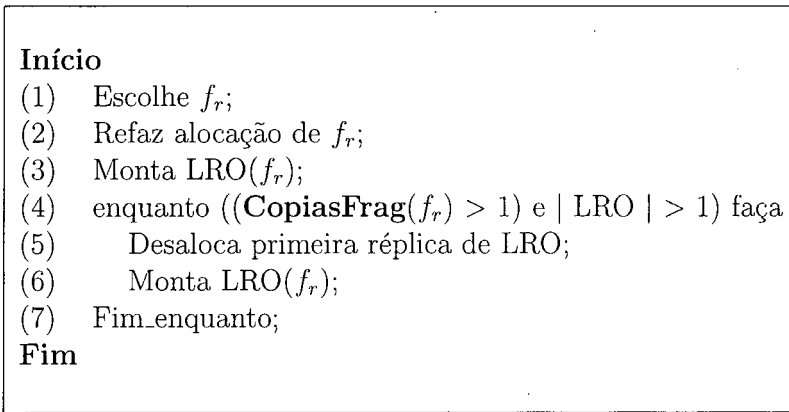


Figura 5.7: Pseudo-código da Fase de Busca Local do *GRADA*

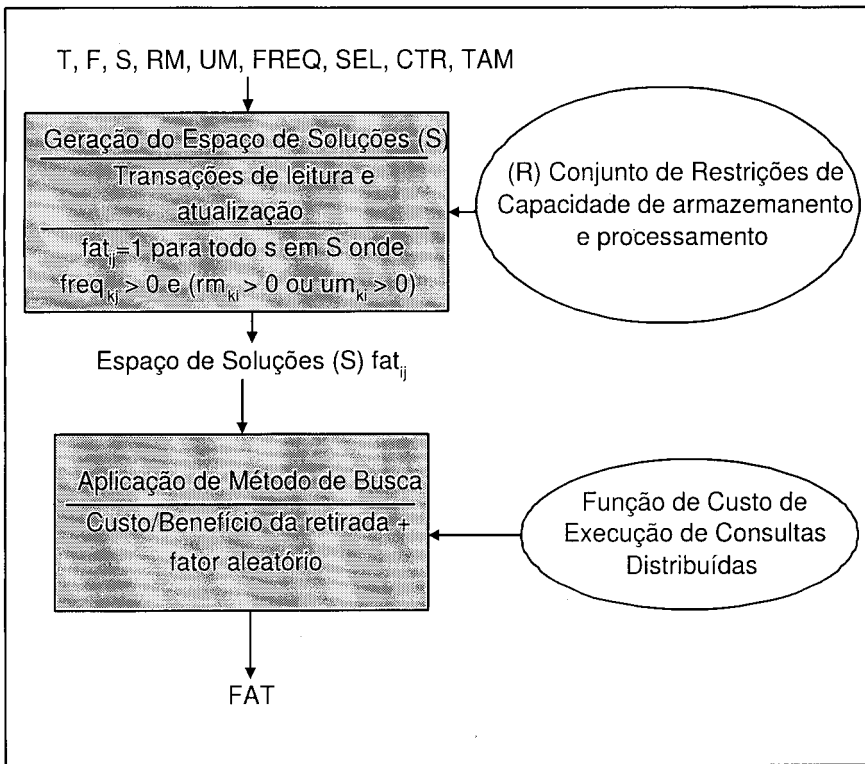


Figura 5.8: Modelo para PAF segundo algoritmo *GRADA*

Na fase de busca local do algoritmo *GRADA*, é escolhido o fragmento que possua a réplica com maior diferença entre o benefício e o custo de sua retirada na linha (1) do algoritmo da Figura 5.7. Na linha (2) é refeita a alocação desse fragmento, ou seja, f_r é alocado em todos os nós da rede. Porém pra garantir que a solução encontrada na fase de busca local será diferente da solução encontrada na

fase de construção, a alocação da réplica com maior diferença entre benefício e custo de sua retirada não será refeita. O pseudo-código da função que refaz a alocação do fragmento f_r (linha (2) da Figura 5.7) está representado na Figura 5.9. As linhas (3)-(6) correspondem ao segundo passo do algoritmo *Aloc*, ou seja, as réplicas do fragmento são removidas de forma seqüencial de acordo com a lista LRO (Lista de Réplicas Ordenadas). Assim a vizinhança da fase de busca local do *GRADA* é definida como todas as possíveis alocação do fragmento f_r para a solução definida na fase de construção.

<p>Início (1) $f_i =$ Réplica com maior diferença; (2) Distribui réplicas de f_r em todas os nós exceto f_i; Fim</p>

Figura 5.9: Pseudo-código da função de re-alocação da Fase de Busca Local do *GRADA*

O objetivo da redistribuição definida pela função representada na Figura 5.9 é obter ganhos com a variação do conteúdo e da ordem das listas LRO e conseqüentemente LRRCR. Isso é possível desconsiderando a alocação da réplica de f_r no nó com maior diferença entre o benefício e o custo de sua retirada (linha (2) da Figura 5.9).

O modelo de alocação apresentado na Figura 2.2 pode ser adaptado como mostrado na Figura 5.8 para representar o algoritmo *GRADA*. Como dito na Seção 5.3, o algoritmo *GRADA* propõe uma alteração da fase de refinamento e busca da melhor solução segundo modelo proposto na Figura 2.2.

Capítulo 6

Avaliação do *GRADA*

Este capítulo apresenta uma análise do algoritmo *GRADA* segundo experimentos realizados. Foram gerados testes aleatórios e comparados os custos das matrizes de alocação resultantes dos algoritmos *Aloc* e *GRADA*.

As matrizes utilizadas nas simulações foram geradas com valores aleatórios de forma semelhante ao primeiro grupo de testes no Capítulo 4. Aqui, foram gerados alguns problemas variando o número de fragmentos e de transações mantendo fixo o número de nós. Em cada problema, foram gerados, de forma aleatória, os dados das matrizes de entrada, entre elas as matrizes que representam frequência das transações, fatores de seletividade, custo de comunicação e o vetor de tamanho dos fragmentos. O objetivo destes experimentos é mostrar até que ponto a utilização de uma adaptação aleatória no método de alocação, segundo definido no algoritmo *GRADA*, pode apresentar ganhos nos resultados obtidos para o problema de alocação de fragmentos. Por se tratarem de problemas maiores com um grande número de soluções possíveis, ficou inviável a comparação dos resultados dos algoritmos com a solução ótima segundo o algoritmo de busca exaustiva utilizado nos experimentos realizados no Capítulo 4.

A função de custo utilizada nestes experimentos foi a mesma utilizada no Capítulo 4 e definida na Seção 3.1. Além disso foram mantidos os valores definidos nos experimentos realizados em [26] para as constantes definidas no modelo. Assim $C_{ini} = 0,032$ ms/byte, $p_size = 6250$ bytes e $VC_{ini} = 65,5676$ ms. Estes valores

também foram utilizados nos testes realizados no Capítulo 4.

Como definido no Capítulo 5, é necessária a definição de k como sendo um valor entre um e o número de elementos da lista LRO. Este valor é usado para a criação da lista LRCR.

O valor de k utilizado nos experimentos é definido como a média do número de cópias dos fragmentos. Assim o pseudo-código da função utilizada para encontrar o valor de k está definido na Figura 6.1.

```
Função Define k
Início
(1)  $k = \lfloor \frac{\text{tamanho}(LRO)}{2} \rfloor$ ;
(2) Se ( $k = 0$ ) então
(3)    $k = 1$ ;
(3) Retorna  $k$ ;
Fim
```

Figura 6.1: Pseudo-código da função de definição de k

Como dito no Capítulo 5, a escolha de um valor pra k é um fator importante para do desempenho do algoritmo. Se $k = 1$, os resultados dos algoritmos *GRADA* e *Aloc* são exatamente iguais. Assim, para problemas onde o número de réplicas dos fragmentos é pequeno, o valor de k tende a um, levando o algoritmo *GRADA* a apresentar, na maioria dos casos, o mesmo resultado alcançado pelo algoritmo *Aloc* ao final de sua execução.

Nos experimentos realizados, algumas variações do valor de k foram testadas sem resultados que beneficiassem o algoritmo *GRADA*. Além do valor apresentado nos resultados dos experimentos, foi testado o algoritmo *GRADA* com k sendo 3 ou o tamanho da lista LRCR como mostrado na Figura 6.2. Outra tentativa foi assumir k como o tamanho da lista LRCR. Como na primeira tentativa, os resultados dos experimentos não foram satisfatórios. O pseudo-código da função de definição do valor de k como sendo o tamanho da lista LRCR está representado na Figura 6.3.

As Seções 6.1 e 6.2 e 6.3 descrevem os ambientes onde os experimentos foram realizados.

```

Função Define k
Início
(1) Se tamanho(LRO) ≤ 3 então
(2)   k = 3
(3) senão
(4)   k = tamanho(LRO);
(5) Se k > 0 então
(6)   Retorna k
(7) senão
(8)   Retorna 1;
Fim

```

Figura 6.2: Pseudo-código da função de definição de k fixo

```

Função Define k
Início
(1) k := tamanho(LRO);
(2) Retorna k;
Fim

```

Figura 6.3: Pseudo-código da função de definição de k baseado na LRCR

6.1 Experimentos com alto volume de transações

O ganho do algoritmo *GRADA* em relação ao algoritmo *Aloc* ocorre em situações onde não exista uma intensidade alta de transações de leitura e atualização. A Tabela 6.1 mostra os resultados do algoritmo *GRADA* nos experimentos realizados no Capítulo 4 com 5 nós, 4 transações e 4 fragmentos. Em todos os casos o algoritmo *GRADA* obteve os mesmos resultados do *Aloc*. Isto ocorre por duas razões: nestes experimentos o número de transações de leitura e atualização disparadas por cada nó é muito alto fazendo com que a matriz de alocação resultante do primeiro passo do algoritmo *Aloc* seja semelhante a matriz gerada no algoritmo *GRADA*, ou seja, o espaço de soluções considerado no *GRADA* não é, consideravelmente, maior que no *Aloc*. Além disso, o número de nós é pequeno, reduzindo assim o fator aleatório do algoritmo *GRADA* pois a lista LRO e, conseqüentemente, a LRCR são menores. Nestes casos, a ordem de retirada das réplicas de um fragmento pode não alterar o resultado final. Além disso, a variação da ordem de retirada das réplicas é menor

podendo até não existir em alguns casos.

Tabela 6.1: Desempenho do *GRADA* com transações intensas.

Variação	Aloc (A)	GRADA (G)	Ganho % $((A-G)/A)$
1	1.473.378	1.473.378	0,00
2	2.890.625	2.890.625	0,00
3	3.098.563	3.098.563	0,00
4	204.824	204.824	0,00
5	1.001.374	1.001.374	0,00

6.2 Experimentos com baixo volume de transações

Para o ambiente onde o *GRADA* apresenta ganho em relação ao *Aloc* foi gerado um novo cenário onde o volume de transações de leitura e atualização é menor. Os resultados desses experimentos estão representados na Tabela 6.2. Nestes testes, foi considerada, na geração dos dados de entrada, uma rede com 60 nós.

Tabela 6.2: Desempenho do *GRADA* com número fixo de nós.

Transações	Fragmentos	Aloc (A)	GRADA (G)	Ganho % $((A-G)/A)$
80	80	12.351.821	10.274.823	16,81
80	90	11.258.260	8.968.226	20,34
80	100	10.120.228	7.845.888	22,47
60	60	11.320.340	9.524.058	15,87
60	70	5.299.633	4.139.981	21,88
60	80	4.943.549	3.932.438	20,45
40	40	1.543.536	1.041.434	32,53
40	50	2.056.810	1.442.766	29,85
40	60	4.845.277	3.937.091	18,74
30	30	5.098.529	3.721.619	27,01
30	40	1.407.876	1.080.771	23,23
30	50	4.804.812	3.510.630	26,94
20	10	1.475.466	962.881	34,74
20	20	964.820	812.112	15,83
20	30	1.441.387	1.182.053	17,99
20	40	1.816.655	1.554.748	14,42
20	50	2.454.769	1.950.927	20,53
10	10	449.093	391.397	12,85
10	20	485.560	375.231	22,72
10	30	1.291.267	1.016.646	21,27

As Figuras 6.4, 6.5, 6.6, 6.7, 6.8 e 6.9 são representações gráficas dos resultados

com os experimentos realizados e mostrados na Tabela 6.2. Estes dados foram agrupados em 10, 20, 30, 40, 60, e 80 transações respectivamente.

