APRENDIZADO DE MÁQUINA APLICADO À FRAGMENTAÇÃO VERTICAL DE BASES DE OBJETOS

Flavia Cardoso de Almeida Cruz

DISSERTAÇÃO 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. Gerson Zaverucha, Ph.D.

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

Drª. Fernanda Araújo Baião Amorim, D.Sc.

Prof. Geraldo Zimbrão da Silva, D.Sc.

Prof. Marcelo Finger, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

SETEMBRO DE 2002

CRUZ, FLAVIA CARDOSO DE ALMEIDA.

Aprendizado de Máquina aplicado à Fragmentação Vertical de Bases de Objetos [Rio de Janeiro] 2002

X, 163 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2002)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Fragmentação Vertical
2. Aprendizado de Máquina
3. Banco de Dados

I. COPPE/UFRJ II. Título (série)

Aos meus pais Jorge e Gloria,

meus irmãos Alexandre e Renata,

minha sobrinha Maria Eduarda

e minha avó Maria Aparecida

AGRADECIMENTOS

Ao Professor Gerson Zaverucha, pela orientação exemplar, incentivo constantes e ajuda durante o desenvolvimento deste trabalho.

À Professora Marta Mattoso, pela orientação e seriedade durante o desenvolvimento deste trabalho.

À Fernanda Baião pela orientação exemplar, ajuda e incentivo constantes, sem o qual não seria possível o término deste trabalho.

Aos membros da banca, professores Marcelo Finger e Geraldo Zimbrão.

À professora Inês pela ajuda e dicas sobre Prolog.

Ao Professor José Roberto Blaschek, que acompanha já algum tempo as etapas de minha vida, por sua preocupação, apoio e incentivo, além das oportunidades de crescimento profissional.

Aos demais professores e funcionários da COPPE Sistemas, principalmente às meninas da secretaria (Claudia, Solange, Sueli, Mercedes e Lúcia) por estarem sempre dispostas a ajudar e à Patty e à Marilene, pelo incentivo, apoio e soluções de problemas, principalmente durante o tempo em que trabalhei nos projetos COPPETEC.

Ao grande amigo Beto Boullosa pelo incentivo, ajuda e estudo principalmente no início do mestrado e que mesmo na Espanha continua me incentivando e oferecendo ajuda por e-mail.

A todos os amigos que me incentivaram e ajudaram: Daniel Borges, Pepe, Newton, José Afonso, Rodolfo Borges, Lenadro Vieira, Carla Delagado, Kate e Marcelo Teixeira. A Beto Quintanilha pelos conselhos (via ICQ). Em especial a André Ormastroni, Lucio Fialho e Melise pelos incentivos, conselhos e momentos de alegria e descontração.

Em especial ao amigo Edson Raposo que ajudou muito para a elaboração e conclusão deste trabalho, me acompanhou durante toda a jornada sempre com palavras de apoio e incentivo e por sua preocupação constante.

Às amigas do ballet: Maysa Machado, Carol Costa, Viviane Pires, Elana Criz, Glória e Márcia, e também Tony Couto, pelas alegrias e preocupações compartilhadas. À Maysa Machado (novamente) e Isabela Reisner por darem minhas aulas de ballet no período final da conclusão desta dissertação.

Em especial ao amigo Felipe Acker pela ajuda e discussões sobre o trabalho e pelo apoio, preocupação e incentivos constantes, principalmente neste último ano.

Aos meus pais Jorge e Gloria, meus irmãos, Alexandre e Renata e minha avó Maria Aparecida pelo carinho, atenção e ajuda constantes.

À minha sobrinha Maria Eduarda nascida durante o mestrado.

A todos que me ajudaram e incentivaram, obrigada.

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

APRENDIZADO DE MÁQUINA APLICADO À FRAGMENTAÇÃO VERTICAL DE BASES DE OBJETOS

Flavia Cardoso de Almeida Cruz

Setembro/2002

Orientadores: Gerson Zaverucha
Marta Lima de Queirós Mattoso

Programa: Engenharia de Sistemas e Computação

O desempenho das aplicações em Sistemas de Gerência de Base de Dados Orientado a Objetos Distribuídos é afetado fortemente pelo Projeto de Distribuição, que visa reduzir ao máximo o acesso das aplicações às informações irrelevantes e a troca de dados entre os nós de um sistema distribuído. Em um ambiente Orientado a Objetos, o Projeto de Distribuição é uma tarefa complexa e um problema em aberto. Nesta dissertação é apresentada uma abordagem baseada em conhecimento para a fase de Fragmentação Vertical do Projeto de Distribuição de Bases de Dados Orientadas a Objetos. O objetivo é mostrar como melhorar automaticamente o algoritmo de Fragmentação Vertical para produzir esquemas de fragmentação mais eficientes, através do uso de um Sistema de Revisão de Teorias em que a teoria inicial é representada por um algoritmo de Fragmentação Vertical reconhecidamente eficiente existente na literatura. É mostrada uma implementação em Prolog desse algoritmo de Fragmentação Vertical, e descrito como este pode ser usado como base de conhecimento para o processo do descoberta/revisão de conhecimento através da programação em lógica indutiva (ILP). Foram realizados experimentos que apontam para mudanças para melhoria do algoritmo original resultantes do processo de ILP.

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

MACHINE LEARNING APPLIED TO
VERTICAL FRAGMENTATION OF OBJECT BASES

Flavia Cardoso de Almeida Cruz

September/2002

Advisors: Gerson Zaverucha
Marta Lima de Queirós Mattoso

Department: Computer Science and Systems Engineering

The performance of applications on Distributed Object Oriented Database Management Systems is strongly affected by Distribution Design, which reduces irrelevant data accessed by applications and data exchange among sites. In an object-oriented environment, the Distribution Design is a complex task, and an open research problem. This work presents a knowledge-based approach for the vertical fragmentation phase of the distribution design of object-oriented databases. The objective of this work is to show the viability of automatically improving the vertical fragmentation algorithm to produce more efficient fragmentation schemas, using a theory revision system. A vertical fragmentation algorithm from the literature was chosen, based on its recognized efficiency, to represent the initial theory to be revised. This approach shows a Prolog implementation of this vertical fragmentation algorithm, and describes how it can be used as background knowledge for a knowledge discovery/revision process through Inductive Logic Programming (ILP). The experiments have pointed out changes to improve the original algorithm based on the ILP process.

ÍNDICE

Capítulo 1

INTRODUÇÃO

Este capítulo apresenta a motivação desta tese, os principais objetivos pretendidos e a organização do texto nos próximos capítulos.

1.1 MOTIVAÇÃO

Distribuição de dados e paralelismo em Sistemas Gerenciadores de Bases de Dados (SGBD) são duas das técnicas mais eficientes para o aumento de desempenho em aplicações que manipulam um grande volume de dados. No entanto, para maximizar o aumento de desempenho que pode ser alcançado, é essencial que a distribuição de dados seja realizada de forma adequada, visando reduzir ao máximo o acesso das aplicações às informações irrelevantes e a troca de dados entre os nós de um sistema distribuído, que são os objetivos do projeto de distribuição (ÖZSU e VALDURIEZ, 1999). Segundo BAIÃO (2001), o projeto de distribuição de bases de dados orientadas a objetos (PDBDOO) é um problema complexo, pelas seguintes razões: (i) a não existência de uma formalização consensual do problema na literatura, em especial do modelo de dados orientado a objetos; (ii) a existência de muitos parâmetros de entrada ao problema que devem ser levados em consideração; (iii) a existência de muitas diferenças entre os modelos relacional e orientado a objetos, que limitam uma adaptação direta dos algoritmos existentes; (iv) a existência de passos intermediários no problema com objetivos conflitantes; e (v) a necessidade de se obter estimativas e heurísticas para direcionar a solução do problema, que muitas vezes também são conflitantes quando combinadas.

A abordagem descendente para o projeto de distribuição de bases de dados divide-se em duas fases, denominadas fragmentação e alocação. A fragmentação de dados é responsável por agrupar, em fragmentos, informações que são acessadas simultaneamente pelas aplicações. Já a etapa de alocação é responsável pelo armazenamento físico dos fragmentos de classe que foram gerados pela etapa de fragmentação entre os nós do sistema distribuído, e pela replicação de dados. Devemos lembrar que nos sistemas orientados a objetos (OO) a replicação dos dados se encontra ainda como um problema em aberto (BAIÃO, 2001).

Para a fragmentação de uma classe podem ser utilizadas duas técnicas básicas: fragmentação horizontal e fragmentação vertical. No modelo orientado a objetos, a fragmentação horizontal distribui as instâncias da classe entre os fragmentos, os quais vão conter um subconjunto das instâncias da classe, compartilhando a mesma estrutura lógica (atributos e métodos). Por outro lado, a fragmentação vertical fragmenta a estrutura lógica

da classe e a distribui entre os fragmentos, que vão portanto conter as mesmas instâncias, mas com estruturas lógicas diferentes. A fragmentação horizontal de uma classe pode ainda ser subdividida em primária e derivada. A fragmentação horizontal primária (PHF) visa otimizar operações sobre um conjunto de dados (como a busca na extensão de uma classe) através da redução dos dados que precisam ser acessados e do aumento do grau de paralelismo que pode ser aplicado durante a execução da operação. A fragmentação horizontal derivada (DHF), no entanto, visa otimizar operações de navegação entre os dados através do agrupamento no disco de objetos de classes diferentes que estejam relacionados. Também é possível aplicar as técnicas de fragmentação vertical e horizontal simultaneamente na mesma classe (fragmentação híbrida) ou em classes diferentes (fragmentação mista) do esquema.

Existem diversos trabalhos na literatura que endereçam o PDBDOO, incluindo BELLATRECHE *et al.* (2000), BELLATRECHE *et al.* (1998), BELLATRECHE *et al.* (1996), CHEN e SU (1996), EZEIFE e BARKER (1995), EZEIFE e BARKER (1998), KARLAPALEM *et al.* (1994), MAIER *et al.* (1994), SAVONNET *et al.* (1998). Entretanto, devido à complexidade do problema, a maioria destes trabalhos baseia-se em um subconjunto limitado de estimativas e heurísticas. Além disso, algumas abordagens trabalham sobre a base de dados instanciada, o que limita a sua aplicabilidade. É importante ressaltar também que os trabalhos relacionados limitam a sua análise a apenas uma das técnicas de fragmentação existentes (vertical ou horizontal) em todas as classes do esquema, resultando em esquemas de fragmentação puramente horizontal ou puramente vertical.

Em (BAIÃO, 1997, BAIÃO e MATTOSO, 1997) são mostrados os benefícios alcançados com as técnicas de fragmentação mista e híbrida no aumento de desempenho das aplicações avaliadas.

BAIÃO (2001) propõe uma metodologia completa com algoritmos para o projeto de distribuição de bases de dados orientadas a objetos. A principal contribuição desta metodologia é a etapa de análise, que escolhe automaticamente a técnica de fragmentação mais adequada a cada classe do esquema (horizontal, vertical ou híbrida) baseando-se em heurísticas obtidas através de resultados experimentais. No mesmo trabalho é proposta uma abordagem baseada em conhecimento para refinamento de algoritmos. A abordagem utiliza

técnicas de inteligência artificial (revisão de teorias) com o objetivo de automaticamente aumentar a eficiência do algoritmo. Mais especificamente, é proposto o refinamento do algoritmo de análise através de revisão de teoria. Entretanto, uma vez decidido se a classe será fragmentada de modo horizontal, vertical ou híbrido, é necessário usar um algoritmo eficiente para o tipo de fragmentação escolhido. Como os algoritmos existentes são tipicamente heurísticos é difícil garantir sua abrangência e eficiência. Uma possibilidade de refinar as heurísticas é através do aprendizado a partir de exemplos bem sucedidos.

1.2 OBJETIVOS

É extremamente interessante estender o uso da abordagem de aprendizagem indutiva de BAIÃO (2001) a outras fases do Projeto de Distribuição. Nesta dissertação, nós apresentamos uma abordagem baseada em conhecimento para a fase de Fragmentação Vertical do Projeto de Distribuição de Bases de Dados Orientada a Objetos, fornecendo uma maneira alternativa de automaticamente modificar um algoritmo eficiente já existente. Um resumo desta abordagem pode ser encontrado em (CRUZ *et al.*, 2002).

O objetivo do trabalho é estender a estrutura proposta por (BAIÃO 2001) para lidar com o problema da fragmentação vertical de uma classe, mostrando a viabilidade de melhorar automaticamente o algoritmo de Fragmentação Vertical para produzir esquemas de fragmentação ainda mais eficientes, através do uso de um Sistema de Revisão de Teoria.

A estrutura geral do projeto de Fragmentação Vertical está ilustrada nas figuras 1.1 e 1.2. O algoritmo utilizado nesta dissertação para a fragmentação vertical é uma extensão de (NAVATHE e RA, 1989, NAVATHE et al., 1984), que por sua vez foi estendido por (BAIÃO 2001) para envolver aspectos de OO tais como métodos. Chamamos esse algoritmo de VFNBC. O Módulo Heurístico é constituído desse algoritmo e será detalhado no capítulo 3.

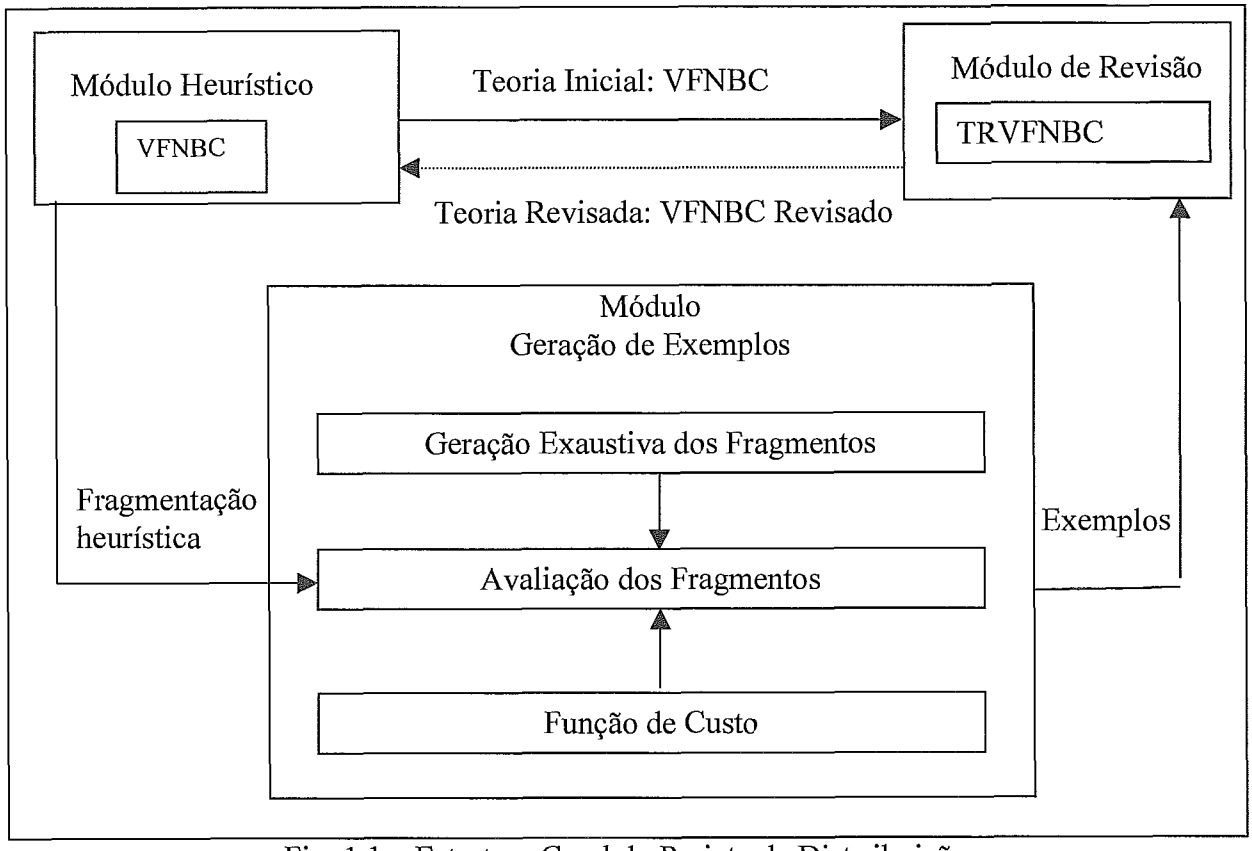


Fig. 1.1 – Estrutura Geral do Projeto de Distrribuição

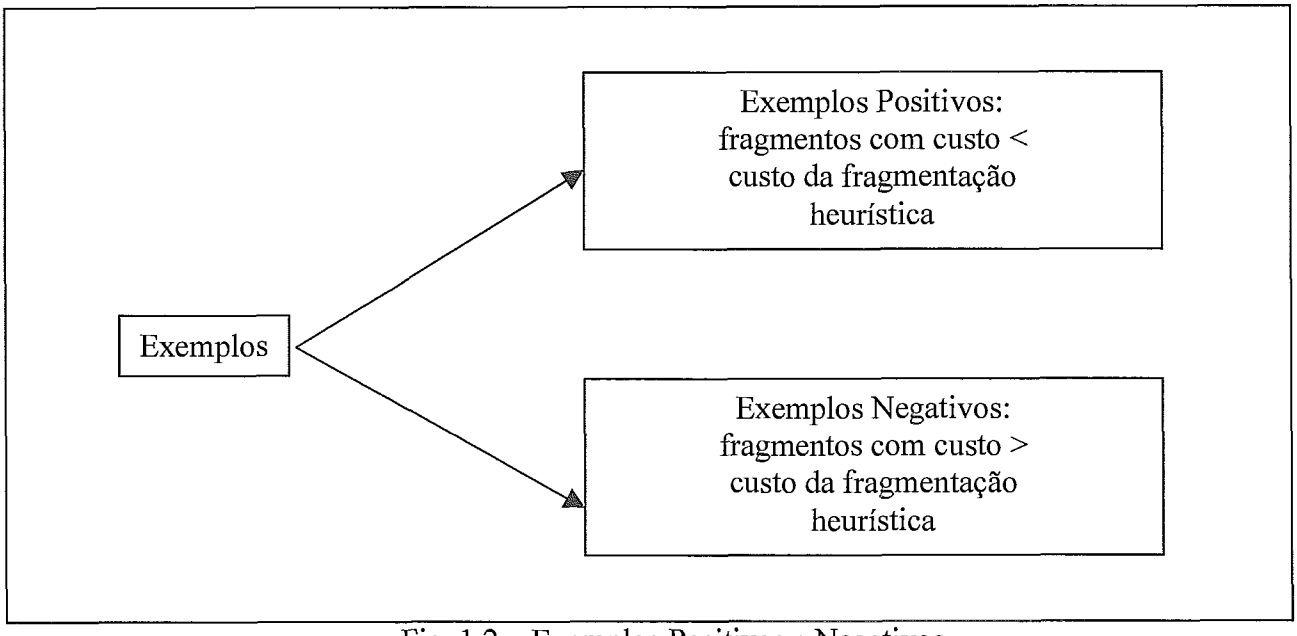


Fig. 1.2 – Exemplos Positivos e Negativos

A fragmentação heurística (gerada pelo algoritmo VFNBC) será enviada para o Módulo de Busca Exausiva. O Módulo Busca Exaustiva apresentado na Fig. 1.1 é formado por três componentes: Geração dos Fragmentos, Avaliação dos Fragmentos e Função de Custo. Através do Módulo Função de Custo, será calculado o custo da fragmentação heurística, que chamaremos de custo heurístico. O Módulo Geração dos Fragmentos gera, utilizando busca exaustiva, todas as possíveis fragmentações de uma classe a ser verticalmente fragmentada. Todas essas fragmentações serão avaliadas pelo Módulo Avaliação dos Fragmentos, que utilizando a função de custo de RUBERG (2001) e o custo heurístico, serão divididas em dois grupos: boas fragmentações verticais para uma determinada classe (com custo menor que o custo heurístico) e fragmentações verticais ruins, ou seja, com custo total de processamento maior que o custo heurístico. Esses dois grupos representarão respectivamente os exemplos positivos e negativos necessários para a revisão do FVNBC pelo Módulo de Revisão (Fig. 1.2).

O Módulo de Revisão (capítulo 4) é responsável pela revisão do VFNBC, sendo este usado como base de conhecimento para o processo de descoberta/revisão de conhecimento através da programação em lógica indutiva - ILP (MUGGLETON e DE RAEDT, 1994, LAVRAC e DZREROSKI, 1996). Esta revisão será feita utilizando uma abordagem baseada em conhecimento que implementa uma técnica de inteligência artificial denominada revisão de teorias (WROBEL, 1996). O processo de revisão de teorias é responsável por alterar automaticamente o algoritmo original VFNBC (denominado teoria inicial, ou conhecimento preliminar) de forma a adequá-lo para a geração do melhor esquema de fragmentação vertical encontrado pela busca (exemplos). O resultado do procedimento de refinamento é o algoritmo alterado (denominado teoria revisada) capaz de obter a solução ótima que lhe foi apresentada como entrada. A Revisão de Teoria do VFNBC chamaremos de TRVFNBC e a Teoria Inicial será o VFNBC e a teoria revisada será o VFNBC revisado. Ao final, a teoria revisada é passada para o Módulo Heurístico que, assim, conterá um algoritmo de fragmentação vertical revisado que irá propor, de forma extremamente rápida, os fragmentos verticais ótimos da classe (ou próximos do ótimo). A abordagem heurística é utilizada pois, como apontado por (ÖZSU e VALDURIEZ, 1999), a fragmentação de classes é um problema NP-difícil, o que

impossibilita a aplicação de métodos de busca exaustiva, justificando desta forma a abordagem heurística (mais eficiente).

Para esta abordagem da base de conhecimento, foi necessário representar o algoritmo de fragmentação vertical (VFNBC) como um conjunto de regras (cláusulas em Prolog). A revisão é feita por meio de um refinamento do algoritmo, a fim de assim descobrir um novo conjunto de regras que representarão o algoritmo revisado (seção 4.4). O objetivo não é propor o melhor algoritmo de Fragmentação Vertical e sim verificar o processo de revisão de um algoritmo de Fragmentação Vertical reconhecidamente eficiente através de técnicas de descoberta de conhecimento.

O uso do VFNBC resultou em proposta e implementação de melhorias com base em experimentos e análises realizadas. Além disso, foram refinados alguns conceitos presentes na proposta original do algoritmo de NAVATHE E RA (1989) .

O algoritmo revisado pode refletir importantes resultados implícitos sobre o problema da fragmentação de classes, isto é, ainda não descobertos pelos algoritmos de Projeto de Distribuição propostos na literatura.

1.3 ORGANIZAÇÃO DO TEXTO

A organização desta dissertação é como segue: o capítulo seguinte apresenta algumas definições da literatura a respeito da tarefa do Projeto de Distribuição, identifica algumas dificuldades que motivam o uso de uma abordagem baseada em conhecimento para este problema e mostra o estado da arte na área de pesquisa de Projeto de Distribuição (em particular na área de Fragmentação Vertical) e no campo de ILP.

No capítulo 3 é descrito o algoritmo de fragmentação vertical, apresentando as contribuições desta dissertação para sua melhoria e conceitos, notações e terminologias utilizadas para a descrição do mesmo.

O capítulo 4 discute o uso de ILP para revisar o Algoritmo de Fragmentação Vertical, descreve sua implementação em Prolog e modelagem dos dados necessária para ser feita a revisão do algoritmo.

Finalmente, o capítulo 5 conclui esta dissertação.

Capítulo 2

PROJETO DE DISTRIBUIÇÃO E APRENDIZADO DE MÁQUINA

Este capítulo apresenta uma caracterização do problema tratado nesta dissertação, e descreve o estado da arte na área.

2.1 APRESENTAÇÃO DO PROBLEMA

O Projeto de Distribuição (PD) envolve tomar decisões quanto a fragmentação e colocação dos dados entre os nós de um sistema distribuído (ÖZSU e VALDURIEZ, 1999). A abordagem descendente para o projeto de distribuição de bases de dados divide-se em duas fases, denominadas fragmentação e alocação. A fase de fragmentação é o processo de agrupar, em fragmentos, informações que são acessadas simultaneamente pelas aplicações. A etapa de alocação, por sua vez, é responsável pelo armazenamento físico dos fragmentos de classe, que foram gerados pela fase de fragmentação, entre os nós do sistema distribuído e pela replicação de dados. Para fragmentar uma classe é possível usar duas técnicas básicas (ÖZSU e VALDURIEZ, 1999): fragmentação horizontal (primária ou derivada) e fragmentação vertical (que é o escopo deste trabalho).