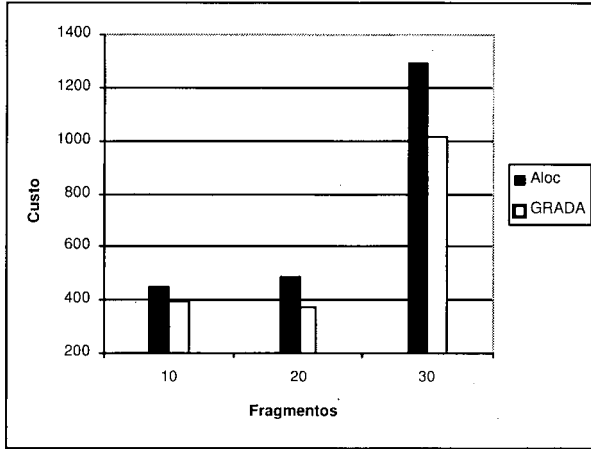


Figura 6.4: Resultado de experimentos com os algoritmos *Aloc* e *GRADA* com 10 transações em cenários com baixo volume de transações

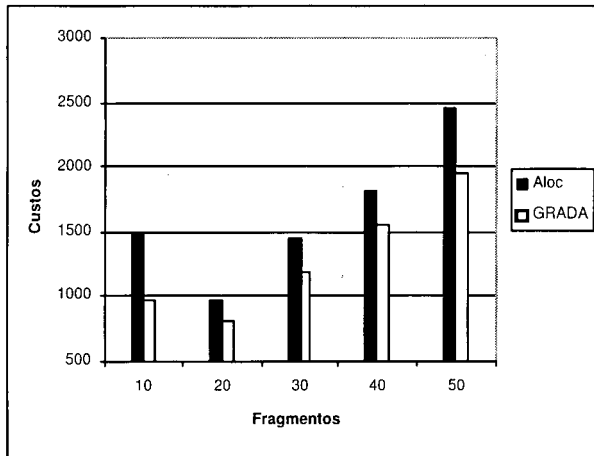


Figura 6.5: Resultado de experimentos com os algoritmos *Aloc* e *GRADA* com 20 transações em cenários com baixo volume de transações

O algoritmo *GRADA* consegue obter resultados satisfatórios em situações onde o número de transações de leitura e atualização seja reduzido. Isto porque, nestes casos, o espaço de soluções analisadas é maior que nos demais algoritmos. Nos experimentos realizados, o caso onde o ganho do *GRADA* em relação ao algoritmo *Aloc* foi de 34,74%, 14 cópias de fragmentos foram alocadas em nós que não dispõem nenhuma transação de leitura ou atualização para este fragmento. Além disso, estas

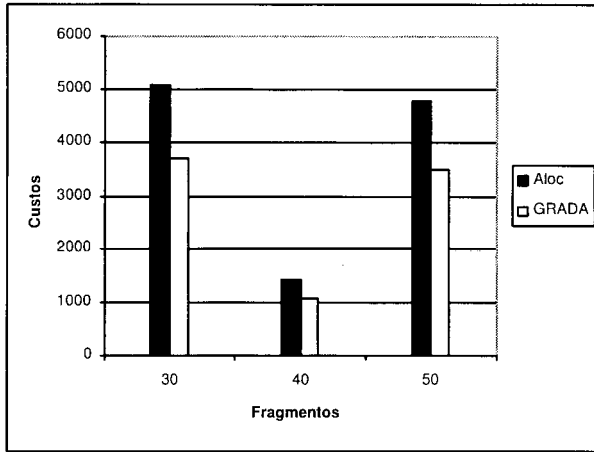


Figura 6.6: Resultado de experimentos com os algoritmos *Aloc* e *GRADA* com 30 transações em cenários com baixo volume de transações

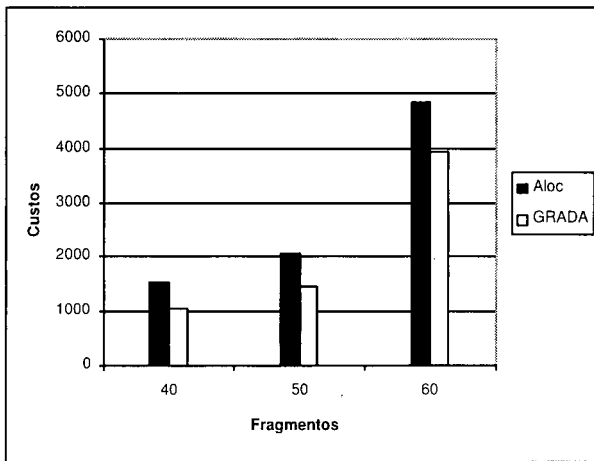


Figura 6.7: Resultado de experimentos com os algoritmos *Aloc* e *GRADA* com 40 transações em cenários com baixo volume de transações

alocações influenciam a ordem da lista de réplicas e conseqüentemente, a solução final do problema.

A Figura 6.10 representa uma situação que favorece ao algoritmo *GRADA*. Considere uma rede com os nós S_1 , S_2 e S_3 . Além disso, o fragmento F_1 é atualizado somente por transações disparadas do nó S_1 e S_2 . Assim, nos algoritmos *Huang* e *Aloc*, F_1 será alocado somente em S_1 ou S_2 . Porém, se $CTR_{13} > CTR_{12} + CTR_{23}$, ou seja, $Y > X + Z$ então a melhor alocação de F_1 será em S_3 . Esta alocação somente é considerada no algoritmo *GRADA*.

Em virtude do número de iterações do algoritmo *GRADA*, o tempo de execução

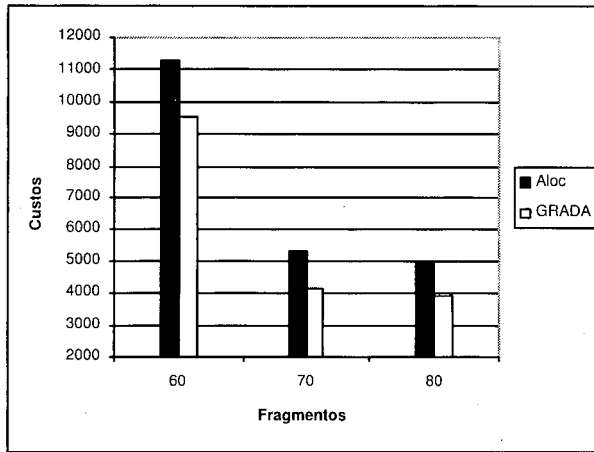


Figura 6.8: Resultado de experimentos com os algoritmos *Aloc* e *GRADA* com 60 transações em cenários com baixo volume de transações

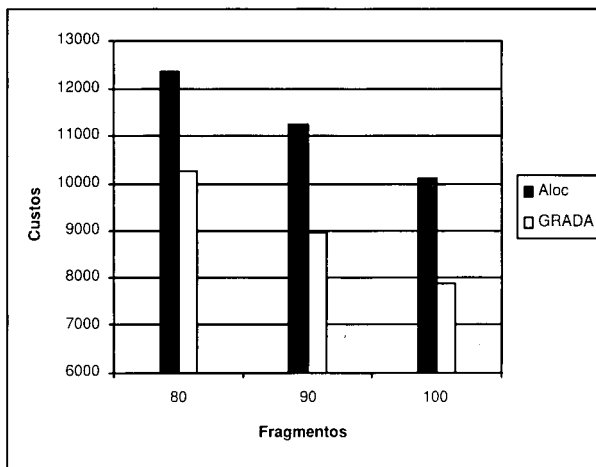


Figura 6.9: Resultado de experimentos com os algoritmos *Aloc* e *GRADA* com 80 transações em cenários com baixo volume de transações

deste algoritmo aumentou. Para um problema com 60 nós, 80 transações e 100 fragmentos o tempo de execução do algoritmo *Aloc* é de 10 segundos enquanto que o algoritmo *GRADA*, para o mesmo problema, define a matriz de alocação em 22 minutos e 16 segundos.

Nestes experimentos, o custo da matriz de alocação definida nos algoritmos *Huang* e *Aloc* é idêntico. Por existirem quantidades proporcionais de transações de leitura e de atualização, o espaço de soluções dos dois algoritmos se torna semelhante, ou seja, a matriz inicial definida no algoritmo *Huang* é semelhante a matriz inicial definida no algoritmo *Aloc*. Assim, nestas situações, o algoritmo *GRADA* é mais

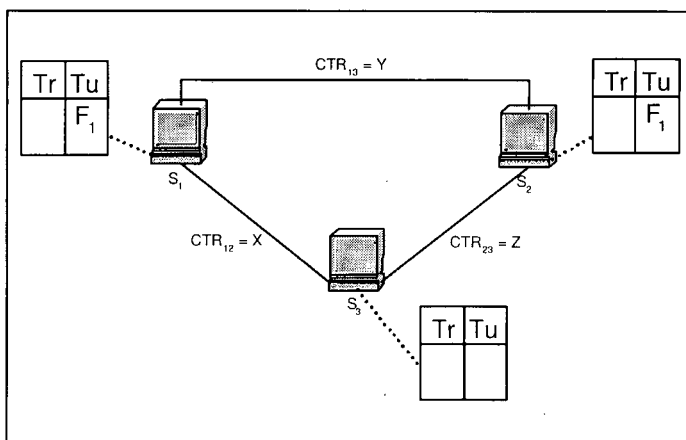


Figura 6.10: Cenário que favorece ao algoritmo *GRADA*

eficiente que os demais.

6.3 Experimentos com atualização intensa

Para o cenário que representa operações de atualização intensa apresentado na Seção 4.3, o algoritmo *GRADA* continua apresentando um custo da matriz de alocação muito reduzido se comparado ao algoritmo *Huang*, porém o ganho em relação ao *Aloc* é menor. A Tabela 6.3 mostra os custos das matrizes de alocação definidas pelos algoritmos *Aloc* e *GRADA* em cada caso gerado.

Tabela 6.3: Desempenho do *GRADA* com atualização intensa.

Transações	Fragmentos	Aloc (A)	GRADA (G)	Ganho % $((A-G)/A)$
80	80	33.149.440	30.818.624	7,03
80	90	32.752.995	30.996.188	5,36
80	100	42.949.030	40.742.122	5,13
60	60	23.479.190	21.464.758	8,57
60	70	23.166.997	21.161.206	8,65
60	80	30.492.727	27.254.680	10,61
50	50	20.633.763	19.636.653	4,83
50	60	23.137.071	20.921.526	9,57
40	40	9.694.974	9.015.492	7,00
40	50	14.486.761	12.860.379	11,22
40	60	15.848.411	14.767.712	6,81
30	30	6.821.662	6.567.259	3,72
30	40	9.801.203	8.853.159	9,67
30	50	6.268.469	5.660.011	9,70

O ganho do algoritmo *GRADA* em relação ao *Aloc* é menor nestes casos pois

aqui o espaço de soluções do algoritmo *Aloc* é maior que em situações onde o volume de transações seja menor, reduzindo assim a possibilidade de otimização do problema. Nestes casos, por apresentar um volume de transações maior que nos experimentos anteriores, a matriz de alocação inicial apresentada pelo algoritmo *Aloc* contempla mais replicações que nos casos apresentados nos experimentos anteriores, aumentando assim o espaço de busca do algoritmo. Isso reduz a possibilidade do algoritmo *GRADA* encontrar uma solução com custo inferior ao encontrado pelo algoritmo *Aloc*. Ainda assim, existem casos onde o custo da matriz de alocação final do algoritmo *GRADA* seja mais de 11% menor que o custo da matriz de alocação final do algoritmo *Aloc*. Esta diferença pode ser percebida no caso com 40 transações e 50 fragmentos apresentado na Tabela 6.3.

Baseado nos experimentos realizados, conclui-se que o *GRADA* se destaca em problemas com baixa intensidade de transações de leitura e atualização e principalmente em cenários com maior número de nós. Em ambientes com transações intensas, este algoritmo apresenta um ganho muito baixo se comparado ao algoritmo *Aloc* com um tempo de execução mais elevado.

Trabalhos podem ser desenvolvidos no sentido de analisar o melhor valor de k na criação da lista de réplicas candidatas a retirada (LRCR). O valor deste componente apresenta grande influência no resultado do algoritmo.

Capítulo 7

Conclusão

A técnica de distribuição de dados é empregada como solução para a manipulação de grandes massas de dados. Para que esta distribuição alcance os objetivos esperados é necessário que seja realizado o projeto de distribuição considerando o padrão de acesso a essas massas de dados.

A alocação de recursos em nós de uma rede de computadores é um problema antigo e, embora muito estudado na computação, é reconhecidamente complexo. Em muitos dos trabalhos existentes, os recursos a alocar dizem respeito a arquivos individuais em uma rede de computadores. A alocação de fragmentos de tabelas em SBDD envolve diversos relacionamentos entre os fragmentos da base de dados, desconsiderados na alocação de arquivos individuais. Assim, o problema de alocação de fragmentos aumenta a complexidade do problema de alocação de arquivos e não pode ser tratado da mesma forma.

O projeto de distribuição em SBDD parte do princípio que as transações mais freqüentes são conhecidas. Através da análise das operações das transações é possível identificar os relacionamentos entre os fragmentos e os pontos de acesso das operações. Essas informações devem ser aproveitadas na fase de alocação dos fragmentos, mas aumentam o espaço de soluções e os conflitos entre as decisões de alocação quando comparado ao projeto de alocação de arquivos individuais. Um dos problemas é tomar decisões quanto à necessidade de replicar fragmentos versus o custo em mantê-los consistentes. Por esses motivos, à exceção de Huang [26], os

demais algoritmos da literatura resultam na alocação de fragmentos desconsiderando as decisões de replicação. Essa alternativa simplifica bastante a solução, porém não é realista, já que com frequência um mesmo fragmento é acessado por diferentes pontos da rede.

Nesta dissertação foi apresentada uma solução para o problema de alocação de fragmentos de tabelas em SBDD, que considera a replicação, através dos algoritmos *Aloc* e *GRADA*. Além disso, o algoritmo *Aloc* foi avaliado na geração de esquemas de fragmentação para aplicações OLTP representadas pelo *benchmark* TPC-C.

Visto que este é um problema reconhecido na literatura como NP-Completo, o objetivo inicial foi encontrar um algoritmo simples heurístico que considerasse um grande número de informações quanto às aplicações de acesso à base de dados. O algoritmo definido em [26], aqui denominado de *Huang*, atende a estes quesitos e mostra sua superioridade frente ao estado da arte.

Aloc estende as heurísticas e introduz uma pequena variação no método de solução do algoritmo *Huang*, o que permitiu encontrar uma solução próxima de ótima mantendo a mesma complexidade. Foi obtida uma redução no custo de execução das aplicações ao utilizar esquema de alocação resultante de *Aloc*. Essa redução é observada para todos os casos onde existe ao menos um fragmento que é acessado por um nó exclusivamente para atualização. Além disso, *Aloc* gera esquemas de alocação mais eficientes que o algoritmo *Huang* em cenários mais próximos de aplicações reais, como o *benchmark* TPC-C, onde o número de transações é grande e o número de nós avaliados também. Mantendo a ordem de complexidade do algoritmo *Huang* ($O(nm^2q)$, sendo n o número de fragmentos distintos, m o número de nós e q o número de transações), *Aloc* conseguiu ganhos de até 16% em relação ao algoritmo *Huang*. Não foi encontrada uma situação em que o algoritmo *Huang* obtivesse um resultado superior a *Aloc*. Em aplicações pequenas, tipo exemplo ("toy examples"), o resultado de *Aloc* obteve o mesmo custo de execução que *Huang*.

O algoritmo *GRADA* estende o *Aloc* de modo original, ao aplicar conceitos da meta-heurística GRASP em seu método de solução para a alocação de fragmentos. Com a utilização do GRASP, o algoritmo *GRADA* varre um espaço de soluções maior que os demais algoritmos, aumentando a possibilidade da escolha do caminho

que leve a uma solução próxima de ótima. Os resultados obtidos com *GRADA* mostram sua superioridade ao *Aloc* (conseqüentemente ao *Huang*) ao gerar esquemas de alocação para ambientes com um grande número de nós, já que o espaço de soluções tende a ser muito grande. Porém, para que seja possível esse ganho do algoritmo *ADAG*, o tempo para sua execução é superior ao tempo de execução do algoritmo *Aloc*. Para um problema com 60 nós, 80 transações e 100 fragmentos o tempo de execução do algoritmo *Aloc* é de 10 segundos enquanto que o algoritmo *GRADA*, para o mesmo problema, define a matriz de alocação em 22 minutos e 16 segundos. A melhoria no resultado da alocação também foi observada em ambientes com baixa freqüência de atualizações. Por outro lado, para problemas com menor número de nós na rede, o algoritmo *GRADA* tende a se tornar guloso. Isso acontece pois o tamanho da lista de réplicas candidatas à retirada (LRCR) tende a ser pequeno e a chance de o algoritmo *GRADA* conseguir reduzir o custo da matriz de alocação é menor se comparado com *Aloc*.

Além de gerar exemplos de aplicações de modo aleatório, essa dissertação é inovadora ao utilizar um *benchmark* para a geração dos experimentos. Na análise dos trabalhos relacionados, poucos apresentam resultados com experimentos ou simulações. Além disso, não foi encontrado nenhum trabalho que avaliasse soluções para problemas próximos de aplicações reais como é o caso dos *benchmarks*. Os testes realizados em [26] tratam problemas pequenos com um número bastante limitado de nós, transações e fragmentos.

Dentre os *benchmarks* propostos pelo conselho TPC, a aplicação que mais se aproxima de um cenário de distribuição envolvendo operações de atualização é o TPC-C, que representa transações comerciais rápidas (OLTP). Como o TPC-C não foi proposto para SBDD, foi necessário, inicialmente, aplicar algoritmos de fragmentação sobre o esquema conceitual "global" definido no *benchmark*. Foram utilizados os algoritmos propostos em [35] e [36], considerando as transações definidas na especificação do TPC-C. Com a definição dos fragmentos, a representação das operações contidas nas transações, a definição do número de nós do ambiente e os pontos de acesso das transações, foi possível gerar os parâmetros de entrada dos algoritmos de alocação propostos nesta dissertação. Assim, foi obtida a análise do

comportamento dos algoritmos em cenários próximos de reais, até então desconhecidos.

Os algoritmos apresentados nesta dissertação foram implementados na ferramenta descrita no Apêndice A. Esta ferramenta permite executar os algoritmos com três funções de custo. Estas funções de custo implementam as duas abordagens de execução de transações de atualização, i.e., envio de dados e de funções. Mais detalhes da ferramenta podem ser obtidos em [34].

Os algoritmos apresentados nesta dissertação podem ser estendidos também para problemas de alocação de dados em ambientes com distribuição e replicação como servidores Web e alguns sistemas de integração de banco de dados heterogêneos. Isto é possível, pois os algoritmos propostos nesta dissertação tratam os fragmentos de forma genérica sem considerar o modelo de representação dos dados. Para isso bastaria modelar o problema de acordo com os parâmetros de entrada do algoritmo, identificando os fragmentos e o conjunto de transações que operam sobre eles.

Os resultados descritos nos Capítulos 4 e 6 evidenciam o sucesso no desenvolvimento dos algoritmos *Aloc* e *GRADA*. Entretanto, através dos diversos experimentos realizados, podem-se observar algumas oportunidades de melhorias ou extensões para outras aplicações sobre os algoritmos propostos. Como no algoritmo *Huang*, os algoritmos propostos nesta dissertação não consideram a re-alocação de fragmentos. Como trabalhos futuros a re-alocação e alocação dinâmica de fragmentos podem complementar as soluções geradas, considerando a variação das frequências de execução das transações ao longo do tempo.

A função de custo utilizada nos algoritmos propostos nesta dissertação foi baseada na função de custo apresentada em [2]. Em [2] os autores fazem uma comparação entre os custos obtidos com sua função de custo e com experimentos reais realizados, comprovando a proximidade entre os valores estimados e reais. Existem na literatura outras funções de custo de execução em SBDD que consideram custo de processamento local como a apresentada em [1], o que foi desprezado no modelo de [2] visando favorecer o custo de transmissão. Para efeitos comparativos, os resultados apresentados se limitaram a essa função. Estudos com a utilização de

outras funções de custo podem ser realizados através da ferramenta gerada, visando analisar o comportamento do algoritmo em ambientes que privilegiam o custo de processamento local. Testes preliminares já foram realizados com as funções já implementadas na ferramenta. Porém, no caso específico da alocação em uma rede não local, acredita-se que o maior impacto está de fato na comunicação e não no processamento local.

Algumas pesquisas futuras também podem ser realizadas sobre o algoritmo *GRADA*. Foi mostrado, através de experimentos, que a utilização do GRASP no método de solução para o problema de alocação em SBDD é bem sucedido. Porém, o valor do componente k utilizado na fase de busca local do *GRADA* não sofreu um estudo mais detalhado. Este valor define a aleatoriedade da solução e apresenta grande influência na obtenção do resultado final. Portanto, outra sugestão de trabalhos futuros é estudar uma variação deste componente buscando um valor que não elimine a aleatoriedade da solução (o que pode tornar o algoritmo *GRADA* um algoritmo guloso), nem aumente a diversidade das soluções com a possibilidade de escolha de caminhos não satisfatórios.

REFERÊNCIAS BIBLIOGRÁFICAS

- [1] LAM, K., AND YU, C. T. An approximation algorithm for a file allocation problem in a hierarchical distributed system. In *ACM SIGMOD Int. Conf. on Management of Data*, ACM Press (1980), pp. 125–132.
- [2] CASEY, R. G. Allocation of copies of a file in an information network. In *Spring Joint Computer Conf.* (1972), pp. 617–625.
- [3] ESWARAN, K. P. Placement of records in a file and file allocation in a computer network. In *Information Processing '74* (1974), pp. 304–307.
- [4] CHANG, S. K., AND LIU, A. C. File allocation in a distributed database. *International Journal of Computer Information Sciences* 11, 5 (1982), 325–340.
- [5] BARKER, K., AND BHAR, S. A graphical approach to allocating class fragments in distributed objectbase systems. *Distributed and Parallel Databases*, Kluwer Academic Publishers 10, 3 (2001), 207–239.
- [6] SACCA, D., AND WIEDERHOLD, G. Database partitioning in a cluster of processors. *ACM Transactions on Database Systems* 10, 1 (1985), 29–56.
- [7] CERI, S., PERNICI, B., AND WIEDERHOLD, G. Distributed database design methodologies. *Proceedings of the IEEE* 75, 5 (1987), 533–546.

- [8] YOSHIDA, M., MIZUMACHI, K., WAKINO, A., OYAKE, I., AND MATSUSHITA, Y. Time and cost evaluation schemes of multiple copies of data in distributed database systems. *IEEE Trans Software Eng.* 11, 9 (1985), 954–958.
- [9] MURO, S., IBARAKI, T., MIYAJIMA, H., AND HASEGAWA, T. File redundancy issues in distributed database systems. In *9th International Conference on Very Large Data Bases, Italy, Morgan Kaufman Publishers* (1983), pp. 275–277.
- [10] MURO, S., IBARAKI, T., MIYAJIMA, H., AND HASEGAWA, T. Evaluation of file redundancy in distributed database systems. *IEEE Trans Software Eng.* 11, 2 (1985), 233–242.
- [11] BAIÃO, F. Uma metodologia e algoritmos para o projeto de bancos de dados distribuídos utilizando revisão de teoria, 2001.
- [12] NAVATTHE, S., CERI, S., WIEDERHOLD, G., AND DOU, J. Vertical partitioning of algorithms for database design. *ACM Trans. on Database Systems* 9, 4 (1984), 680–710.
- [13] BELLATRECHE, L., SIMONET, A., AND SIMONET, M. Vertical fragmentation in distributed object database systems with complex attributes and methods. *7th International Workshop on Database and Expert Systems Applications DEXA '96, IEEE-CS Press* (1996), 15–21.
- [14] CHEN, Y., AND SU, S. Implementation and evaluation of parallel query processing algorithms and data partitioning heuristics in object oriented databases. *Distributed and Parallel Databases* 4, 2 (1996), 107–142.
- [15] EZEIFE, C., AND BARKER, K. A comprehensive approach to horizontal class fragmentation in a distributed object based system. *Distributed and Parallel Databases, Kluwer Academic Publishers* 3, 3 (1995), 247–272.
- [16] SAVONNET, M., TERRASSE, M., AND YETONGNON, K. Using structural schema information as heuristics for horizontal fragmentation of object classes

- in distributed oodb. In *The IX International Conference on Parallel and Distributed Computing Systems, ISCA/IEEE, Dijon, France* (1996), pp. 732–737.
- [17] KARLPALEM, K., NAVATHE, S., AND MORSI, M. Issues in distribution design of object-oriented databases. In *Özsu, M., Dayal, U., Valduriez, P. (eds), Distributed Object Management, Morgan Kaufman Publishers* (1994).
- [18] MAIER, D., GRAEFE, G., SHAPIRO, L., DANIELS, S., KELLER, T., AND VANCE, B. Issues in distributed object assembly. In *Özsu, M., Dayal, U., Valduriez, P. (eds), Distributed Object Management, Morgan Kaufman Publishers* (1994).
- [19] CERI, S., AND PERNICI, B. Dataid-d: Methodology for distributed database design. In *A. Albano, V.de Antonellis e A. di Leva (eds), Computer-Aided Database Design* (1985), pp. 157–183.
- [20] MOLINA, H., AND HSU, M. Distributed databases. In *Kim, W. (ed), Modern Database Systems, ACM Press and Addison-Wesley* (1995), pp. 484–485.
- [21] JOHNSON, D. S. Near-optimal bin packing algorithms. Relatório Técnico MIT-LCS-TR-109, Massachusetts Institute of Technology, 1973.
- [22] JOHNSON, D., DEMERS, A., ULLMAN, J., GAREY, M., AND GRAHAM, R. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing* 3 (1974), 299–325.
- [23] SHEPHERD, J., HARANGSRI, B., CHEN, H., AND NGU, A. A two-phase approach to data allocation in distributed database. In *Database Systems for Advanced Applications* (1995), pp. 380–387.
- [24] KARLPALEM, K., AND PUN, N. Query driven data allocation algorithms for distributed database system. In *8th International Conference on Database and Expert Systems Applications* (1997), pp. 347–356.
- [25] BRUNSTROM, A., LEUTENEGGER, S. T., AND SIMHA, R. Experimental Evaluation of Dynamic Data Allocation Strategies in a Distributed Database