Em um ambiente orientado a objetos (OO), a fragmentação vertical quebra a estrutura lógica da classe (seus atributos e métodos) distribuindo-a através dos fragmentos, que, desta maneira, conterão logicamente os mesmos objetos mas com estruturas diferentes. A fragmentação vertical favorece o acesso à extensão da classe e o uso de atributos e métodos da classe, removendo dados irrelevantes acessados por operações. A fragmentação horizontal distribui as instâncias de uma classe em diversos fragmentos, os quais vão conter um subconjunto das instâncias da classe, compartilhando a mesma estrutura lógica mas conteúdo diferente. Existe também a fragmentação híbrida, que combina as fragmentações vertical e horizontal na mesma classe (NAVATHE *et al.*, 1984, ÖZSU e VALDURIEZ, 1999, SU *et al.*, 1998, BAIÃO, 1997, BAIÃO e MATTOSO, 1997, 1998, BAIÃO *et al.*, 1998, 2000, 2001 e BAIÃO, 2001).

Sabemos que o PDBDOO é um problema NP-difícil (ÖZSU e VALDURIEZ, 1999). Quanto ao modelo de objetos, questões adicionais contribuem para o aumento da dificuldade da tarefa e tornam-no em um problema ainda mais complexo.

2.2 MODELO DE OBJETO

Existem variações, entre diferentes pesquisadores, quando se trata de descrever o modelo de objetos. Esta falta de padronização formal do modelo, que seja consensual,

contribui para a dificuldade de pesquisa na área de Sistemas de Gerenciamento de Banco de Dados Orientados a Objetos (SGBDOO). Há propostas de padronizações de especificações de modelos de objetos, assim como o padrão ODMG 3.0 (escrito por membros do "Object Data Management Group"), que também pode ser utilizado para conversão e armazenamento de Objetos para Relacional ou para outras representações de banco de dados (CATTEL et al., 2000), e o SQL:1999 (EISENBERG e MELTON, 1999). Nesta seção apresentamos alguns conceitos que utilizaremos nesta dissertação.

2.2.1 CLASSES

A classe é um grupo de todas as instâncias ou objetos de um determinado tipo. Utilizaremos a mesma representação utilizada por (BAIÃO 2001) que utiliza a seguinte notação para determinar uma classe:

$$C = (K, A, M, I) \quad (1)$$

onde K é o identificador da classe, A é o conjunto de atributos, M representa o conjunto de métodos e I é o conjunto das instâncias ou objetos da classe C, que representa sua extensão.

2.2.2 OBJETOS

Utilizaremos a mesma representação utilizada por (BAIÃO 2001), onde é especificado que um objeto é composto por seu identificador (OID) e um conjunto de variáveis instanciadas definindo seu estado. O conjunto de variáveis instanciadas de um objeto determina o valor corrente dos atributos definidos em sua classe.

$$o = (OID,estado) \quad (2)$$

O identificador do objeto (OID) distingue fisicamente e logicamente os objetos, sem levar em consideração o seu estado (KHOSHAFIAN e COPELAND, 1986).

Considere a classe Pessoa representada de acordo com (1) como:

Pessoa = (id1, {nome: string, anoNascimento: integer, filhos: set(Pessoa)}, {idade: integer}, {p1, p2, p3, p4, p5, p6})

Objetos da classe Pessoa podem ser representados de acordo com (2) como:

(p1, {nome: "Jorge", anoNascimento: 1950, filhos: {p3, p4, p5} })

(p2, {nome: "Gloria", anoNascimento: 1948, filhos: { p3, p4, p5} })

(p3, {nome: "Alexandre", anoNascimento: 1973, filhos: { } })

(p4, {nome: "Flavia", anoNascimento: 1975, filhos: { } })

(p5, {nome: "Renata", anoNascimento: 1978, filhos: {p6} })

(p6, {nome: "Maria Eduarda", anoNascimento: 2001, filhos: {} })

2.2.3 ELEMENTOS

Para tratarmos os atributos e métodos de uma classe de maneira uniforme, quando possível, utilizaremos a definição encontrada em (BAIÃO 2001), que chama estes de elementos(E). Portanto, sendo A e M conforme (1) desta seção, temos:

$$E = A \cup M \qquad (3)$$

e os n elementos de E serão representados por

$$E = \{E1, E2, ..., En\} \qquad (4)$$

2.3 TRABALHOS RELACIONADOS

Como o PDBDOO é um problema NP-difícil (ÖZSU e VALDURIEZ, 1999) e a falta de padronização de um sistema formal do modelo de objetos contribui para a dificuldade de pesquisa na área de SGBDOO (visto na seção anterior), algoritmos baseados em heurísticas têm sido propostos na literatura para tratar o problema de uma maneira eficiente (BAIÃO, 2001, NAVATHE e RA, 1989, NAVATHE et al., 1984, EZEIFE e BARKER, 1998, BELLATRECHE et al., 1996, CHEN e SU, 1996, MALINOWSKI, 1996). Adicionalmente, alguns pesquisadores propõem a aplicação de técnicas de aprendizado de máquina (MITCHELL, 1997) no contexto de banco de dados. Por exemplo, BLOCKEEL e DE RAEDT (1996) e BLOCKEEL e DE RAEDT (1998) apresentam uma abordagem para o projeto indutivo de bancos de dados dedutivos, baseando-se nas instâncias existentes no banco de dados para a definição de predicados. GETOOR et al. (2001a) e GETOOR et al. (2001b) utiliza modelos probabilísticos originados da área de redes Bayesianas a fim de estimar a seletividade de consultas em um processador de consultas e prever a estrutura de banco de dados relacionais.

Existem na literatura muitos trabalhos que tratam do projeto de distribuição no modelo de dados relacional, incluindo (MESQUITA E FINGER, 1998, CERI e NAVATHE, 1983, MOLINA e HSU, 1995, NAVATHE e RA, 1989, NAVATHE et al.,

1995, ÖZSU e VALDURIEZ, 1999). No modelo de dados orientado a objetos, muitos trabalhos também evidenciam a importância do projeto de distribuição para o aumento de desempenho de aplicações que manipulam um grande volume de dados (KARLAPALEM et al., 1994, KARLAPALEM e LI, 2000, MAIER et al., 1994). Um breve resumo e análise detalhada destes trabalhos pode ser encontrado em (BAIÃO, 2001).

Outros trabalhos (BAIÃO, 2001, BAIÃO et al., 1998a, BAIÃO et al., 2001, BAIÃO et al., 1998b, BAIÃO et al., 2000) apresentaram uma estrutura para tratar o problema da fragmentação de uma classe, incluindo um módulo de revisão de teoria que melhorasse automaticamente a escolha entre as técnicas horizontais e/ou verticais de fragmentação (será apresentado na seção 2.4). Neste trabalho, nós estendemos estas idéias e propomos uma abordagem de revisão de teoria para melhorar automaticamente um Algoritmo de Fragmentação Vertical.

Os trabalhos mais relevantes que abordam a fragmentação vertical de classes são os de BELLATRECHE et al. 1996) e EZEIFE e BARKER (1998).

BELLATRECHE et al. (1996) aplica o algoritmo proposto por NAVATHE e RA (1989) para fragmentação vertical em cada classe do esquema do banco de dados, baseado somente nos métodos uma vez que é assumido encapsulamento. As freqüências de acesso das consultas são consideradas de modo a agrupar em um mesmo fragmento métodos freqüentemente acessados simultaneamente. Este trabalho basicamente aponta um conjunto de operações em uma extensão da classe. Contudo, relacionamentos entre classes não são considerados no processo de fragmentação, e consequentemente operações de navegação não são beneficiadas pela fragmentação resultante.

O trabalho de EZEIFE e BARKER (1998) segue o mesmo esquema de classificação para classes do esquema de banco do dados proposto em seu trabalho anterior EZEIFE e BARKER (1995), e também apresenta algoritmos específicos para a fragmentação vertical de todas as classes em cada grupo. De forma a agrupar métodos, os autores usam as mesmas técnicas que outros trabalhos na literatura utilizam para agrupar atributos, e também capturar relações de herança entre as classes e formar partições verticais com os atributos e métodos da classe considerando os acessos das aplicações a todas as suas subclasses. O maior problema no uso desse algoritmo está no fato de depender da existência dos dados da base antes da execução do projeto. Isso pode não estar disponível além de uma execução exaustiva, desprezando relacionamentos especificados no esquema.

As principais características das estratégias de fragmentação vertical mencionadas neste capítulo estão resumidas na tabela 2.1. Considerar os métodos e chamadas de métodos a outros métodos durante a fragmentação vertical, significa considerar os atributos acessados pelos métodos. Devido às vantagens da proposta de BAIÃO *et al.* (1998) optou-se por seguir com o algoritmo de BAIÃO baseado em NAVATHE e RA (1989). Além disso, experimentos de NAVATHE e RA (1989) mostram a eficiência de sua proposta.

Tabela 2.1: Trabalhos relacionados sobre fragmentação vertical de classes -Fonte: (BAIÃO, 2001)

Características	Bellatreche *et al.* (1996)	Ezeife & Barker (1998)	Baiao *et al.* (1998)
Analisa adequação da classe para fragmentação			√
Considera chamada de métodos a outros métodos		√	√
Considera consultas definidas pelo usuário	√	√	√
Considera tanto atributos quanto métodos da classe		√	√
Considera freqüência de execução de aplicações	√	√	√
Trabalha independentemente da implementação do SGBD, e não requer uma base de dados instanciada	√		√
Considera os relacionamentos definidos no esquema da base			√
Apresenta algoritmos		√	√

2.4 Visão Geral do Projeto de Distribuição e Metodologia

A maioria dos trabalhos da literatura sobre projeto de distribuição no modelo orientado a objetos tratam apenas de uma técnica de fragmentação (horizontal ou vertical). Desta forma, não são abordadas a escolha da melhor técnica de fragmentação a ser aplicada a uma classe, e não é analisada a adequação de cada classe ao processo de fragmentação. Isto leva à aplicação da mesma técnica em todas as classes.

No entanto, existem trabalhos (BAIÃO, 1997, BAIÃO e MATTOSO, 1997, NAVATHE *et al.*, 1984) que mostram o benefício da fragmentação mista (combinação de fragmentação vertical e horizontal em diferentes classes do esquema) e fragmentação híbrida (combinação de fragmentação vertical e horizontal na mesma classes) para aumentar o desempenho das aplicações. Segundo estes trabalhos, para propor melhores

esquemas de fragmentação também é importante analisar o esquema do banco de dados e as características das aplicações (BAIÃO, 2001).

Portanto, existe a necessidade de uma estratégia completa para o projeto de distribuição de bases de dados orientadas a objetos (PDBDOO), como evidenciado em (ÖZSU e VALDURIEZ, 1999, MOLINA e HSU, 1995). Esta estratégia se faz ainda mais necessária porque a decisão entre todos os tipos possíveis de fragmentação não é trivial em um ambiente genérico, levando-se em conta uma série de aplicações que executarão sobre os dados. Isto porque a combinação entre estas diversas operações e todos os parâmetros relevantes ao problema não levam o projetista da distribuição diretamente ao esquema de fragmentação com melhor desempenho. Especialmente em aplicações complexas do mundo real, a dificuldade na tomada das decisões de projeto é ainda maior. A maior dificuldade é decidir quais classes devem ser fragmentadas horizontalmente e/ou verticalmente. Ainda, na fragmentação horizontal as classes podem seguir para fragmentação horizontal primária ou derivada, e esta combinação com a fragmentação vertical pode, em alguns casos, ser conflitante e tornar inviável a implementação do esquema de fragmentação.

De modo a tratar este problema, BAIÃO (2001) descreve uma metodologia completa para o projeto de distribuição de bases de dados orientadas a objetos. Na Fig. 2.1 é apresentada uma visão geral da metodologia, onde é destacado o fluxo de dados (representados pelas elipses) entre as suas fases (representadas pelos retângulos). As seguintes fases são envolvidas:

i) Fase de Análise: analisa informações adquiridas pelo Módulo de Interface e define heurísticas para decidir a técnica de fragmentação mais adequada (nenhuma, horizontal – primária ou derivada – e/ou vertical) para cada classe;

ii) Fase de Fragmentação Vertical: para cada classe apontada pela Etapa de Análise a ser verticalmente fragmentada, é realizada a fragmentação vertical;

iii) Fase de Fragmentação Horizontal: recebe uma lista de duplas de classes da forma (CP, CD) onde deve-se aplicar a Fragmentação Horizontal Primária na classe CP e a Fragmentação Horizontal Derivada na classe CD. Além disso gera o conjunto de classes que terão fragmentação híbrida (aplicação das duas técnicas - horizontal e vertical - na mesma classe).