- With Changing Workloads. Relatório Técnico TR-95-2, University of Denver, 1995.
- [26] HUANG, Y., AND CHEN, J. Fragment allocation in distributed database design. *Information Science and Engineering* 13, 3 (2001), 491–506.
- [27] LIN, X., ORLOWSKA, M., AND ZHANG, Y. On data allocation with the minimum overall communication costs in distributed database design. *Fifth International Conference on Computing and Information* (1993), 539–544.
- [28] BOWMAN, I. T. Hybrid shipping architectures: A survey. Relatório Técnico CS748T, University of Waterloo, 2001.
- [29] FRANKLIN, M. J., JONSSON, B. T., AND KOSSMANN, D. Performance tradeoffs for client-server query processing. In *ACM SIGMOD Int. Conf. on Management of Data*, ACM Press (1996), pp. 149–160.
- [30] KOSSMANN, D., AND FRANKLIN, M. J. A study of query execution strategies for client-server database systems. Relatório Técnico CS-TR-3512, University of Maryland, 1995.
- [31] ÖZSU, M. T., VORUGANTI, K., AND UNRAU, R. C. An asynchronous avoidance-based cache consistency algorithm for client caching DBMSs. In *24th Int. Conf. Very Large Data Bases, VLDB* (24–27 1998), pp. 440–451.
- [32] WILDEMBERG, M., PAULA, M., BAIÃO, F., AND MATTOSO, M. Alocação de dados em bancos de dados distribuídos. In *XVIII Simpósio Brasileiro de Banco de Dados, SBBD* (2003), pp. 215–228.
- [33] WILDEMBERG, M., PAULA, M., BAIÃO, F., AND MATTOSO, M. Xaloc: Uma ferramenta para avaliar algoritmos de alocação de dados. In *XIX Simpósio Brasileiro de Banco de Dados, Demos-SBBD* (2004).
- [34] TPC-C. Transaction processing performance council benchmark c. www.tpc.org/tpcc, 2003.

- [35] BAIÃO, F., MATTOSO, M., AND ZAVERUCHA, G. A distribution design methodology for object dbms. *Distributed and Parallel Databases, Kluwer Academic Publishers* 16, 1 (2004), 45–90.
- [36] CRUZ, F., BAIÃO, F., MATTOSO, M., AND ZAVERUCHA, G. Towards a theory revision approach for the vertical fragmentation of object oriented databases. In *XVI Brazilian Symposium on Artificial Intelligence SBIA'02, Recife, Brasil, Lecture Notes in Artificial Intelligence, Springer-Verlag* (2002), vol. 2507, pp. 216–226.
- [37] GLOVER, F. Tabu search - part i. *ORSA Journal of Computing* 1, 3 (1989), 190–206.
- [38] GLOVER, F. Tabu search - part ii. *ORSA Journal of Computing* 2, 1 (1990), 4–32.
- [39] FEO, T., AND RESENDE, M. Greedy randomized adaptive search procedures. *Journal of Global Optimization* 6 (1995), 109–133.
- [40] RESENDE, M. Greedy randomized adaptive search procedures (GRASP). *Encyclopedia of Optimization, Kluwer Academic Press, 1999.*, 1999.
- [41] ÖZSU, M. T., AND VALDURIEZ, P. *Principles of distributed database systems*, second ed. Prentice Hall, 1999.
- [42] PEREZ, J., PAZOS, R., SOLIS, J. F., ROMERO, D., AND CRUZ, L. Vertical fragmentation and allocation in distributed databases with site capacity restrictions using the threshold accepting algorithm. In *The Mexican International Conference on Artificial Intelligence (MICAI)* (2000), pp. 75–81.

Apêndice A

XALOC: Uma ferramenta para avaliar algoritmos de alocação de dados

A ferramenta apresentada, *XALoc*, propõe um esquema de alocação de fragmentos de uma base de dados pelos nós de uma rede distribuída. Através de uma interface gráfica bastante simples, ilustrada pela Figura A.1, o usuário (projetista da distribuição) da ferramenta *XALoc* fornece os parâmetros de entrada para o problema de alocação de dados, que são: o conjunto de fragmentos, nós e transações e as relações entre eles. Para a definição do esquema de alocação, o usuário tem a opção de escolher dentre quatro algoritmos distintos implementados em *XALoc*: *Huang* [26], *Aloc* [32], *GRADA* além de um algoritmo de busca exaustiva que encontra a solução ótima. Para a avaliação de cada solução considerada no algoritmo escolhido, é utilizada uma função de custo. Na *XALoc*, esta função de custo pode ser escolhida pelo usuário dentre três possíveis.

A.1 Componentes da ferramenta *XALoc*

O objetivo desta ferramenta é permitir a comparação entre os custos dos esquemas de alocação propostos pelos diversos algoritmos implementados. São quatro os algoritmos implementados: *Huang* [26], Solução Ótima, *Aloc* e *GRADA*. Os algoritmos *Aloc* e *GRADA* propõem variações do algoritmo *Huang* buscando encontrar

Algoritmo Hush (1 - 47,5Kp)

Algoritmo de Alocação

Quantidade de Nós: 4 Custo de transferir um pacote de dados: 0,032

Total de Transações: 4 Tamanho de um pacote de dados: 6250

Quantidade de Fragmentos: 4 Custo de construção do circuito virtual: 65,5676

Dimensiona Matrizes

Matriz de Frequência	Matriz de Custo de Comunicação	Vetor de Tamanho de Fragmento
Matriz de Acesso	Matriz de Atualização	Matriz de Seletividade

	F1	F2	F3	F4
T1	1	0	0	3
T2	0	1	0	0
T3	2	0	3	0
T4	0	4	0	5

Gerar Matriz de Alocação

Figura A.1: Dados de entrada

um esquema de alocação onde o custo de execução das aplicações seja menor. A Solução Ótima foi utilizada para validar quão próximos os resultados encontrados pelos algoritmos implementados se encontram da solução ótima para o problema.

A Figura A.2 representa *XAlloc*. Sobre as informações dos fragmentos, são executados os algoritmos de alocação utilizando uma função de custo para encontrar a melhor distribuição dos fragmentos nos nós da rede.

A.2 Preparação dos Dados

Os algoritmos consideram um conjunto de dados de entrada [32].

- A matriz de acesso representa a frequência de leitura que uma transação t realiza em um fragmento f a cada vez que é executada.
- A matriz de atualização representa a frequência em que uma transação t atualiza um fragmento f a cada vez que é executada.

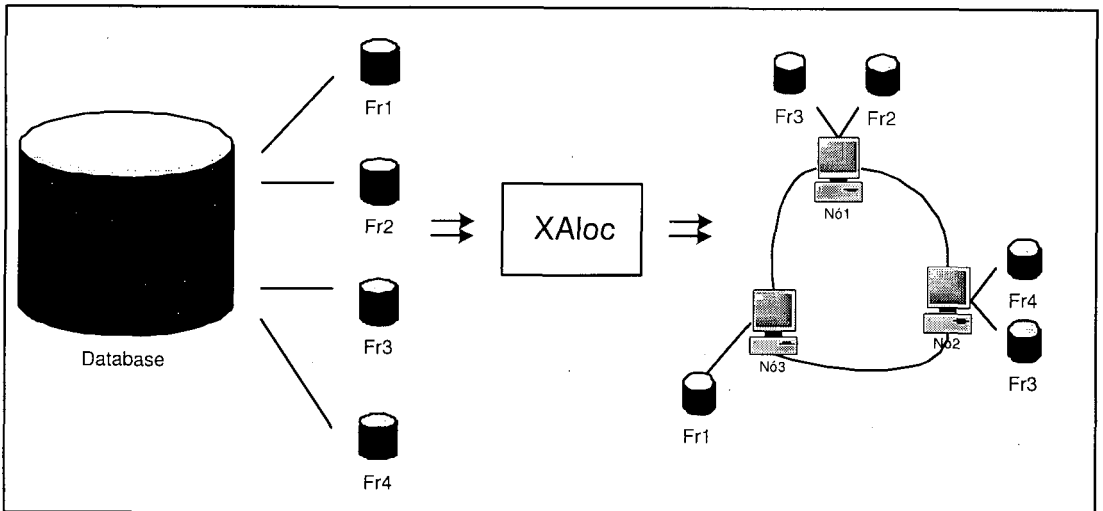


Figura A.2: Características do *XAloc*

- A matriz de seletividade representa a porção de dados de um fragmento f que é acessada por uma transação t .
- A matriz de frequência representa a frequência de execução de cada transação em cada nó.
- A matriz de custo de comunicação representa o custo de transferir uma unidade de dado entre dois nós da rede.
- E finalmente o vetor de tamanho dos fragmentos que representa o tamanho dos fragmentos.

As dimensões destas matrizes variam de acordo com o número de nós, fragmentos e transações também considerados como dados de entrada dos algoritmos.

A.3 Algoritmos

Os algoritmos implementados na ferramenta *XAloc* são os algoritmos *Huang*, *Aloc*, *GRADA* descritos nas Seções 3.2, 3.3 e 5.3. Além destes algoritmos, *XAloc* permite a execução de um algoritmo de busca exaustiva que propõe a solução ótima para o problema de alocação. A solução considerada ótima é definida como a solução viável com menor custo de alocação. Esta solução ótima é encontrada por um algoritmo de busca exaustiva. Pelo fato do algoritmo considerar todas as soluções viáveis

para o problema, torna-se impraticável sua utilização para problemas que envolvem um maior número de soluções por necessitar de uma capacidade de processamento muito grande. O objetivo de sua utilização é validar as soluções encontradas pelos algoritmos em problemas com menor número de soluções.

A.4 Função de custo

A ferramenta permite a comparação dos algoritmos de acordo com três tipos de função de custo (Figura A.3). A função de custo denominada HuCh implementa a função de custo proposta em [26] e definida na Seção 3.1. Esta função de custo considera que a atualização de um fragmento envolve o tráfego dos dados deste fragmento de um nó da rede para todos os outros nós que alocam este fragmento (transporte de dados [41]). Nesta função é desconsiderado o custo de envio de uma mensagem do nó que executa a transação de atualização para o nó de onde os dados do fragmento serão transferidos.

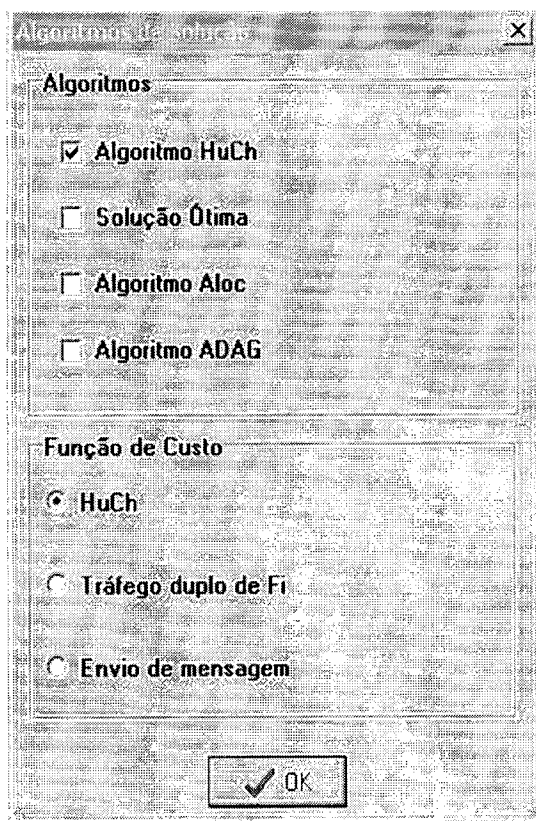


Figura A.3: Configuração da função de custo

A segunda função de custo, denominada Tráfego duplo de F_i , faz uma alteração da função de custo proposta em [26]. Esta função de custo considera que o nó que atualiza um determinado fragmento recupera seus dados, realiza sua atualização e transfere de volta ao nó de origem. Assim o custo de atualização é a soma do custo de recuperação dos dados do fragmento de cada nó que o aloca para o nó que executa a atualização e o custo de retornar estes dados atualizados para os nós que alocam este fragmento. Como a função de custo anterior, esta função de custo é denominada transporte de dados. Esta abordagem destaca o custo de transferência de dados.

Finalmente, a terceira função de custo implementa a atualização de fragmentos definida como transporte de mensagem [41]. Nesta abordagem, o custo de atualização é definido como o envio de uma mensagem do nó que executa a transação de atualização para cada nó que aloca uma cópia do fragmento. Esta abordagem destaca o custo de processamento local.

A.5 Funcionalidades

Esta seção descreve algumas propriedades da aplicação que facilitam a entrada de dados para os algoritmos e a manipulação dos resultados destes algoritmos.