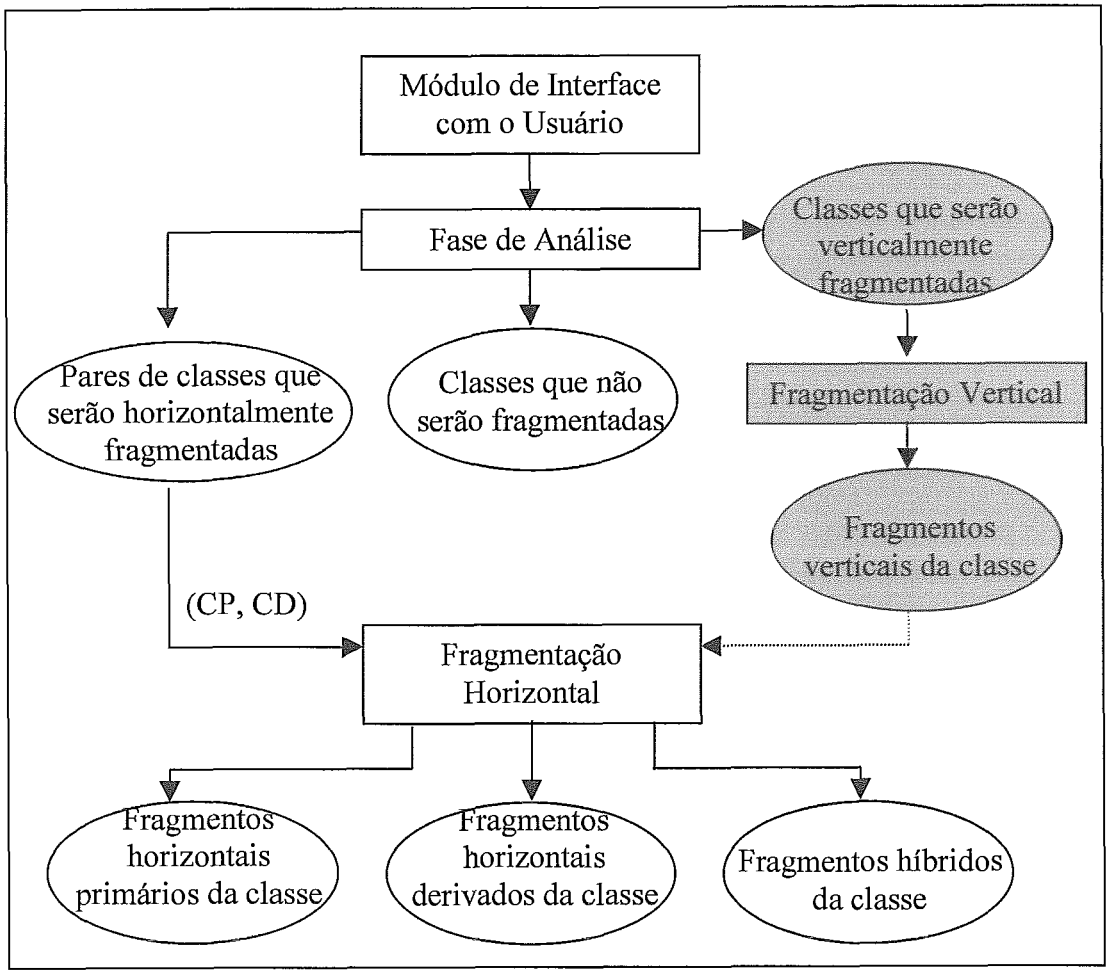


Fig. 2.1. Visão geral da metodologia para a fragmentação da classe para o PDBDOOs (BAIÃO, 2001)

Desta forma, seguindo esta metodologia, é gerado automaticamente um esquema de fragmentação adequado e eficiente a partir das informações fornecidas como entrada.

Na próximo capítulo são descritas as mudanças feitas no algoritmo utilizado na Etapa de Fragmentação Vertical por (BAIÃO, 2001), que é uma extensão de (NAVATHE E RA, 1989, NAVATHE et al., 1984), assim como maiores detalhes sobre o algoritmo. Detalhes sobre os demais algoritmos utilizados na metodologia acima descrita podem ser encontrados em (BAIÃO, 2001), bem como a metodologia completa.

Capítulo 3

O ALGORITMO DE FRAGMENTAÇÃO VERTICAL

Este capítulo descreve o algoritmo para a fase de Fragmentação Vertical, incluindo as propostas desta dissertação para sua melhoria.

3.1 INTRODUÇÃO

A Fragmentação Vertical contribui para aumentar o desempenho de sistemas sobre bases distribuídas na medida em que reduz o acesso das aplicações às informações irrelevantes, agrupando no mesmo fragmento os atributos e métodos da classe que são acessados simultaneamente pelas operações mais freqüentes. Desta forma, a estrutura lógica da classe é quebrada em fragmentos distintos.

Em (ÖZSU e VALDURIEZ,1999) é apontado que para a fragmentação vertical se uma relação tem m atributos que não são chaves primárias, então o número de possíveis fragmentos é igual a B(m), onde este é o m-ésimo número de Bell (*"mth Bell number"*). Para grandes valores de m, B(m) é aproximadamente m^m. Desta forma se torna evidente recorrer a heurísticas. Um dos tipos de heurísticas apontados em (ÖZSU e VALDURIEZ,1999) é o agrupamento (*"grouping"*). Porém NAVATHE e RA (1989) apresentam uma nova abordagem utilizando um grafo completo chamado grafo de afinidades onde o valor de uma aresta representa a afinidade entre dois atributos.

O algoritmo utilizado nesta dissertação, que será apresentado neste capítulo, para a fragmentação vertical. é uma extensão do algoritmo proposto por (NAVATHE e RA, 1989, NAVATHE *et al.*, 1984), que por sua vez foi estendido por (BAIÃO, 2001) para envolver aspectos de OO tais como métodos. Chamamos esse algoritmo de VFNBC (*Vertical Fragmentation Algorithm from Navathe, extended by Baião and Cruz*). No apêndice D é encontrado um breve resumo da proposta de (NAVATHE e RA, 1989).

Somente as operações de projeção e elementos envolvidos nos predicados de seleção influenciam na fase de Fragmentação Vertical. Desta forma, o algoritmo VFNBC recebe como entrada, estendendo as idéias de (BAIÃO, 2001), para cada classe a ser verticalmente fragmentada, as seguintes informações: transações, frequências, operações de projeção e seleção e elementos envolvidos (atributos da classe envolvidos em operações de projeção e seleção e acessos aos atributos feitos dentro do método).

Com essas informações são gerados como saída os elementos da classe que são acessados simultaneamente pelas operações mais freqüentes.

Com base nestas informações, a idéia geral da proposta de NAVATHE *et al.* (1989) é definir os fragmentos verticais através da construção de um grafo de afinidades entre os elementos, onde cada ciclo resultante no grafo representa um fragmento.

As principais vantagens do algoritmo VFNB, mais detalhado em (BAIÃO, 2001), são:

(i) Geração de todos os fragmentos verticais em uma única iteração com tempo na ordem $O(n^2)$ (onde n é o número de elementos da classe), tornando-o mais eficiente que outros algoritmos da literatura, como o de MCCORMICK *et al.* (1972), assim como o algoritmo proposto por NAVATHE *et al.* (1984) que gera fragmentos verticais com tempo $O(n^2 \log n)$;

(ii) Não requer procedimento de partição binária iterativa como os algoritmos de EZEIFE e BARKER (1998), NAVATHE *et al.* (1984) e CORNELL e YU (1987), o que reduz a sua complexidade computacional;

(iii) É adaptável para o modelo de dados orientado a objetos;

(iv) É adaptável para a geração de fragmentos horizontais da classe (utilizado na Fase de Fragmentação Horizontal da metodologia de BAIÃO (2001)).

Na próxima seção será apresentada uma visão geral do algoritmo VFNB (Vertical Fragmentation Algorithm by Navathe and extend by Baião) e as extensões feitas neste dissertação (por Cruz), que chamaremos de VFNBC, e no decorrer do capítulo serão mais detalhadas tais idéias.

3.2 VISÃO GERAL DO ALGORITMO

Nas próximas seções serão apresentados conceitos, notações, terminologias, discussões sobre a formação e extensão dos ciclos e sua extensão, e , finalmente, todo o algoritmo juntamente com o pseudo-código do mesmo. Com o intuito de tornar mais fácil a leitura e compreensão das seções a seguir, apresentaremos agora uma idéia geral do funcionamento do algoritmo.

Para cada classe a ser verticalmente fragmentada, juntamente com informações de entrada do algoritmo, apresentadas na seção 3.1, o Algoritmo VFNBC irá definir a divisão dos elementos da classe em fragmentos (ou partições). Os fragmentos serão compostos por elementos que são acessados simultaneamente por operações e dizemos que estes possuem, portanto, um alto grau de afinidade. Chamaremos o conjunto de todos os fragmentos, ou partições, de fragmentação de uma classe. Devemos ressaltar, que na verdade, a fragmentação vertical de uma classe, segundo (ÖZSU e

VALDURIEZ, 1999), de uma relação R produz fragmentos R_i, para $1 \leq i \leq r$, onde $\forall i$, Ri contém um subconjunto dos atributos de R assim como a chave primária de R, e r é o número de fragmentos.

Através dos dados de entrada, é construída a Matriz de Uso dos Elementos, que contém informações sobre o uso dos elementos pelas operações, como, por exemplo, a freqüência de acesso. Com os dados contidos nesta matriz é gerada a Matriz de Afinidades entre os Elementos. A afinidade entre dois atributos é indicada pela soma da freqüência de acesso dos elementos acessados simultaneamente por operações. Supondo que a classe tenha n elementos, esta matriz terá dimensão n x n e cada entrada (i, j) representará a afinidade entre os elementos i e j, para $i \neq j$.

Esta Matriz de Afinidades entre os Elementos representará a lista inicial de arestas candidatas que serão utilizadas a seguir para a construção do Grafo de Afinidades. Cada entrada (i, j) da matriz representa uma aresta, onde os dois nós que definem uma aresta são os elementos i e j e o valor de afinidade entre os elementos contido na entrada (i, j) é o peso da aresta. Esta lista representa as arestas que podem ser inseridas na contrução do Grafo de Afinidades. As arestas do grafo representarão nosso conjunto inicial de arestas, que chamaremos de arestas candidatas, e os nós serão os elementos.

A partir das arestas candidatas, é construída uma árvore geradora linearmente conectada (*linearly connected spanning tree*) que representa o Grafo de Afinidades. Podemos entender por árvore geradora linearmente conectada uma árvore geradora que contém apenas duas pontas. A construção é feita começando por um nó aleatório e a cada passo é selecionada a aresta de maior valor, dentre as arestas candidatas, que possa ser conectada linearmente à árvore (conectada a uma das pontas). Quando a aresta selecionada formar um ciclo, o algoritmo tenta estender este ciclo, desprezando na seleção as arestas que formam outros ciclos ou que não sejam candidatas a extensão do ciclo, ou seja, selecionando uma aresta que possa ser utilizada para estender o ciclo em questão. O algoritmo tenta sempre estender o ciclo encontrado, aumentando a área interna do ciclo, para diminuir o número de fragmentos encontrados, diminuindo assim a sobrecarga (*overhead*) de gerência do SGBD sobre um grande número de fragmentos. Verificada a impossibilidade de estender o ciclo, este é identificado como um fragmento (ou partição). Somente depois desta identificação que o algoritmo voltará a considerar todas as arestas candidatas que ainda não foram utilizadas na construção da árvore. O algoritmo, então, seleciona a próxima aresta a ser inserida na árvore, buscando novos ciclos, e termina quando não houver mais arestas candidatas.

Após o término do algoritmo é criado um fragmento contendo todos os nós que não foram incluídos em nenhum fragmento antes encontrado. Isto se deve ao fato desses nós possuírem baixa afinidade com os demais, e, portanto, não sendo muito acessados por operações, podem assim ficar agrupados em um único fragmento.

3.3 CONCEITOS PRELIMINARES DO ALGORITMO

Nesta seção apresentamos alguns conceitos preliminares, apresentados em (BAIÃO, 2001, NAVATHE e RA, 1989, NAVATHE *et al.*, 1984) sobre o algoritmo de fragmentação vertical que utilizaremos nesta dissertação.