Para facilitar o preenchimento dos parâmetros de entrada definidos na Seção A.2, a ferramenta apresenta algumas funcionalidades. As matrizes de entrada de um determinado problema podem ser salvas em um arquivo. Estes dados podem ser recuperados para re-execuções futuras de algum algoritmo. Estas matrizes podem ser geradas de forma aleatória de acordo com o número de nós, transações e fragmentos. Nesta funcionalidade, o custo de comunicação entre os nós da rede varia de 0,16 a 2,88 variando de 0,16.

Além destas propriedades, a ferramenta proposta permite salvar os resultados das matrizes de alocação dos passos 1, 2 e 3 de cada algoritmo em arquivos XML. A partir destes resultados é possível alterar, através de uma interface gráfica, alguma alocação de fragmento e recalcular o custo da nova alocação. Esta propriedade facilita a análise dos resultados alcançados pelos algoritmos.

Esta ferramenta foi desenvolvida em Delphi 5 e pode ser executada em Win-

dows 98, ME, 2000 ou XP.

Apêndice B

Implementação das funções do algoritmo de alocação por uma classe abstrata

```
type
matriz_int = array of array of word;
matriz_real = array of array of real;
vetor_int = array of word;
vetor_real = array of real;

{ Estrutura que armazena o custo/benefício de retirar fr do nó }
diferencas = record
  valor: real; { Diferença entre o benefício e o custo. }
  indice: smallint; { Índice do nó referente a diferença. }
end;

{ Classe com atributos e métodos comuns. }
TAlocacaoDados = class
protected
  { Protected declarations }
  oTr, { Total de Transações. }
  oNo, { Total de Nós. }
  oFr, { Total de fragmentos. }
  oCusto_trans, { Custo de transferir um pacote de dados. }
  oTam_Dados, { Tamanho de um pacote de dados. }
  oVCini: real; { Custo de criação do circuito virtual. }
  ORM, { Matriz de acessos. Tr por Fr. }
  OUM, { Matriz de atualização. Tr por Fr. }
  OFREQ, { Matriz de frequência. Tr por Nó. }
  OFAT: matriz_int; { Matriz de alocação. Fr por Nó. }
  OSEL, { Matriz de seletividade. Tr por Fr. }
  OCTR: matriz_real; { Matriz de custo de rede. Nó por Nó. }
```

```
oTAM: vetor_real; { Vetor com o tamanho de cada fragmento. }
oCusto_Tr, { Custa das transações de leitura. }
oCusto_Tu: real; { Custo das transações de atualização. }
```

```
(* * * * *
    Funções utilizadas para o cálculo do custo de alocação.
* * * * *
{ Cálculo do custo de transferir um fragmento entre dois nós. }
function Custo_Comunicacao(aCusto, aTAM: real): real;
{ Cálculo do custo das transações de leitura. }
function TR(aIndice_i, aIndice_k: word; aFAT: matriz_int): real;
{ Cálculo do menor custo de comunicação de tr de leitura. }
function Min_Custo_TR(aIndice_i, aIndice_j, aIndice_k: word;
    aFAT: matriz_int): real;
{ Cálculo do custo das transações de atualização. }
function TU(aIndice_i, aIndice_k: word; aFAT: matriz_int): real;
{ Limpa itens da TList. }
procedure Limpa_TList(var aLista: TList; aIndice: integer = 0);
(* * * * *
```

```
public
```

```
{ Public declarations }
```

```
{ Propriedades. }
```

```
property No_Count: integer read oNo;
property Fr_Count: integer read oFr;
property RM: matriz_int read oRM;
property UM: matriz_int read oUM;
property FREQ: matriz_int read oFREQ;
property FAT: matriz_int read oFAT write oFAT;
property SEL: matriz_real read oSEL;
property CTR: matriz_real read oCTR;
property Tempo: word read oTr_execut write oTr_execut;
property Custo_TR: real read oCusto_Tr;
property Custo_TU: real read oCusto_Tu;
```

```
{ Construtor da Classe. }
```

```
constructor create(aTr, aNo, aFr, aFunc_Custo: integer); virtual;
```

```
{ Destructor da Classe. }
```

```
destructor destroy; override;
```

```
{ Preenche matrizes. }
```

```
procedure SetInPut(aRM, aUM, aFREQ: matriz_int; aCTR, aSEL:
    matriz_real; aTAM: vetor_real; aTam_Dados, aCusto_Trans,
    aVCini: real); virtual;
```

```
{ Aplica função de custo para a FAT. }
```

```
function Custo_Total(aFAT: matriz_int): real;
```

```
end;
```

```

TBase = class(TAlocacao_Dados)
protected
  { Protected declarations }

  { Calcula o nó possui o menor custo de comunicacao para aNo. }
  function Menor_Delay(aFrag, aNo: word; aNo1: smallint = -1):
    smallint;
  { Calcula o nó possui o menor custo de comunicação para o fr. }
  function Min_Delay(aFrag: word; aUV: vetor_int): smallint;
  { Verifica se um Nó lê um fragmento. }
  function Verifica_Leitura(aFrag, aNo: word): boolean;
  { Cálculo do ganho com a remoção do fragmento do nó. }
  function Beneficio(aFrag, aNo: word): real;
  { Cálculo do perda com a remoção do fragmento do nó. }
  function Custo_Perm(aFrag, aNo: integer): real;
  { Calcula o número de alocações de um fragmento. }
  function NumFragAloc(aFrag: word): word;
  { Procura transações que atualizem aFrag. }
  function Trans_Atualiza_Frag(aFrag: word; var aUV: vetor_int):
    boolean;
  { Monta lista de candidatas a retirada. }
  procedure Monta_LCO(aFr: integer; var aLista: TList);
public
  { Public declarations }
end;

```

implementation

```

{ TAlocacao_Dados }
(* * * * * *)
* Função : Custo.Comunicacao *
*
* Objetivo : Calcula o custo de transferir fr entre dois nós. *
*
* Parâmetros : *
* aCusto - Custo de transferir uma unidade de dados entre os fr*
* aTam - Tamanho do fragmento a ser transferido *
*
* Comentários : *
*
* Retorno : *
* Custo de transferência do fragmento entre os nós *
*
(* * * * * *)
function TAlocacao_Dados.Custo.Comunicacao(aCusto, aTam: real):
  real;
begin
  Result := (Self.oCusto_trans * (aTam / Self.oTam_Dados)) +
    (aCusto * aTam);
end;

```

```

(* * * * *
* Função : Custo_Total
*
* Objetivo : Aplica a função de custo em FAT
* Parâmetros : aFAT - Matriz de alocação
*
* Comentários :
*
* Retorno :
* Custo de alocação se executado com sucesso
* -1 senao
*
* * * * *
function TAlocacao_Dados.Custo.Total(aFAT: matriz_int): real;
var vTR, { Custo das transacoes de leitura. }
    vTU, { Custo das transações de atualização. }
    vCCproc: real; { Custo de processar a transação. }
    k, { Conjunto de nós pertencentes a S. }
    i: word; { Conjunto de transações pertencentes a T. }
begin
    try
        vCCproc := 0;
        for k := 0 to Self.oNo - 1 do begin
            for i := 0 to Self.oTr - 1 do begin
                vTR := TR(i, k, aFAT);
                Application.ProcessMessages;
                vTU := TU(i, k, aFAT);
                Application.ProcessMessages;
                if vTR = -1 then begin
                    raise Exception.Create('Situação inesperada! Custo das
                        transações de leitura na função de custo.')
                end
                else
                    if vTU = -1 then begin
                        raise Exception.Create('Situação inesperada! Custo das
                            transações de atualização na função de custo.')
                    end
                    else begin
                        vCCproc := vCCproc + (Self.oFREQ[i, k] * (vTR + vTU +
                            Self.oVCini));
                        Self.oCusto_Tr := Self.oCusto_Tr + vTR;
                        Self.oCusto_Tu := Self.oCusto_Tu + vTU;
                    end;
                end;
            end;
        end;
    except
        on e: Exception do begin
            vCCproc := -1;
        end;
    end;
end;

```

```
Result := vCCproc;
end;
```

```
(* * * * *
* Função : TR
*
* Objetivo : Calculo do custo das transações de leitura
*
* Parâmetros :
* aIndice_i - Índice i (Transações) da função de custo
* aIndice_k - Índice k (Nó) da função de custo
* aFAT - Matriz de alocação
*
* Comentários :
*
* Retorno :
* Custo se executado com sucesso
* -1 se algum erro ocorreu
*)
```

```
function TAlocacao_Dados.TR(aIndice_i, aIndice_k: word; aFAT:
    matriz_int): real;
var vCusto, { Custo das transações de leitura. }
    vMin_Custo: real; { Menor custo de comunicação. }
    j: word; { Conjunto de fragmentos pertencentes a F. }
begin
    vCusto := 0;
    try
        for j := 0 to oFr - 1 do begin
            vMin_Custo := Min_Custo.TR(aIndice_i, j, aIndice_k, aFAT);
            if vMin_Custo = -1 then
                vCusto := -1
            else begin
                vCusto := vCusto + oRM[aIndice_i, j] * vMin_Custo;
            end;
            Application.ProcessMessages;
        end;
    except
        vCusto := -1;
    end;
    Result := vCusto;
end;
```

```
(* * * * *
* Função : Min_Custo_TR
*
* Objetivo : Calcula o menor custo para execução de tr de RM
*
* Parâmetros :
```



```

* aIndice_i - Índice i (Transações) da função de custo          *
* aIndice_j - Índice j (Fragmentos) da função de custo         *
* aIndice_k - Índice k (Nó) da função de custo                 *
* aFAT - Matriz de alocação                                     *
*                                                                *
* Comentários :                                               *
*                                                                *
* Retorno :                                                    *
* Menor Custo se executado com sucesso                          *
* -1 se algum erro ocorreu                                     *
*                                                                *
*)

```

```

function TAlocacaoDados.Min_Custo_TR(aIndice_i, aIndice_j,
          aIndice_k: word; aFAT: matriz_int): real;
var vMin_Custo: real; { Valor do menor custo. }
    s: word; { Nó que aloca o fragmento aIndice_i. }
    vCusto: real; { Custo a cada loop. }
begin
  Min_Custo := -1;
  try
    { Para cada Nó. }
    for s := 0 to oNo - 1 do begin
      { Se o fragmento aIndice_j está alocado nele. }
      if aFAT[aIndice_j, s] = 1 then begin
        { Calcula o custo. }
        vCusto := Custo_Comunicacao(Self.oCTR[aIndice_k, s],
          (Self.oSEL[aIndice_i, aIndice_j] / 100) *
          Self.oTAM[aIndice_j]);
        if vMin_Custo <> -1 then begin
          if vCusto < vMin_Custo then
            vMin_Custo := vCusto
          end
        else
          vMin_Custo := vCusto;
        end;
        Application.ProcessMessages;
      end;
    { Se vMin_Custo = -1 então fr não foi alocado em nenhum nó. }
    if vMin_Custo = -1 then
      vMin_Custo := 0
    except
      vMin_Custo := -1;
    end;
    Result := vMin_Custo;
  end;
end;

```

```

( * * * * * )
* Função : TU *
* * * * * *

```

```

* Objetivo : Calculo do custo das transações de atualização *
*
* Parâmetros : *
* aIndice_i - Índice i (Transações) da função de custo *
* aIndice_k - Índice k (Nó) da função de custo *
* aFAT - Matriz de alocação *
*
* Comentários : *
*
* Retorno : *
* Custo se executado com sucesso *
* -1 se algum erro ocorreu *
*
* * * * *
function TAlocacaoDados.TU(aIndice_i, aIndice_k: word; aFAT:
    matriz_int): real;
var vCusto: real; { Custo das transações de atualização. }
    l, { Nós que alocam o fragmento j. }
    j: word; { Conjunto de fragmentos pertencentes a F. }
begin
    vCusto := 0;
    try
        for j := 0 to Self.oFr - 1 do begin
            for l := 0 to Self.oNo - 1 do begin
                vCusto := vCusto + (Self.oUM[aIndice_i, j] * aFAT[j, l] *
                    Custo Comunicacao(Self.oCTR[aIndice_k, l],
                    (Self.oSEL[aIndice_i, j] / 100) * Self.oTAM[j]));
            end;
            Application.ProcessMessages;
        end;
    except
        vCusto := -1;
    end;
    Result := vCusto;
end;

(* * * * *
* Procedure: Limpa_TList *
*
* Objetivo : Liberar espaço de memoria dos itens de uma TList *
*
* Parâmetros : aLista - TList que se deseja apagar elementos *
* aIndice - Índice do primeiro elemento a ser apagado. *
*
* Comentários : *
*
* Retorno : *
*
* * * * *

```

```

procedure TAlocacaoDados.Limpa_TList(var aLista: TList; aIndice:
    integer);
var i: integer;
    vDiferenca: ^Diferencas;
begin
    for i := aLista.Count - 1 downto aIndice do begin
        vDiferenca := aLista.Items[i];
        Dispose(vDiferenca);
    end;
end;

```

```

(* * * * * *)
* Procedure : Create *
*
* Objetivo : Inicia a classe *
*
* Parâmetros : *
* aTr - Número de Transações *
* aNo - Número de Nós *
* aFr - Número de fragmentos *
*
* Comentários : *
*
(* * * * * *)