3.3.1 MATRIZ DE USO DOS ELEMENTOS

É uma extensão da Matriz de Uso dos Atributos, usada em (NAVATHE e RA, 1989, NAVATHE *et al.*, 1984), que também leva em consideração os métodos. Na Matriz de Uso dos Elementos são disponibilizadas informações sobre os elementos(E) utilizados em importantes transações (T) da aplicação. Na tabela 3.1 mostramos uma matriz com 8 transações e 10 elementos de uma classe, utilizado por (NAVATHE e RA, 1989, NAVATHE *et al.*, 1984). As informações destacadas na tabela 3.1 em negrito serão utilizadas no exemplo apresentado na seção 3.3.3.

A Matriz de Uso dos Elementos é composta pelas seguintes estruturas (detalhadas a seguir):

- Matriz Transações X Elementos
- Vetor Tipo
- Vetor Acesso

A Matriz de Transações X Elementos é definida como a matriz de dimensão k x n, formada por k transações e n elementos. Cada linha se refere a uma transação e as colunas são representadas pelos elementos. Cada entrada da matriz, que indicaremos por u_{ij}, representa a utilização, ou uso, do elemento j na transação i, de acordo com a seguinte definição:

$$u_{ij} = \begin{cases} 1, \text{ se a transação i usa o elemento j} \\ 0, \text{ caso contrário} \end{cases} \qquad (1)$$

No vetor "Tipo"[1] são distinguidas as transações de recuperação por R (Retrieval) das transações de atualização por U (Update). Cada entrada nesta coluna indica o tipo de transação (R ou U) para cada transação i, $1 \leq i \leq k$.

Para finalizar, o último vetor identifica o número de acessos por período de tempo para cada transação i, para $1 \leq i \leq k$, indicado por "Acc" (Access) e cada entrada da transação i, indicaremos por acc_i.

Desta maneira, a Matriz de Uso dos Elementos é definida como a matriz de dimensão k x n, formada por k transações e n elementos, onde cada entrada (i,j) é igual a u_{ij} e adicionada das colunas Tipo e Acc, para $1 \leq i \leq k$.

Tabela 3.1: Matriz de Uso dos Elementos

Transações x Elementos											Tipo	Acc
T \ E	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}		
T_1	1	0	0	0	1	0	1	0	0	0	R	25
T_2	0	**1**	1	0	0	0	0	**1**	1	0	R	**50**
T_3	0	0	0	1	0	1	0	0	0	1	R	25
T_4	0	**1**	0	0	0	0	1	**1**	0	0	R	**35**
T_5	1	**1**	1	0	1	0	1	**1**	1	0	U	**25**
T_6	1	0	0	0	1	0	0	0	0	0	U	25
T_7	0	0	1	0	0	0	0	0	1	0	U	25
T_8	0	0	1	1	0	1	0	0	1	1	U	15

3.3.2 MATRIZ DE AFINIDADES ENTRE OS ELEMENTOS

Na tabela 3.2 podemos verificar a Matriz de Afinidades entre os Elementos gerada a partir da Matriz de Uso dos Atributos apresentada na tabela 3.1. As informações destacadas na tabela 3.2 em negrito serão utilizadas no exemplo apresentado na seção 3.3.3.

[1] Apesar de saber que o tipo (recuperação e atualização) interfere no custo, nesta dissertação, simplificadamente, só consideraremos os acessos à informação.

Para definirmos a Matriz de Afinidades entre os Elementos (*Element Affinity Matrix*), chamaremos de aff_{ij} as entradas da matriz, representando o valor de afinidade entre os elementos E_i e E_j, da forma

$$aff_{ij} = \sum_{\tau} acc_k \quad , \text{onde} \quad \tau = \{ k \in T \mid u_{ki}=1 \wedge u_{kj}=1 \} \qquad (2)$$

onde $\sum_{\tau} acc_k$ representa a soma do número de acessos da transação k referenciando ambos os elementos i e j, ou seja, por (1), $u_{ki}=1$ e $u_{kj}=1$, e T representa todas as transações importantes da aplicação descritas na tabela 3.1.

Desta maneira, podemos definir a Matriz de Afinidades entre os Elementos pela matriz de dimensão n x n, formada por n elementos, onde cada entrada (i,j) é igual a aff_{ij}. Podemos notar que esta matriz é simétrica já que $aff_{ij}=aff_{ji}$, para todo i e j tal que $1 \leq i \leq n$ e $1 \leq j \leq n$. Além disso, os elementos da diagonal representados pelas entradas (i,i) para todo i, tal que $1 \leq i \leq n$, é definido segundo (2), como a soma do número de acessos de todas as transação que referenciam o elemento i, mostrando a ligação do elemento i em termos do seu uso em todas as transações. As demais entradas (i,j) indicam a ligação existente entre dois elementos que são usados juntos em uma ou mais transações.

Na fragmentação vertical queremos agrupar em um mesmo fragmento elementos que são acessados simultaneamente por um subconjunto de T e que tenham valores próximos (afinidades parecidas), indicando o que chamamos de grande grau de afinidade.

Tabela 3.2: Matriz de Afinidades entre os Elementos

E	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}
E_1	75	25	25	0	75	0	50	25	25	0
E_2	25	110	75	0	25	0	60	**110**	75	0
E_3	25	75	115	15	25	15	25	75	115	15
E_4	0	0	15	40	0	40	0	0	15	40
E_5	75	25	25	0	75	0	50	25	25	0
E_6	0	0	15	40	0	40	0	0	15	40
E_7	50	60	25	0	50	0	85	60	25	0
E_8	25	**110**	75	0	25	0	60	110	75	0
E_9	25	75	115	15	25	15	25	75	115	15
E_{10}	0	0	15	40	0	40	0	0	15	40

3.3.3 EXEMPLO

Através da tabela 3.1 podemos identificar que as transações T_2, T_4 e T_5 acessam simultaneamente os elementos E_2 e E_8 (destacadas em negrito na tabela 3.1). Utilizando a definição (2) da seção anterior, podemos calcular

$$aff_{28} = \sum_{k} acc_k \quad , \text{onde} \quad k \in \{2, 4 \text{ e } 5\}$$

Portanto, $aff_{28} = 50 + 35 + 25 = 110$. Na tabela 3.2 está destacado em negrito esta afinidade entre os elementos E_2 e E_8.

3.4 DEFINIÇÕES, NOTAÇÕES E TERMINOLOGIAS DO ALGORITMO

Esta seção descreve as definições, notações e terminologias necessárias para o entendimento do algoritmo que está sendo apresentado neste capítulo e que será mais detalhado nas próximas seções.

3.4.1 NÓS

Indicados por letras maiúsculas, acompanhadas ou não de índices (como A, B, C, D_i, E_3,...), simboliza os nós do grafo de afinidades a ser construído entre elementos da classe. Cada nó representará um elemento da classe a ser verticalmente fragmentada.

3.4.2 ARESTAS

Indicados por letras minúsculas, acompanhadas ou não de índices (como a, b, c, d_i, e_3, e_{ij},...), simboliza as arestas. Denominamos

$$a_{ij} = (Ei, Ej) \qquad (3)$$

a aresta entre os elementos, definidos na seção 2.2.3, E_i e E_j. Caso Ei seja representado pelo nó N_i e Ej pelo nó N_j, então podemos também referenciar a aresta a_{ij} por

$$a_{ij} = (N_i, N_j) \qquad (4).$$

3.4.3 VALOR DE AFINIDADE

O valor de afinidade da aresta a_{ij} é denotado por $p(a_{ij})$. Este valor, oriundo da Matriz de Afinidades entre os Elementos, é definido utilizando-se aff_{ij} determinada em (2) e a_{ij} determinada em (3), por

$$\text{se } a_{ij} = (Ei, Ej) \text{ então } p(a_{ij}) = aff_{ij} \qquad (5)$$

3.4.4 CICLOS

Se um ciclo é formado, por exemplo, pelos nós Ni, Nj e Nk, então identificaremos este ciclo por C(Ni, Nj, Nk) e conseqüentemente as arestas a_{ij}, a_{jk} e a_{ik} farão parte deste ciclo. Com ciclos com mais de três nós basta colocar a identificação de cada um deles.

Em geral, representaremos por

$$C(N_1, N_2, ..., N_m) \qquad (6)$$

o ciclo formado por N_1, N_2, ..., Nm, onde $m \leq n$, n é o número de elementos da classe, m o número de nós no ciclo e cada N_i, para $1 \leq i \leq m$, representa um elemento E_i da classe.

3.4.5 CICLO PRIMITIVO (*PRIMITIVE CYCLE*)

Indica qualquer ciclo no grafo de afinidades.

3.4.6 CICLO DE AFINIDADE (*AFFINITY CYCLE*)

Indica um Ciclo Primitivo que contém um Nó de Ciclo (explicado mais adiante). Nesta dissertação nós assumimos que um ciclo significa um Ciclo de Afinidade, a menos que este seja determinado de outra maneira (por exemplo, um ciclo pode tornar-se uma Partição Candidata, neste momento passa a não ser mais tratado como um Ciclo de Afinidade).

3.4.7 ARESTA FORMADORA DE CICLO (*CYCLE COMPLETING EDGE*)

Indica a aresta que, sendo selecionada, forma um ciclo no grafo.

3.4.8 NÓ DE CICLO (*CYCLE NODE*)

Indica o nó previamente selecionado que faz parte da Aresta Formadora de Ciclo, ou seja, dos nós da Aresta Formadora de Ciclo é aquele que foi inserido no grafo anteriormente ao outro nó.

3.4.9 ARESTA CORTADA (*CUT EDGE*)

Se tentamos estender um ciclo por uma aresta e não conseguimos, esta aresta se tornará uma aresta cortada para identificar que esta não pode ser mais usada para aumentar o ciclo. A aresta cortada também não pode ser utilizada para formar outros

ciclos, ou seja, a partir do momento que a aresta foi cortada esta não pode fazer parte de um ciclo qualquer, não podendo, por exemplo, ser utilizada para estender um ciclo. Graficamente será representada por uma aresta com dois traços (ver Fig. 3.1). As arestas cortadas são utilizadas para isolar partições candidatas, que quando isolada, torna-se uma partição (ver seção 3.5.3).

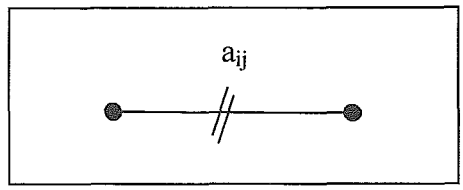


Fig. 3.1 – Aresta a_{ij} cortada

3.4.10 ARESTA DE EXTENSÃO (FORMER EDGE)

Indica a aresta previamente selecionada que não faz parte do ciclo porém contém o Nó de Ciclo, ou seja, esta aresta está conectada ao ciclo.

- Definição de NAVATHE (1989)

A Aresta de Extensão é definida por NAVATHE (1989) como a aresta que foi selecionada entre o último Corte e o Nó de Ciclo. Assim, a_{28} é aresta de extensão na Fig. 3.2 já que se encontra entre o Nó de Ciclo N_2 e a aresta cortada a_{78}.