```

```

constructor TAlocacaoDados.create(aTr, aNo, aFr, aFunc_Custo:
    integer);

```

```

begin
    Self.oTr_execut := 0;
    Self.oCusto_Tr := 0;
    Self.oCusto_Tu := 0;
    Self.oTr := aTr;
    Self.oNo := aNo;
    Self.oFr := aFr;
    Self.oFunc_Custo := aFunc_Custo;
    SetLength(Self.oRM, aFr, aTr);
    SetLength(Self.oUM, aFr, aTr);
    SetLength(Self.oSEL, aFr, aTr);
    SetLength(Self.oFREQ, aNo, aTr);
    SetLength(Self.oCTR, aNo, aNo);
    SetLength(Self.oFAT, aFr, aNo);
    SetLength(Self.oTAM, aFr);
end;

```

```

(* * * * * *)
* Procedure: Destroy *
*
* Objetivo : Destructor padrão da classe. *
*
* Parâmetros : *

```

```

*                                                                 *
* Comentários :                                                                 *
*                                                                 *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
destructor TAlocacao_Dados.destroy; begin
    inherited destroy;
end;

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
* Procedure : SetInPut                                                                 *
*                                                                 *
* Objetivo : Preenche valores das matrizes.                                         *
*                                                                 *
* Parâmetros :                                                                 *
* aRM - Matriz de Acesso                                                             *
* aUM - Matriz de Atualização                                                         *
* aSEL - Matriz de Seletividade                                                       *
* aFREQ - Matriz de Frequência                                                         *
* aCTR - Matriz de Custo de Comunicação                                              *
* aTAM - Vetor com tamanho de cada fragmento                                         *
* aTam_Dados - Tamanho do pacote de dados transferido na rede                       *
* aCusto_Trans - Custo de transferir um pacote de dados                             *
* aVCini - Custo de criação do circuito virtual                                     *
*                                                                 *
* Comentários :                                                                 *
*                                                                 *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
procedure TAlocacao_Dados.SetInPut(aRM, aUM, aFREQ: matriz_int;
    aCTR, aSEL: matriz_real; aTAM: vetor_real; aTam_Dados,
    aCusto_Trans, aVCini: real);
begin
    Self.oRM := aRM;
    Self.oUM := aUM;
    Self.oSEL := aSEL;
    Self.oFREQ := aFREQ;
    Self.oCTR := aCTR;
    Self.oTAM := aTAM;
    Self.oTam_Dados := aTam_Dados;
    Self.oCusto_trans := aCusto_Trans;
    Self.oVCini := aVCini;
end;

{ TBase }
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
* Função : Menor_Delay                                                                 *
*                                                                 *
* Objetivo : Calculo do Nó que possui menor ou segundo menor                       *
* custo para transferir aFrag para aNo                                              *
*                                                                 *
*                                                                 *

```

```

* Parâmetros :
* aFrag - Índice do fragmento a ser transferido
* aNo - Índice do nó para o qual o fragmento será transferido
* aNo1 - Índice do nó de menor custo de transferência
*
* Comentários : O retorno pode ser aNo.
* Se aNo1 for -1 ele encontra o primeiro senão o segundo
*
* Retorno :
* -1 Se o fr não foi alocado em nenhum nó ou somente em aNo1
* Índice do nó com menor custo de transferência
*
* * * * *
function TBase.Menor_Delay(aFrag, aNo: word; aNo1: smallint):
    smallint;
var i: smallint; { Nó que armazena aFrag. }
    vMenor_Delay: smallint; { Nó com o menor custo. }
    vCusto: real; { Valor do menor custo. }
begin
    vMenor_Delay := -1;
    vCusto := -1;
    { Para todo nó. }
    for i := 0 to oNo - 1 do begin
        { Se o fragmento está alocado no nó. }
        if Self.oFAT[aFrag, i] = 1 then
            { Se custo > -1 este não é o 1º nó que possui frag. }
            if vCusto > -1 then begin
                if Self.oCTR[aNo, i] < vCusto then
                    { Se aNo1 = i significa que i é o primeiro. }
                    if (aNo1 <> i) then begin
                        vCusto := Self.oCTR[aNo, i];
                        vMenor_Delay := i;
                    end
                end
            else { O custo não pode ser comparado com vcusto. }
                if (aNo1 <> i) then begin
                    vCusto := Self.oCTR[aNo, i];
                    vMenor_Delay := i;
                end;
            end;
        if vCusto > -1 then
            Result := vMenor_Delay
        else
            Result := -1;
        end;
    end;

(* * * * *
* Função : Min_Delay
*

```

```

* Objetivo : Calcula o Nó com menor custo de atualização ao      *
* alocar o fragmento aFrag.                                     *
*                                                             *
* Parâmetros :                                               *
* aFrag - Índice do fragmento a ser armazenado                *
* aUV - Vetor com nós que atualizam aFrag                     *
*                                                             *
* Comentários :                                             *
*                                                             *
* Retorno :                                                  *
* -1 - Se fragmento não puder ser alocado em nenhum nó      *
* Índice do nó com menor custo de transferência              *
*                                                             *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
function TBase.Min_Delay(aFrag: word; aUV: vetor_int): smallint;
var vIndice: smallint; { Índice do Nó com menor custo. }
    i, { Nó que atualiza aFrag. }
    j, { Transação de atualiza aFrag. }
    k: word; { Conjunto de nós pertencentes a S. }
    vCusto, { Custo a cada iteração. }
    vMin_Custo: real; { Menor custo. }
begin
    { Calcula o menor custo. }
    vMin_Custo := 0;
    vIndice := -1;
    { Para todo nó. }
    for k := 0 to Self.oNo - 1 do begin
        { Se o nó atualiza aFrag. }
        if aUV[k] = 1 then begin
            vCusto := 0;
            { Calcula o custo de atualizar aFrag alocado em k. }
            for i := 0 to Self.oNo - 1 do
                for j := 0 to Self.oTr - 1 do
                    vCusto := vCusto + (Self.oUM[j, aFrag] * Self.oFREQ[j, i]
                        * Custo_Comunicacao(Self.oCTR[k, i], (Self.oSEL[j,
                            aFrag])/100) * Self.oTAM[aFrag]));
                if vMin_Custo > 0 then begin
                    if vCusto < vMin_Custo then begin
                        vMin_Custo := vCusto;
                        vIndice := k;
                    end;
                end
            else begin
                vMin_Custo := vCusto;
                vIndice := k;
            end;
        end;
    end;
    Result := vIndice;
end;

```

```

(* * * * *
* Função : VerificaLeitura
*
* Objetivo : Verifica se um Nó lê um fragmento.
*
* Parâmetros :
* aFrag - Índice do fragmento em questão
* aNo - Índice do Nó em questão
*
* Comentários :
*
* Retorno :
* true - se aNo lê aFrag
* false - se aNo não lê aFrag
*
* * * * *
)

```

```

function TBase.VerificaLeitura(aFrag, aNo: word): boolean;
var i: integer; { Conjunto de transações pertencentes a T. }
begin
  Result := false;
  for i := 0 to oTr - 1 do
    if oFREQ[i, aNo] > 0 then begin
      { Se a transação acesso o nó com uma freq acima de 0. }
      if (oRM[i, aFrag] * oFREQ[i, aNo]) > 0 then begin
        Result := true;
        break;
      end;
    end;
  end;
end;

```

```

(* * * * *
* Função : Beneficio
*
* Objetivo : Calculo do ganho da remoção do fragmento do nó.
*
* Parâmetros :
* aFrag - Índice do fragmento em questão
* aNo - Índice do Nó em questão
*
* Comentários :
*
* Retorno :
* 0 - se algum erro ocorreu
* Beneficio - Se executado com sucesso
*
* * * * *
)

```

```

function TBase.Beneficio(aFrag, aNo: word): real;
var vBem: real; { Valor do benefício. }
    i, { Transação que atualiza aFrag. }

```

```

k: word; { Nó que atualiza aFrag. }
begin
  try
    vBem := 0;
    for k := 0 to Self.oNo - 1 do
      for i := 0 to Self.oTr - 1 do begin
        vBem := vBem + (Self.oFREQ[i, k] * Self.oUM[i, aFrag] *
          Custo.Comunicacao(Self.oCTR[k, aNo], (Self.oSEL[i,
            aFrag] / 100) * Self.oTAM[aFrag]));
      end;
      Result := vBem;
    except
      Result := 0;
    end;
  end;
end;

```

```

( * * * * *
* Função : Custo_Perm *
* * * * *
* Objetivo : Calculo do perda da remoção do fragmento do nó. *
* * * * *
* Parâmetros : *
* aFrag - Índice do fragmento em questão *
* aNo - Índice do Nó em questão *
* * * * *
* Comentários : *
* * * * *
* Retorno : *
* 0 - se algum erro ocorreu *
* Custo - Se executado com sucesso *
* * * * *
* * * * *

```

```

function TBase.Custo_Perm(aFrag, aNo: integer): real;
var vCusto: real; { Valor do custo. }
    i, { Transações que lêem aFrag. }
    k: word; { Conjunto de nós pertencentes a S. }
    vNo: smallint; { Nó com menor Delay. }
    vTot1, { T1 do algoritmo. }
    vTot2: real; { T2 do algoritmo. }
begin
  try
    vCusto := 0;
    { Para cada nó. }
    for k := 0 to oNo - 1 do begin
      vTot1 := 0;
      vTot2 := 0;
      { Se o nó lê o fragmento. Esta verificação não é feita pelo
        algoritmo Huang. É usada para otimizar, pois somente nós que
        lêem o fragmento terão vTot1 e vTot2 > 0. }
    end;
  end;
end;

```



```

if Verifica_Leitura(aFrag, k) then begin
  vNo := Menor_Delay(aFrag, k);
  { Se o fragmento está alocado em algum nó e o nó de menor
  custo é aNo. }
  if (vNo > -1) and (vNo = aNo) then begin
    for i := 0 to oTr - 1 do begin
      vTot1 := vTot1 + (Self.oFREQ[i, k] * Self.oRM[i, aFrag]
        * Custo_Comunicacao(Self.oCTR[k, vNo], (Self.oSEL[i,
        aFrag] / 100) * Self.oTAM[aFrag]));
    end;
    { Segundo menor custo de alocação. }
    vNo := Menor_Delay(aFrag, k, vNo);
    { Se vNo = -1 significa que o fragmento não está alocado em
    nenhum nó ou em so um. Isto não acontece pq esta função só
    é chamada se o número de fragmentos alocados for maior que
    1. Mesmo assim o teste é feito para garantir. }
    if vNo > -1 then begin
      for i := 0 to Self.oTr - 1 do begin
        vTot2 := vTot2 + (Self.oFREQ[i, k] * Self.oRM[i, aFrag]
          * Custo_Comunicacao(Self.oCTR[k, vNo], (Self.oSEL[i,
          aFrag] / 100) * Self.oTAM[aFrag]));
      end;
    end;
    else
      vTot2 := vTot1;
      vCusto := vCusto + (vTot2 - vTot1);
    end;
  end;
end;
Result := vCusto;
except
  Result := 0;
end;
end;

```

```

(* * * * *
* Função : NumFragAloc *
*
* Objetivo : Calcula o número de alocações de um fragmento. *
*
* Parâmetros : *
* aFrag - Índice do fragmento em questão *
*
* Comentários : *
* aFrag in (0 .. oFr - 1) *
*
* Retorno : *
* 0 - se algum erro ocorreu *
* Réplicas - Se executado com sucesso *

```

```

*
* * * * *
function TBase.NumFragAloc(aFrag: word): word;
var vReplicas, { Número de réplicas. }
    k: word; { Conjunto de nós que pertencem a S. }
begin
    try
        vReplicas := 0;
        for k := 0 to oNo do begin
            if Self.oFAT[aFrag, k] = 1 then
                inc(vReplicas);
        end;
        Result := vReplicas;
    except
        Result := 0;
    end;
end;

```

```

(* * * * *
* Função : Trans_Atualiza_Frag
*
* Objetivo : Procurar transações que atualizam aFrag.
*
* Parâmetros :
* aFrag - Índice do fragmento em questão
* aUV - Vetor com os nós que atualizam aFrag
*
* Comentários :
*
* Retorno :
* True - se existe
* False - senão
*
* * * * *

```

```

function TBase.Trans_Atualiza_Frag(aFrag: word; var aUV:
    vetor_int): boolean;
var vExiste: boolean; { Indica se existe tr que atualiza aFrag. }
    i, { Conjunto de transações pertencentes a T. }
    j: word; { Conjunto de nós pertencentes a S. }
begin
    { Monta vetor com todos os nós que atualizam aFrag. }
    SetLength(aUV, Self.oNo);
    for j := 0 to Self.oNo - 1 do
        aUV[j] := 0;
    vExiste := False;
    for i := 0 to Self.oTr - 1 do begin
        if oUM[i, aFrag] > 0 then
            for j := 0 to Self.oNo - 1 do begin
                if Self.oFREQ[i, j] > 0 then begin

```



```
    inc(k);  
end;  
end;  
aLista.Sort(ComparaçãO);  
end;
```

Apêndice C

Implementação do algoritmo Huang pela Classe THuang

```
(* * * * *
* Unit : uHuang
*
* Objetivo : Classe que implementa a solução Huang
*
* Comentários:
* No passo 2 a LCO é reordenada a cada retirada
* * * * * *)
type
THuang = class(TBase)
private
{ Private declarations }

public
{ Public declarations }

{ Método Construtor. }
constructor create(aTr, aNo, aFr, aFunc_Custo: integer);
override;
{ Executa o passo 1 do algoritmo. }
function ExecPasso_1: string;
{ Executa o passo 2 do algoritmo. }
function ExecPasso_2: string;
{ Executa o passo 3 do algoritmo. }
function ExecPasso_3: string;
end;

implementation
{ THuang }
constructor THuang.create(aTr, aNo, aFr, aFunc_Custo: integer);
begin
```

```
inherited;
end;
```

```
(* * * * *
* Função : ExecPasso_1
*
* Objetivo : Executa passo 1 do algoritmo.
*
* Parâmetros :
*
* Comentários :
* Inicia a melhor solução para a recuperação. Com replicação.
*
* Retorno :
* OK - Se executado com sucesso
* E.message - se algum erro ocorreu
*
* * * * * *)
```

```
function THuang.ExecPasso_1: string;
var i, { Conjunto de nós pertencentes a S. }
    j, { Conjunto de transações pertencentes a T. }
    k: integer; { Conjunto de fragmentos pertencentes a F. }
begin
    Self.oCusto_Tr := 0;
    Self.oCusto_Tu := 0;
    try
        for i := 0 to oNo - 1 do begin
            for j := 0 to oTr - 1 do
                if oFREQ[j, i] > 0 then
                    for k := 0 to oFr - 1 do
                        { Se tr acessa o nó com uma freqüência acima de 0. }
                        if (oRM[j, k] * oFREQ[j, i]) > 0 then begin
                            oFAT[k, i] := 1;
                        end;
                    end;
                end;
            result := 'OK';
        except
            on E: Exception do
                result := E.Message;
            end;
        end;
    end;
```

```
(* * * * *
* Função : ExecPasso_2
*
* Objetivo : Executa passo 2 do algoritmo.
*
* Parâmetros :
*
* * * * *
```

```

* Comentários :
* Reduz o número de réplicas analisando o custo e benefício da
* permanência do fragmento no nó.
* A lista de cópias candidatas é reordenada a cada retirada.
*
* Retorno :
* OK - Se executado com sucesso
* E.message - se algum erro ocorreu
*
* * * * *
function THuang.ExecPasso.2: string;
var j: word; { Conjunto de fragmentos pertencentes a F. }
    vDiferenca: ^diferencas; { custo/benefício da retirada do nó.}
    vDiferencas: TList; { Lista de vDiferenca. }
begin
    Self.oCusto.Tr := 0;
    Self.oCusto.Tu := 0;
    try
        vDiferencas := TList.Create;
        try
            { Para cada fragmento. }
            for j := 0 to oFr - 1 do begin
                Monta_LCO(j, vDiferencas);
                while (NumFragAloc(j) > 1) and (vDiferencas.Count > 0) do
                    begin
                        vDiferenca := vDiferencas.Items[0];
                        oFAT[j, vDiferenca^.indice] := 0;
                        Self.Limpa_TList(vDiferencas);
                        vDiferencas.Clear;
                        Monta_LCO(j, vDiferencas);
                    end;
                Self.Limpa_TList(vDiferencas);
                vDiferencas.Clear;
            end;
        finally
            Self.Limpa_TList(vDiferencas);
            vDiferencas.Clear;
            vDiferencas.Free;
        end;
        result := 'OK';
    except
        on E: Exception do
            result := E.Message;
        end;
    end;
end;

(* * * * *
* Função : ExecPasso.3
*

```

```

* Objetivo : Executa passo 3 do algoritmo. *
* *
* Parâmetros : *
* *
* Comentários : *
* Faz alocação de fragmentos não alocados nos passos anteriores*
* *
* Retorno : *
* OK - Se executado com sucesso *
* E.message - se algum erro ocorreu *
* *
* * * * * *)
function THuang.ExecPasso_3: string;
var j: word; { Conjunto de fragmentos pertencentes a F. }
    k: Smallint; { Nó com menor Delay. }
begin
    Self.oCusto_Tr := 0;
    Self.oCusto_Tu := 0;
    try
        k := 0;
        for j := 0 to oFr - 1 do begin
            if (NumFragAloc(j) = 0) then begin
                oFAT[j, k] := 1
                if k = oNo then
                    k := 0
                else
                    inc(k);
            end;
        end;
        result := 'OK';
    except
        on E: Exception do
            result := E.Message;
        end;
    end;
end;

```

Apêndice D

Implementação do algoritmo Aloc pela Classe TAlloc

```
(* * * * *
* Unit :   uAlloc
*
* Objetivo : Classe que implementa a alocação dos fragmentos
* segundo transações de leitura e atualização.
*
* Comentários:
* A lista de cópias candidatas é reordenada a cada retirada no
* passo 2
*
* * * * *
type
TAlloc = class(TBase)
  private
    { Private declarations }

  public
    { Public declarations }

    { Método Construtor. }
    constructor create(aTr, aNo, aFr, aFunc.Custo: integer);
      override;
    { Executa o passo 1 do algoritmo. }
    function ExecPasso.1: string;
    { Executa o passo 2 do algoritmo. }
    function ExecPasso.2: string;
    { Executa o passo 3 do algoritmo. }
    function ExecPasso.3: string;
end;

implementation
```

```

{ TAlloc }
constructor TAlloc.create(aTr, aNo, aFr, aFunc_Custo: integer);
begin
  inherited;
end;

(* * * * *
* Função : ExecPasso_1
*
* Objetivo : Executa passo 1 do algoritmo.
*
* Parâmetros :
*
* Comentários :
* Inicia a melhor solução para a recuperação. Com replicação.
*
* Retorno :
* OK - Se executado com sucesso
* E.message - se algum erro ocorreu
*
* * * * *
function TAlloc.ExecPasso_1: string;
var i, { Conjunto de nós pertencentes a S. }
    j, { Conjunto de transações pertencentes a T. }
    k: integer; { Conjunto de fragmentos pertencentes a F. }
begin
  Self.oCusto_Tr := 0;
  Self.oCusto_Tu := 0;
  try
    for i := 0 to oNo - 1 do begin
      for j := 0 to oTr - 1 do
        if oFREQ[j, i] > 0 then begin
          for k := 0 to oFr - 1 do begin
            { Se a tr acesso o nó com uma freqüência acima de 0. }
            if (oRM[j, k] * oFREQ[j, i]) > 0 then begin
              oFAT[k, i] := 1;
            end;
            { Se a tr atualiza o nó com uma freqüência acima de 0. }
            if (oUM[j, k] * oFREQ[j, i]) > 0 then begin
              oFAT[k, i] := 1;
            end;
          end;
        end;
      end;
    end;
    result := 'OK';
  except
    on E: Exception do
      result := E.Message;
  end;
end;

```



```
end;  
end;
```

```
(* * * * *  
* Função : ExecPasso_3 *  
* * * * *  
* Objetivo : Executa passo 3 do algoritmo. *  
* * * * *  
* Parâmetros : *  
* * * * *  
* Comentários : *  
* Faz alocação de fragmentos não alocados nos passos anteriores*  
* * * * *  
* Retorno : *  
* OK - Se executado com sucesso *  
* E.message - se algum erro ocorreu *  
* * * * *  
* * * * * *)
```

```
function TAlloc.ExecPasso_3: string;  
var vUV: vetor_int; { Vetor com nós que atualizam um determinado  
                    nó. }  
    j: word; { Conjunto de fragmentos pertencentes a F. }  
    k: Smallint; { Nó com menor Delay. }  
begin  
    Self.oCusto_Tr := 0;  
    Self.oCusto_Tu := 0;  
    try  
        for j := 0 to oFr - 1 do begin  
            if (NumFragAloc(j) = 0) and (TransAtualizaFrag(j, vUV)) then  
                begin  
                    k := MinDelay(j, vUV);  
                    if (k > -1) then begin  
                        oFAT[j, k] := 1  
                    end  
                    else begin  
                        raise Exception.Create('Situação inesperada!  
                                                Passo 3 do algoritmo.')                    end;  
                end;  
            end;  
        end;  
        result := 'OK';  
    except  
        on E: Exception do  
            result := E.Message;  
        end;  
    end;  
end;  
end.
```

Apêndice E

Implementação do algoritmo GRADA pela Classe TGRADA

```
(* * * * *
* Unit : uGRASP
*
* Objetivo : Classe que implementa a utilização do GRASP
*
* Comentários:
* Utiliza a LCO e LFCR.
* * * * * *)
type
TGRADA = class(TBase)
  private
    { Private declarations }
    { Propriedade utilizada para encontrar o menor custo. }
    oCusto_Total: real;
    { Monta lista de cópias ordenadas. }
    procedure Monta_Lista(aFr: integer; var aLista: TList);
    { Monta Lista de Fragmentos Candidatos a Retirada. }
    procedure Monta_LFCR(aFr: integer; var aLFCR: TList);
    { Define o valor de K. }
    function Define_K(aLCO:TList): smallint;
    { Realiza a Busca Local do GRADA. }
    procedure Busca_Local(aCusto_Total: real; aFat_1: matriz_int; var
      aFat: matriz_int);
    { Atualiza FAT para a Busca Local. }
    procedure Atualiza_FAT(aFat_1: matriz_int);

  public
    { Public declarations }

    { Método Construtor. }
    constructor create(aTr, aNo, aFr, aFunc_Custo: integer);
```

```

        override;
    { Executa o passo 1 do algoritmo. }
    function ExecPasso.1: string;
    { Executa o passo 2 do algoritmo. }
    function ExecPasso.2: string;
end;

implementation
{ TGRADA }
constructor TGRADA.create(aTr, aNo, aFr, aFunc_Custo: integer);
begin
    inherited;
end;

```

```

(* * * * *
* Função : ExecPasso.1
*
* Objetivo : Executa passo 1 do algoritmo.
*
* Parâmetros :
*
* Comentários :
* Inicia a melhor solução para a recuperação. Com replicação.
*
* Retorno :
* OK - Se executado com sucesso
* E.message - se algum erro ocorreu
*
* * * * *
)

```

```

function TGRADA.ExecPasso.1: string;
var i, { Conjunto de nós pertencentes a S. }
    j: integer; { Conjunto de transações pertencentes a T. }
begin
    Self.oCusto.Tr := 0;
    Self.oCusto.Tu := 0;
    try
        for i := 0 to oNo - 1 do begin
            for j := 0 to oFr - 1 do
                oFAT[j, i] := 1;
            end;
            result := 'OK';
        except
            on E: Exception do
                result := E.Message;
            end;
        end;
end;

```

```

(* * * * *

```

```

* Função : ExecPasso_2
*
* Objetivo : Executa passo 2 do algoritmo.
*
* Parâmetros :
*
* Comentários :
* Reduz o número de réplicas analisando o custo e benefício da
* permanência do fragmento no nó. Utiliza um fator aleatório
* para a escolha da réplica a ser retirada
*
* Retorno :
* OK - Se executado com sucesso
* E.message - se algum erro ocorreu
*
* * * * *
function TGRADA.ExecPasso_2: string;
var j, { Conjunto de fragmentos pertencentes a F. }
    k: word; { Conjunto de nós pertencentes a S. }
    vDiferenca: ^diferencas; { Custo/benefício da retirada do nó.}
    vDiferencas: TList; { Lista de vDiferencas. }
    vFat_1, { Fat que armazena a alocação inicial do passo 1. }
    vFat: matriz_int; { Armazena a alocação com menor custo. }
    vCusto_Total: real; { Menor custo de alocação. }
    vIteracoes: word; { Iteracoes do GRASP. }
begin
    Self.oCusto_Tr := 0;
    Self.oCusto_Tu := 0;
    SetLength(vFAT, Self.oFr, Self.oNo);
    SetLength(vFAT_1, Self.oFr, Self.oNo);
    randomize;
    { Inicia o custo como o custo da matriz resultante do passo 1. }
    Self.oCusto_Total := Self.Custo_Total(Self.oFAT);
    { Armazena a alocação inicial em matriz temporária. }
    for j := 0 to Self.oFr - 1 do begin
        for k := 0 to Self.oNo - 1 do begin
            vFAT_1[j, k] := Self.oFAT[j, k];
        end;
    end;
    try
        vDiferencas := TList.Create;
        try
            for vIteracoes := 1 to 20 do begin
                { Início do passo 2. }
                { Para cada fragmento. }
                for j := 0 to oFr - 1 do begin
                    Self.Monta_LFCR(j, vDiferencas);
                end;
                try
                    { Elimina uma cópia aleatória de LFCR. }
                    while (NumFragAloc(j) > 1) and (vDiferencas.Count > 0) do