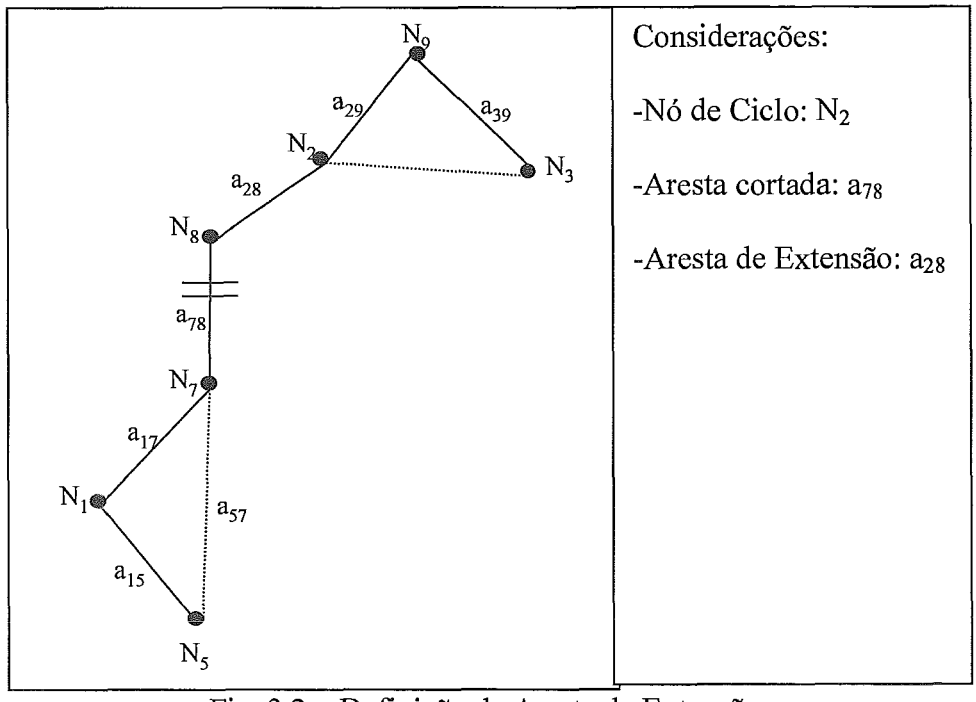


Fig. 3.2 – Definição de Aresta de Extensão

- Extensão da definição de NAVATHE (1989)

Nesta dissertação foi estendida a definição de NAVATHE (1989) para uma aresta previamente selecionada, não cortada, conectada diretamente ao ciclo. Com esta definição são abrangidas as arestas conectadas ao ciclo mas que não estão conectadas a uma aresta cortada. Para explicar o motivo desta mudança temos que usar conceitos que serão explicados na próxima seção mais detalhadamente, pois envolvem formação do ciclo (seção 3.5.1) e criação de partição (seção 3.5.3).

Veja a Fig. 3.3 e suas considerações. Pela definição de NAVATHE (1989), quando é formado o ciclo $C(N_2, N_9, N_3)$, a aresta a_{28} não é considerada aresta de extensão já que não existe aresta cortada que esteja conectadda a esta e, portanto, pela Possibilidade de Ciclo (seção 3.5.1), como não existe aresta de extensão é formado o ciclo $C(N_2, N_9, N_3)$ independente do valor da aresta a_{28}.

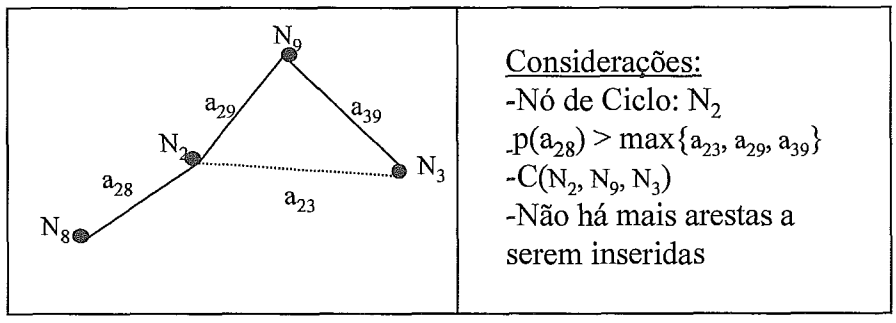


Fig. 3.3 – Extensão da definição de NAVATHE (1989)

Se neste momento acabarem as arestas candidatas para serem inseridas no grafo será formada uma partição com os nós do ciclo $C(N_2, N_9, N_3)$, porém o nó N_8 tem alta afinidade com os demais nós do ciclo. Como o objetivo do algoritmo é agrupar nós com alto valor de afinidade, de acordo com a definição original este objetivo não seria alcançado. Já se for considerada a extensão da definição de NAVATHE (1989), proposta nesta dissertação, quando formado o ciclo, a aresta a_{28} será considerada uma aresta de extensão e como o valor de afinidade da aresta, $p(a_{28})$, é maior do que todos os valores de afinidades das arestas do ciclo, não será satisfeita a possibilidade de extensão (isto porque, intuitivamente, esta aresta deveria fazer parte do ciclo). Como não é satisfeita a condição não será marcado o ciclo $C(N_2, N_9, N_3)$, nem o nó de ciclo N_2. Assim, acabando as arestas da lista de arestas candidatas, é formada uma partição com todos os nós que não pertencem a nenhum ciclo, envolvendo assim a aresta a_{28} corretamente.

3.4.11 ARESTA DO CICLO *(CYCLE EDGE)*

Indica quaisquer das arestas que participam do ciclo.

3.4.12 EXTENSÃO DO CICLO *(EXTENSION OF CYCLE)*

NAVATHE (1989), se refere à Extensão do Ciclo como o novo ciclo que estende o ciclo existente através do Nó de Ciclo, ou seja, o Nó de Ciclo será um dos nós de uma das novas arestas utilizadas pelo novo ciclo que representará a Extensão do Ciclo.

As definições acima mencionadas são usadas no algoritmo proposto para processar o grafo de afinidades e gerar os possíveis ciclos. O processo de Formação do Ciclo se tornará mais claro durante a explicação dos conceitos fundamentais envolvidos no algoritmo, que serão apresentados na seção a seguir.

3.5 CONCEITOS FUNDAMENTAIS DO ALGORITMO – ORIGINAIS E EXTENSÕES

Utilizando as definições e notações apresentadas na seção anterior, podemos explicar mais detalhadamente nesta seção os mecanismos de Formação do Ciclo, Extensão dos Ciclos, criação da partição e critério de parada do algoritmo.

3.5.1 FORMAÇÃO DO CICLO

Para explicar a formação de um ciclo, vamos considerar os exemplos mostrados na Fig. 3.4 e iremos, juntamente com a explicação dos exemplos, definir a condição: Possibilidade de Ciclo (*possibility of cycle*).

Podem acontecer duas situações diferentes, exemplificadas por (a) e (b) da Fig. 3.4, quando for formado um ciclo, que, no nosso exemplo, acontecerá quando for selecionada a aresta a_{13}:

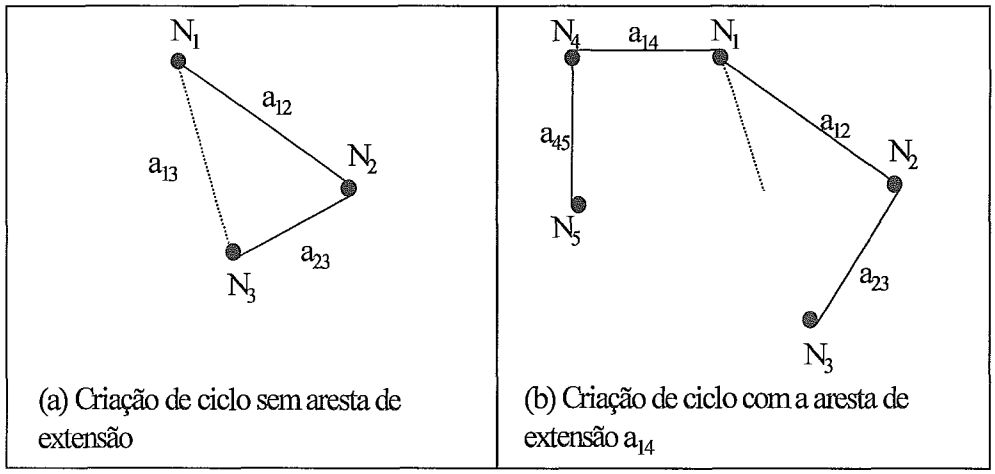
(a) Suponha que as arestas a_{12} e a_{23} já foram selecionadas e a_{13} será a próxima aresta a ser selecionada. Neste caso será formado um ciclo e não existe Aresta de Extensão (Fig. 3.4-a).

(b) Suponha que as arestas a_{45}, a_{14}, a_{12} e a_{23} já foram selecionadas e a_{13} será a próxima aresta a ser selecionada, ou seja, existe uma Aresta de Extensão indicado por a_{14}. Devemos ressaltar que o importante no momento é a existência

ou não de uma Aresta de Extensão, não importando se existe uma aresta, como a aresta a_{45}, que esteja conectada a Aresta de Extensão e que esta esteja conectada a outra e assim por diante (Fig. 3.4-b) .

Em ambos os casos, visto que a_{13} forma um ciclo, nós temos que verificar se este é um ciclo de afinidades.

Intuitivamente, um ciclo será um ciclo de afinidades se não houver uma Aresta de Extensão (ou seja, alguma aresta conectada ao ciclo), que deveria estar fazendo parte deste ciclo por possuir algum grau de afinidade com as demais arestas do ciclo.



(a) Criação de ciclo sem aresta de extensão

(b) Criação de ciclo com a aresta de extensão a_{14}

Fig. 3.4 – Criação de ciclo

Portanto, um ciclo será um ciclo de afinidades caso não exista uma aresta de extensão cujo valor seja maior ou igual a alguma aresta existente no ciclo. No caso (a) o ciclo será um Ciclo de Afinidade já que não há Aresta de Extensão e no caso (b) será um Ciclo de Afinidade somente se $p(a_{14}) \leq \{p(a_{12}), p(a_{23}), p(a_{13})\}$, ou equivalentemente,

$$p(a_{14}) \leq m, \text{ onde } m = \min\{p(a_{12}), p(a_{23}), p(a_{13})\}$$

(se $p(a_{14})$ for menor ou igual do que o mínimo entre $p(a_{12})$, $p(a_{23})$ e $p(a_{13})$ então será menor ou igual a todas as arestas do ciclo).

Portanto, formado um ciclo, deveremos testar a condição Possibilidade de Ciclo definida na Fig. 3.5.

Não existe Aresta de Extensão ou p(Aresta de Extensão) ≤ p(todas as arestas do ciclo formado)

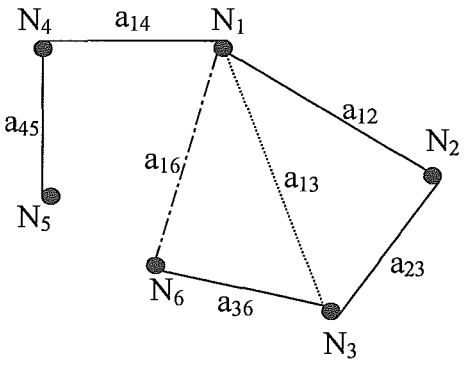
Fig. 3.5 - Definição de Possibilidade de Ciclo

Se a condição Possibilidade de Ciclo for satisfeita então este ciclo será chamado de Ciclo de Afinidade. Quando encontrarmos um Ciclo de Afinidade, marcaremos este como Partição Candidata e o nó incidente à aresta que, quando selecionada, formou o ciclo, e que foi previamente inserido no grafo será chamado de Nó de Ciclo (conforme seção 3.4.8). Voltando ao exemplo da Fig. 3.4, se a Possibilidade de Ciclo for satisfeita em ambos os casos teremos a Partição candidata formada pelo Ciclo C (N_1, N_2 e N_3) e o Nó de Ciclo será N_1. Além disso em (b) guardaremos como Aresta de Extensão a aresta a_{14}.

3.5.2 EXTENSÃO DO CICLO

Para explicar sobre a extensão de um ciclo, vamos considerar o exemplo da Fig. 3.6 e iremos, juntamente com a explicação do exemplo, definir a condição: Possibilidade de Extensão.