```

```

begin
    k := Random(vDiferencas.Count - 1);
    vDiferenca := vDiferencas.Items[k];
    oFAT[j, vDiferenca^.indice] := 0;
    Self.Limpa_TList(vDiferencas);
    vDiferencas.Clear;
    { Remonta LFCR. }
    Self.Monta_LFCR(j, vDiferencas);
end;
finally
    Self.Limpa_TList(vDiferencas);
    vDiferencas.Clear;
end;
{ Fim do passo 2. }
{ Verifica se uma melhor solução foi encontrada. }
vCusto_Total := Self.Custo_Total(Self.oFat);
if vCusto_Total < Self.oCusto_Total then begin
    { Atualiza a matriz temporária. }
    for j := 0 to Self.oFr - 1 do begin
        for k := 0 to Self.oNo - 1 do begin
            vFAT[j, k] := Self.oFAT[j, k];
        end;
    end;
    Self.oCusto_Total := vCusto_Total;
    { Realiza a Busca Local. }
    Self.Busca_Local(vCusto_Total, vFat_1, vFat);
end;
{ Salva FAT do passo 1. }
{ Reinicia oFAT. }
for j := 0 to Self.oFr - 1 do begin
    for k := 0 to Self.oNo - 1 do begin
        Self.oFAT[j, k] := vFAT_1[j, k];
    end;
end;
end;
for j := 0 to Self.oFr - 1 do begin
    for k := 0 to Self.oNo - 1 do begin
        Self.oFAT[j, k] := vFAT[j, k];
    end;
end;
finally
    Self.Limpa_TList(vDiferencas);
    vDiferencas.Clear;
    vDiferencas.Free;
end;
result := 'OK';
except
    on E: Exception do
        result := E.Message;
end;
end;

```



```

* aLFCR - Lista com as cópias ordenadas
*
* Comentários : LFCR é formada pelos K primeiros elementos de
* LCO
*
* Retorno : LFCR
*
* * * * *
procedure TGRADA.Monta_LFCR(aFr: integer; var aLFCR: TList);
var i,
    k: word; { Número de cópias de aLCO que formaram LFCR. }
    vDiferenca: ^diferencas; { Custo/benefício da retirada do nó.}
    vLCO: TList; { Lista de vDiferencas. }
begin
    { Monta LCO. }
    vLCO := TList.Create;
    Self.Monta_LCO(aFr, vLCO);
    try
        if vLCO.Count > 0 then begin
            K := Self.Define_K(vLCO);
            for i := 0 to k - 1 do begin
                vDiferenca := vLCO.Items[i];
                aLFCR.Add(vDiferenca);
            end;
        end
    finally
        Self.Limpa_TList(vLCO, k);
        vLCO.Clear;
        vLCO.Free;
    end;
end;

(* * * * *
* Função : Define_k
*
* Objetivo : Define K (Número de cópias de fragmentos que será
* retirado para cada fragmento.
*
* Parâmetros : aLCO - Lista de Cópias do fragmento Ordenada
*
* Comentários :
* K igual a metade do número de réplicas do fragmento
* K = (LCO.length/2)
*
* Retorno :
* K = Se executado com sucesso
* -1 = se algum erro ocorreu
*
* * * * *)

```

```

function TGRADA.Define_K(aLCO:TList): smallint;
var k: word; { Retorno da função. }
begin
  try
    k := aLCO.Count div 2;
    if k > 0 then
      Result := k
    else
      Result := 1;
  except
    Result := -1;
  end;
end;

```

```

(* * * * *
* Procedure: Busca_Local *
*
* Objetivo : Realiza a busca local do GRADA procurando a melhor*
* solução dentro da vizinhança definida. *
*
* Parâmetros : aCusto_Total - Custo da solução encontrada *
*               na fase de construção *
* aFat_1 - Solução inicial resultante do passo 1 do algoritmo *
* aFat - Matriz de alocação encontrada na fase de construção *
*
* Comentários : *
* A busca local faz alteração da alocação do fragmento com a *
* menor diferença benefício/custo *
*
* * * * *

```

```

procedure TGRADA.Busca_Local(aCusto_Total: real; aFat_1:
  matriz_int; var aFat: matriz_int);
var j, { Conjunto de fragmentos pertencentes a F. }
  k: word; { Conjunto de nos pertencentes a S. }
  vDiferenca: ^diferencas; { Custo/benefício da retirada do no. }
  vDiferencas: TList; { Lista de vDiferencas. }
  vCusto_Total: real; { Menor custo da matriz de alocação. }
begin
  Atualiza_FAT(aFat_1);
  { Executa passo 2 para vFat. }
  vDiferencas := TList.Create;
  try
    { Para cada fragmento. }
    for j := 0 to oFr - 1 do begin
      Self.Monta_LFCR(j, vDiferencas);
      try
        { Elimina a primeira cópia de LFCR. }
        while (NumFragAloc(j) > 1) and (vDiferencas.Count > 0) do begin
          vDiferenca := vDiferencas.Items[0];

```

```

oFAT[j, vDiferenca^.indice] := 0;
Self.Limpa_TList(vDiferencas);
vDiferencas.Clear;
{ Remonta LFCR. }
Self.Monta_LFCR(j, vDiferencas);
end;
finally
Self.Limpa_TList(vDiferencas);
vDiferencas.Clear;
end;
end;
finally
Self.Limpa_TList(vDiferencas);
vDiferencas.Clear;
vDiferencas.Free;
end;
{ Verifica se uma melhor solução foi encontrada. }
vCusto.Total := Self.Custo.Total(Self.oFat);
if vCusto.Total < aCusto.Total then begin
for j := 0 to Self.oFr - 1 do begin
for k := 0 to Self.oNo - 1 do begin
aFAT[j, k] := Self.oFAT[j, k];
end;
end;
end;
Self.oCusto.Total := vCusto.Total;
end;
end;
end;

```

```

(* * * * *
* Procedure: Atualiza_FAT *
* * * * *
* Objetivo : Atualiza FAT para a Busca Local. *
* * * * *
* Parâmetros : aFat.1 - Matriz inicial resultante do passo 1 *
* aFat - Matriz de alocação a ser atualizada. *
* * * * *
* Comentários : *
* Busca a réplica com menor dif. bem/custo para atualizar aFat *
* * * * *
*)
procedure TGRADA.Atualiza_FAT(aFat.1: matriz_int);
var j, { Conjunto de fragmentos pertencentes a F. }
k, { Conjunto de nos pertencentes a S. }
vFr: word; { Frag que possui a réplica com menor diferença. }
vIndice: smallint; { Índice do fragmento com menor diferença. }
vValor: real; { Valor da diferença da replica escolhida. }
vDiferenca: ^diferencas; { Custo/benefício da retirada do no. }
vLCO: TList; { Lista de diferenças entre benefício e custo de
retirada do nó. }

```

```

begin
  vIndice := -1;
  vValor := 0;
  vFr := 0;
  { Busca a réplica com maior diferença benefício/Custo. }
  for j := 0 to Self.oFr - 1 do begin
    { Monta LCO. }
    vLCO := TList.Create;
    Self.MontaLista(j, vLCO);
    try
      if vLCO.Count > 0 then begin
        if vIndice > 0 then begin
          vDiferenca := vLCO.Items[0];
          if vDiferenca^.valor > vValor then begin
            vFr := j;
            vIndice := vDiferenca^.indice;
            vValor := vDiferenca^.valor;
          end;
        end
        else begin
          { Primeira lista. }
          vFr := j;
          vDiferenca := vLCO.Items[0];
          vIndice := vDiferenca^.indice;
          vValor := vDiferenca^.valor;
        end;
      end;
    finally
      { Limpa a LCO. }
      Self.Limpa_TList(vLCO);
      vLCO.Clear;
      vLCO.Free;
    end;
  end;
  { Atualiza FAT de acordo com fragmento encontrado. }
  if vIndice > 0 then
    for k := 0 to vIndice - 1 do begin
      Self.oFAT[vFr, k] := aFAT.l[vFr, k];
    end;
  Self.oFAT[vFr, vIndice] := 0;
  for k := vIndice + 1 to Self.oNo - 1 do begin
    Self.oFAT[vFr, k] := aFAT.l[vFr, k];
  end;
end;
end.

```

Apêndice F

Implementação da Classe TSol_Otima para o algoritmo de busca exaustiva

```
(* * * * *
* Unit : uSol.Otima
*
* Objetivo : Classe que implementa a busca exaustiva
*
* Comentários:
* * * * * *)
type
TSol_Otima = class(TAlocacao.Dados)
private
  { Private declarations }
  oCusto_Total: real; { Custo total da oFAT. }
  { Determina oFAT com custo minimo. }
  function Define.FAT: matriz.int;
  { Calcula a solucao otima. }
  procedure Sol.Otima(aIndice: integer; aFAT: matriz.int);
  { Verifica se uma matriz de alocao eh viavel. }
  function eh_viavel(aFAT: matriz.int): boolean;
  { Faz a alteracao de FAT. }
  procedure Altera.FAT(aFAT: matriz.int; aCusto: real);

public
  { Public declarations }
  { Propriedades. }
  property RM: matriz.int read oRM;
  property UM: matriz.int read oUM;
  property FREQ: matriz.int read oFREQ;
  property FAT: matriz.int read oFAT;
  property SEL: matriz.real read oSEL;
```

```

property CTR: matriz_real read oCTR;
property Custo.Alocacao: real read oCusto.Total;

{ Construtor da Classe. }
constructor create(aTr, aNo, aFr, aFunc_Custo: integer);
    override;
{ Destructor da Classe. }
destructor destroy; override;
{ Preenche matrizes. }
procedure SetInPut(aRM, aUM, aFREQ: matriz_int; aCTR, aSEL:
    matriz_real; aTAM: vetor_real; aTam_Dados,
    aCusto.Trans, aVCini: real); override;
end;

implementation
{ TSol.Otima }
(* * * * * *)
* Procedure : Create *
* *
* Objetivo : Inicializa a classe *
* *
* Parâmetros: *
* aTr - Numero de Transacoes *
* aNo - Numero de Nos *
* aFr - Numero de fragmentos *
* *
* Comentários: *
* *
(* * * * * *)
constructor TSol.Otima.create(aTr, aNo, aFr, aFunc_Custo:
    integer);
begin
    inherited;
end;

(* * * * * *)
* Procedure : Destroy *
* *
* Objetivo : Destructor padrão da classe. *
* *
* Parâmetros: *
* *
* Comentários: *
* *
(* * * * * *)
destructor TSol.Otima.destroy;
begin
    inherited destroy;
end;

```



```

{ Primeira solucao. }
aFAT[Self.oFr - 1, Self.oNo - 1] := 0;
{ Se primeira solucao for viavel. }
if Self.eh.viavel(aFAT) then begin
  Self.Alterar_FAT(aFAT, vCusto);
end
else begin
  if (vCusto.Temp > 0) then begin
    { Segunda solucao. }
    aFAT[Self.oFr - 1, Self.oNo - 1] := 1;
    { Se segunda solucao for viavel. }
    if Self.eh.viavel(aFAT) then begin
      Self.Alterar_FAT(aFAT, vCusto.Temp);
    end;
  end;
end;
end;
end
else begin
  if (vCusto.Temp > 0) then begin
    { Segunda solucao. }
    aFAT[Self.oFr - 1, Self.oNo - 1] := 1;
    { Se segunda solucao for viavel. }
    if Self.eh.viavel(aFAT) then begin
      Self.Alterar_FAT(aFAT, vCusto.Temp);
    end;
  end;
end;
end;
Application.ProcessMessages;
end
else begin
  { Pega a linha correspondente. [0..oFr-1]}
  vLinha := aIndice div Self.oNo;
  { Pega a coluna correspondente. [0..oNo-1]}
  vColuna := (aIndice mod Self.oNo);
  aFAT[vLinha, vColuna] := 0;
  Application.ProcessMessages;
  Self.Sol_Otima(aIndice + 1, aFAT);
  Application.ProcessMessages;
  aFAT[vLinha, vColuna] := 1;
  Self.Sol_Otima(aIndice + 1, aFAT);
  Application.ProcessMessages;
end;
end;
end;

```

```

(* * * * *
* Função : eh.viavel *
* * * * *
* Objetivo : Verifica se uma matriz de alocao eh viavel. *
* * * * *

```



```

begin
  if Self.oCusto.Total > 0 then begin
    if (aCusto < Self.oCusto.Total) and (aCusto > 0) then begin
      for i := 0 to Self.oFr - 1 do begin
        for j := 0 to Self.oNo - 1 do begin
          Self.oFAT[i, j] := aFAT[i, j];
        end;
      end;
      Self.oCusto.Total := aCusto;
    end;
  end
else begin
  if Self.eh.viavel(aFAT) and (aCusto > 0) then begin
    for i := 0 to Self.oFr - 1 do begin
      for j := 0 to Self.oNo - 1 do begin
        Self.oFAT[i, j] := aFAT[i, j];
      end;
    end;
    Self.oCusto.Total := aCusto;
  end;
end;
end;
end.

```