Considerando o exemplo da Fig. 3.4-(b), suponha que o ciclo C(N1, N2, N3) satisfez a condição Possibilidade de Ciclo e portanto N1 foi marcado como Nó de Ciclo e o mesmo ciclo foi marcado como Partição Candidata. Suponha ainda que a próxima aresta a ser selecionada (ou seja, a aresta de maior valor, dentre as arestas candidatas, que possa ser conectada linearmente à árvore) seja a aresta a_{36} (veja Fig. 3.6). Neste momento, a_{36} é indicada como uma aresta potencial para ser utilizada na extensão da Partição Candidata em questão. Para verificar a possibilidade de extensão do ciclo pela aresta a_{36}, deveremos levar em consideração esta aresta e a aresta a_{16} (indicada na Fig. 3.6 pela aresta ponto-tracejada) que conecta o nó N_6, que foi o último inserido, com o Nó de Ciclo



Considerações:

-$p(a_{14}) \leq \min\{p(a_{12}), p(a_{23}), p(a_{13})\}$
-$p(a_{36})$ ou $p(a_{16}) \geq \min\{p(a_{12}), p(a_{23}), p(a_{13})\}$
-Aresta de Extensão: a_{14}
-Nó de Ciclo : N_1
-Partição Candidata : $C(N_1, N_2, N_3, N_6)$
-Nova aresta selecionada (aresta a ser considerada) : a_{36}
-Aresta formadora de ciclo: a_{16}

Fig. 3.6 – Extensão do ciclo

representado por N_1. Intuitivamente, se uma destas arestas (a_{36} ou a_{16}) tiver um grande grau de afinidade com as arestas do ciclo (o Partição Candidata) então deveremos estendê–lo. Ou seja, se $p(a_{36})$ ou $p(a_{16})$ for maior ou igual ao valor de alguma das arestas do ciclo a extensão deverá ser realizada formando o novo Ciclo $C(N_1, N_2, N_3, N_6)$. Basta, portanto, que

$$p(a_{36}) \text{ ou } p(a_{16}) \geq m, \text{ onde } m = \min\{p(a_{12}), p(a_{23}), p(a_{13})\}..$$

Segundo NAVATHE (1989), a aresta selecionada (apontada na Fig. 3.6 por a_{36}) é chamada de Aresta a Ser Considerada e a outra aresta que conecta o nó inserido ao Nó de Ciclo(apontada na Fig. 3.6 por a_{16}) é chamada de Aresta Formadora de Ciclo. Formalmente teremos a definição de Possibilidade de Extensão mostrada na Fig. 3.7.

p(Aresta a Ser Considerada) ou p(Aresta Formadora de Ciclo) ≥ p(alguma aresta do ciclo formado)

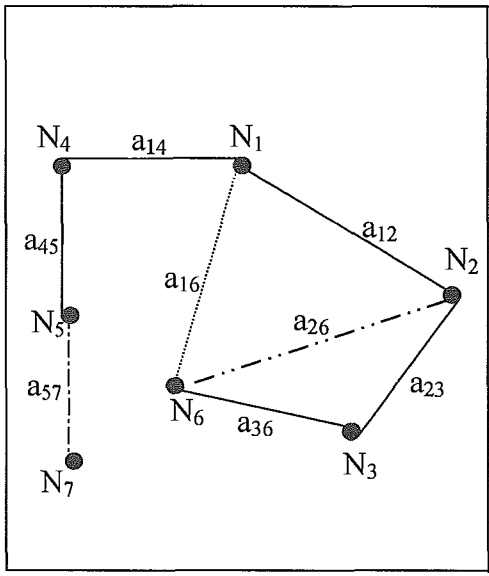
Fig. 3.7 – Definição de Possibilidade de Extensão

Pelo algoritmo de fragmentação vertical (NAVATHE E RA, 1989) que estamos descrevendo ao longo deste capítulo, após a formação de uma Partição Candidata, iremos tentar sempre estendê-lo, através das próximas arestas selecionadas, e iremos descartar as arestas selecionadas que não têm potencial para estender o ciclo. No exemplo da Fig. 3.4, selecionamos uma aresta (aresta a_{36}, indicada pela aresta ponto-tracejada) que tinha potencial para estender o ciclo. Agora veremos dois exemplos em que a aresta selecionada não tem potencial para estender o ciclo e devemos proceder de

duas maneiras distintas, diferenças estas que não estão explícitas em (NAVATHE E RA, 1989) e que foram verificadas durante a implementação do algoritmo.

A todo momento atualizamos uma lista de arestas candidatas que representam as arestas que ainda podem ser selecionadas para serem inseridas no grafo. Analisaremos agora duas situações referentes a lista mencionada:

(a) Suponha que foi selecionada a aresta a_{26} (indicada na Fig. 3.8 pela aresta dois pontos-tracejada). É formado assim o ciclo $C(N_2, N_3, N_6)$, porém pela definição de Extensão do Ciclo (seção 3.2.4), como o mesmo não contém o Nó de Ciclo N_1, este ciclo não será um possível ciclo de extensão do ciclo anterior e portanto a aresta a_{26} não será uma aresta potencial de extensão do ciclo. Porém como esta aresta é formada por dois nós que já fazem parte de um Partição Candidata (valendo o mesmo também se fizessem parte de uma partição, lembrando que formamos uma partição quando não podemos mais estender o ciclo) em nenhum outro momento poderemos utilizar esta aresta e portanto esta deve ser excluída da lista de arestas candidatas.



Considerações:

-$p(a_{14}) \leq \min\{p(a_{12}), p(a_{23}), p(a_{13})\}$

-$p(a_{16})$ ou $p(a_{36}) \geq \min\{p(a_{12}), p(a_{23}), p(a_{13})\}$

-Aresta de Extensão: a_{14}

-Nó de Ciclo : N_1

-Partição Candidata: $C(N_1, N_2, N_3, N_6)$

-Nova aresta selecionada: a_{26}

Fig. 3.8 – Arestas sem potencial para extensão

(b) Suponha que foi selecionada a aresta a57 (indicada na Fig. 3.8 pela aresta ponto-tracejada). Como esta aresta não está conectada a um nó do ciclo já existente, esta não será uma aresta potencial para extensão do ciclo. Porém como esta aresta não é formada por nós que já fazem parte de um Partição Candidata(ou de uma partição) poderemos utilizá-la num outro momento (por exemplo, logo depois de formar uma partição) e portanto esta deve ser excluída

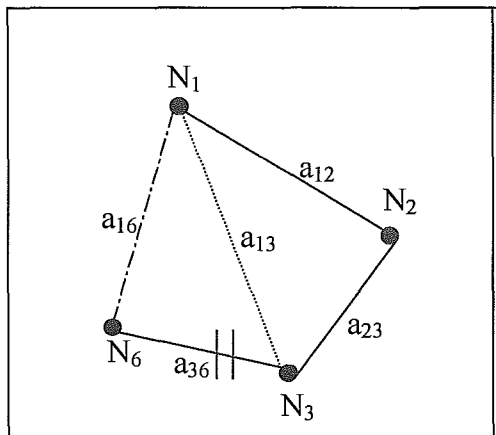
provisoriamente da lista de arestas candidatas. Após a formação da partição associada a esta Partição Candidata, todas as arestas excluídas provisoriamente são re-inseridas na lista de arestas candidatas novamente.

O processo continuará, selecionando novas arestas, e o término do algoritmo será descrito mais adiante em "Critério de parada do algoritmo".

3.5.3 CRIAÇÃO DA PARTIÇÃO

O próximo conceito a ser explicado corresponde ao relacionamento entre um ciclo e uma partição. Há quatro maneiras de ocorrer a fragmentação, sendo que em (NAVATHE E RA, 1989) só as maneiras (a) e (b) foram explicitadas. A (c) foi verificada em (BAIÃO 2001) e (d) foi verificada através da implementação do algoritmo, sendo mais uma contribuição desta dissertação.

(a) Criando uma partição através de uma nova aresta:



Considerações:

-Não existe Aresta de Extensão

-Nó de Ciclo: N_1

-Partição Candidata : $C(N_1, N_2, N_3)$

-Nova aresta selecionada: a_{36}

-Aresta Formadora de Ciclo: a16

Fig. 3.9 – Partição através de uma nova aresta

Veja a Fig. 3.9 e suas considerações apontadas. Se a próxima aresta selecionada (por exemplo a aresta a_{36}) não foi considerada anteriormente, nós a chamamos de nova aresta. Quando já houver uma Partição Candidata (no caso, o ciclo $C(N_1,N_2,N_3)$) e não existir Aresta de Extensão, se a nova aresta não satisfaz a Possibilidade de Extensão, então verificamos a Possibilidade de Extensão pela aresta que completa o ciclo (Aresta Formadora de Ciclo, que no exemplo é a aresta ponto-tracejada a16, da Fig. 3.9). Se a condição não for satisfeita também, é produzido um corte (cut) na aresta (chamada de

Aresta Cortada, representado em a_{36} por dois traços cortando a aresta) isolando o ciclo C(N1, N2, N3). Este ciclo agora será considerado uma partição.

Se há Aresta de Extensão devemos tentar estender o ciclo pela Aresta de Extensão. Esta discussão é apresentada a seguir no ítem (b).

(b) Criando uma partição através de uma Aresta de Extensão

Suponhamos que após o corte em a_{36} similar ao ítem em (a) (ou seja, não houve possibilidade de extensão do ciclo $C(N_1, N_2, N_3)$ pela aresta a_{36} e esta foi cortada), houvesse uma Aresta de Extensão. Na Fig. 3.10 a Aresta de Extensão é apontada por a_{14}. A partição, ao contrário de (a) onde não há Aresta de Extensão, não seria ainda formada e então, de acordo com (NAVATHE e RA, 1989) – ver apêndice D - mudaremos o Nó de Ciclo anterior para o Nó de Ciclo que faz parte da Aresta Cortada (N_3), e verificaremos a Possibilidade de Extensão do ciclo pelo Aresta de Extensão. Por exemplo, na Fig. 3.10, suponha que a_{12}, a_{23}, e a_{13} formem o ciclo $C(N_1, N_2, N_3)$ onde o Nó de Ciclo seja N_1, exista um corte em a_{36} e que a Aresta de Extensão seja a_{14}. Então, o Nó de Ciclo N_1 é modificado para N_3 e verificaremos a possibilidade de estender o ciclo $C(N_1, N_2, N_3)$ para o ciclo $C(N_1, N_2, N_3, N_4)$ que contém a_{14}(a Aresta de Extensão).

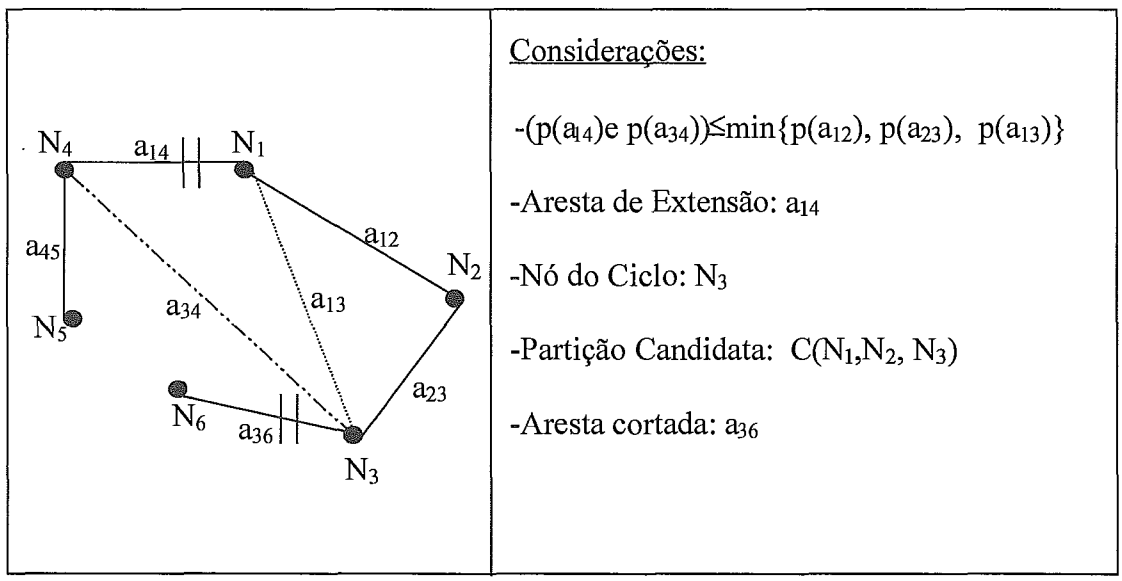


Figura 3.10 – Partição através da Aresta de Extensão

Assuma que a_{14} e a_{34} não satisfazem a possibilidade de extensão, isto é,

$$p(a_{14}) \text{ e } p(a_{34}) \leq \min\{p(a_{12}), p(a_{23}), p(a_{13})\}.$$

Então deveremos proceder da seguinte maneira (ilustrada na Fig. 3.10):

1. Faremos um corte na aresta a_{14} ;
2. N_3 permanece como Nó de Ciclo;
3. $C(N_1, N_2, N_3)$ torna-se uma partição (já que isolamos o ciclo por duas arestas cortadas).

Alternativamente, se a possibilidade de extensão fosse satisfeita, o resultado seria (Fig. 3.11).

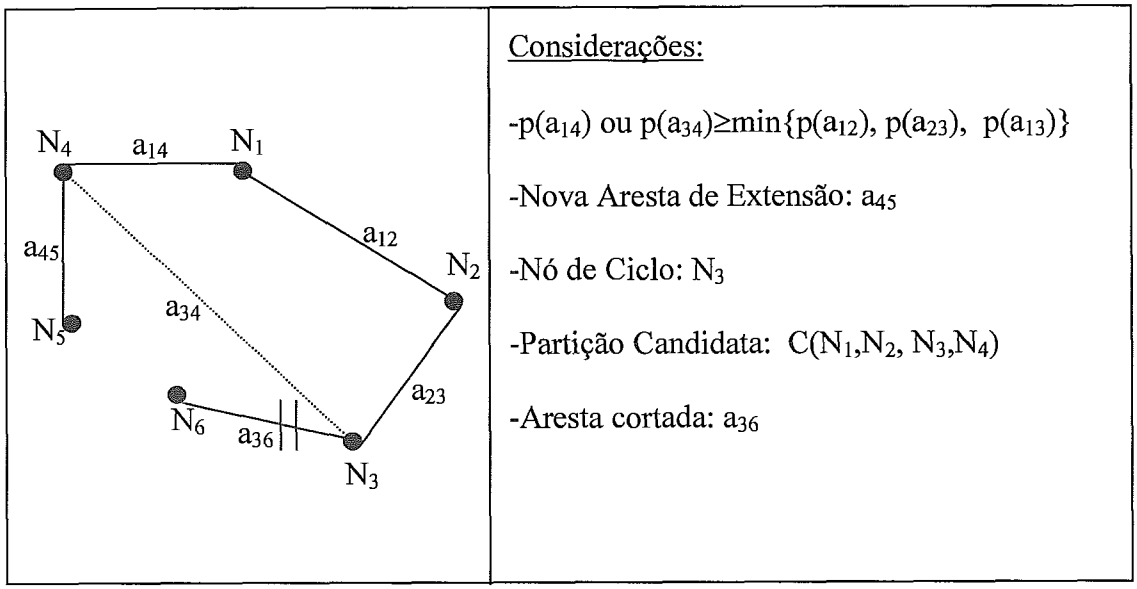


Fig. 3.11 – Ciclo é estendido e não é formada partição

1. $C(N_1, N_2, N_3)$ é estendido pelo ciclo $C(N_1, N_2, N_3, N_4)$;
2. N_3 permanece como Nó de Ciclo;
3. Nenhuma partição é ainda formada e $C(N_1, N_2, N_3, N_4)$ será a nova Partição Candidata;
4. Será apontado a nova Aresta de Extensão a_{45}.

Intuitivamente, o algoritmo tenta estender o ciclo considerando as novas arestas ou Aresta de Extensão que possa expandir a área interna do ciclo. Por exemplo, na Fig. 3.9 nós tentamos "aumentar o ciclo" de área interna formada por $C(N_1, N_2, N_3)$ para a área interna do ciclo $C(N_1, N_2, N_3, N_6)$ considerando a novas arestas a_{16} e a_{36}. Na Fig. 3.10, nós trocamos o Nó de Ciclo de N_1 para N_3 e então tentamos aumentar a área interna do ciclo $C(N_1, N_2, N_3)$ para a área interna do ciclo $C(N_1, N_2, N_3, N_4)$ através da Aresta de Extensão a_{14}.

(c) Criando uma partição após a última aresta ser inserida

Após a inserção da última aresta, ou seja, quando não houver mais arestas na lista de arestas candidatas, é formado, segundo (BAIÃO 2001), uma partição com todos os nós que não fazem parte das partições encontradas anteriormente. Este fragmento adicional é justificado já que assim é reduzido o número de partições(agrupando elementos com menor frequência de uso em uma única partição, ao invés de definir uma partição distinta para cada um desses elementos), e eliminando o gerenciamento sobre mais fragmentações verticais de uma classe que contém dados menos usados.

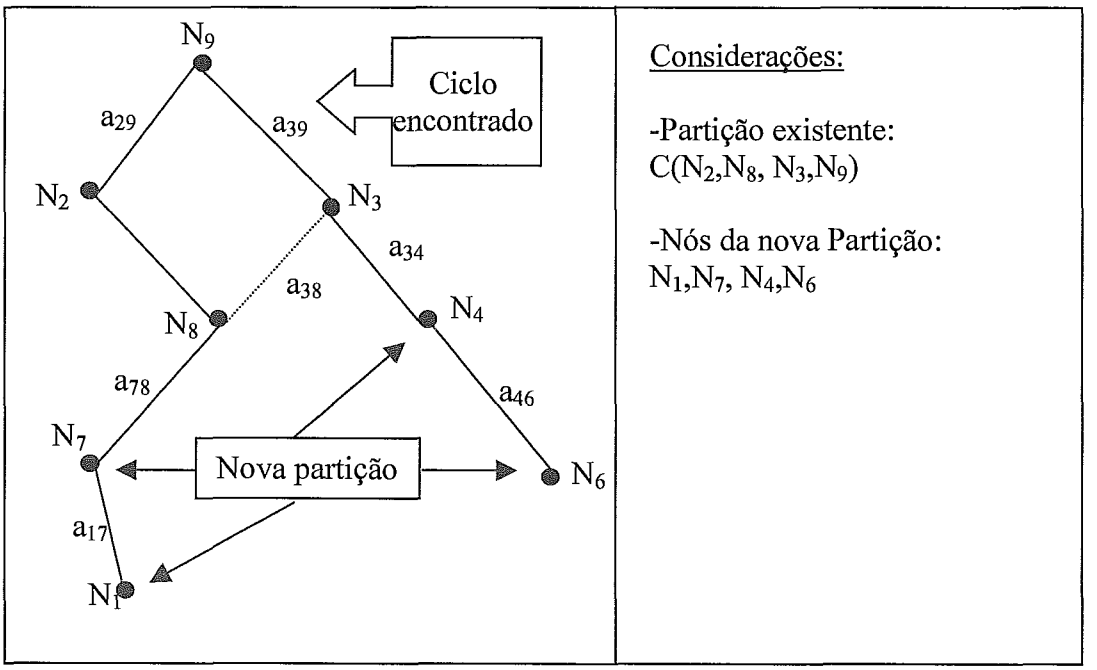
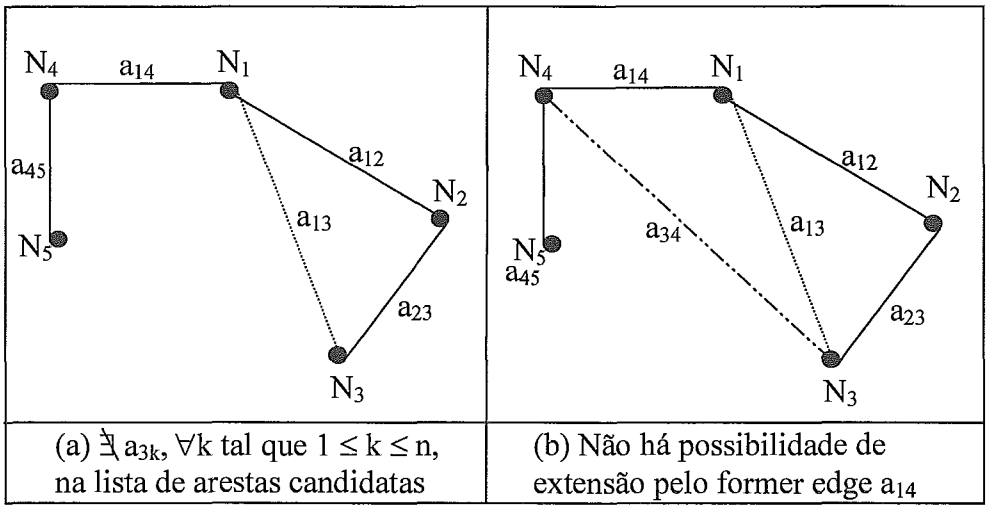


Fig. 3.12 – Criação de partição após a última aresta inserida

Na Fig. 3.12, é exemplificada esta situação. Suponha que após a inserção da última aresta, tenha sido encontrada, pelo algoritmo, apenas a partição formada pelos elementos representados pelos nós pertencentes ao ciclo $C(N_2,N_8,N_3,N_9)$. É então criada uma partição com os elementos representados pelos nós N_1,N_7, N_4 e N_6 , já que estes não participam de nenhuma partição anteriormente encontrada. O algoritmo então é finalizado e as duas partições encontradas neste exemplo serão formadas pelos elementos representados respectivamente por N_2,N_8, N_3 e N_9, e a outra por N_1,N_7, N_4 e N_6.

(d) Criando uma partição se não houver aresta para ser conectada à partição

Uma outra maneira de ser encontrada uma partição é caso não haja mais arestas, na lista de arestas candidatas, que possa ser conectada na ponta do grafo que contém uma partição candidata, pois desta maneira não haverá mais arestas que podemos tentar estender a partição candidata em questão.



(a) $\nexists a_{3k}, \forall k$ tal que $1 \leq k \leq n$, na lista de arestas candidatas

(b) Não há possibilidade de extensão pelo former edge a_{14}

Fig. 3.13 – Criação de partição se não houver aresta para ser conectada à partição

Considere que o grafo formado na Fig. 3.13 – (a) é relativo a uma classe que contém n elementos. Se não existe mais nenhuma aresta que possa ser conectada ao nó N3, ou seja, não existe aresta a_{3k}, $\forall k$ tal que $1 \leq k \leq n$, na lista de arestas candidatas, marcamos então a partição candidata formada pelo ciclo $C(N_1, N_2, N_3)$ como partição (Fig. 3.13 – (a)).

Caso não verificássemos a condição acima mencionada, todas as arestas da lista de arestas candidatas poderiam somente ser conectadas a outra ponta (ou seja, no exemplo da Fig. 3.13 – (a) só poderiam haver arestas que pudessem ser conectadas ao nó N_5) e portanto seriam conectadas, mas como não haveria possibilidade de extensão pela mesma (já que não estaria conectada à partição candidata), estas iriam para a lista de arestas removidas temporariamente e removidas da lista de arestas candidatas. Somente quando a lista das arestas candidatas estivesse vazia que formaria esta partição e todas as arestas da lista das arestas removidas temporariamente voltariam a ser carregadas para a lista de arestas candidatas.

Para diminuir todo esse processamento, antes de selecionar uma aresta da lista de arestas candidatas, verificamos se existe ainda pelo menos uma aresta a ser conectada na

ponta do grafo onde há a partição candidata. Se não houver, podemos já formar a partição e carregar todas as arestas da lista de arestas removidas temporariamente para a lista de arestas candidatas.

Não haveria portanto como estender por uma aresta nova (que não se encontra ainda no grafo), mas uma outra pegunta poderia ser feita: não poderia haver possibilidade de extensão pela aresta de extensão? A resposta é não e utilizando a Fig. 3.13 – (b) podemos exemplificar e explicar este resposta. Para haver possibilidade de extensão pela aresta de extensão, indicada por a_{14} na Fig. 3.13 – (b), deveria existir a aresta a_{34} na lista de arestas candidatas, porém se esta existisse não seria satisfeita a condição identificada na Fig. 3.13 – (a), ou seja existiria $k = 4$, tal que a_{34} pertenceria a lista de arestas candidatas e portanto ainda haveria uma aresta que poderia ser conectada ao nó N_3.

3.5.4 CRITÉRIO DE PARADA DO ALGORITMO

O processo continuará até que a lista de arestas candidatas esteja vazia. Esta é uma importante contribuição desta dissertação já que em (NAVATHE e RA, 1989) o processo continuaria até serem inseridos todos os nós. Durante a implementação do algoritmo constatamos que se o algoritmo parasse quando acabassem os nós, alguns ciclos poderiam não ser achados já que para formar os ciclos ainda precisariam ser inseridas arestas. Na Fig. 3.14 está ilustrada esta situação, inclusive o exemplo utilizado é o mesmo encontrado em (NAVATHE e RA, 1989) e apresentado nas tabelas 3.1 e 3.2. Considerando cada nó N_i representando um elemento E_i da tabela 3.2 e parando o algoritmo quando acabassem os nós, o ciclo $C(N_8, N_2, N_9, N_3)$ é encontrado porém os ciclos $C(N_5, N_1, N_7)$ e $C(N_4, N_6, N_{10})$ não são encontrados pois o algoritmo pára antes de serem selecionadas as arestas a_{57} e $a_{4\,10}$ (representadas pelo ponto-tracejado).

Esta modificação apesar de aumentar a complexidade do algoritmo para $O(n^3)$, é importante já que tornou possível a obtenção de uma fragmentação melhor, ou seja de menor custo, o que no contexto deste trabalho é mais relevante do que o tempo de execução.

Outra possibilidade seria criar um procedimento especial após a inserção de todos os nós no grafo, para inserir arestas ainda não selecionadas e caso fosse criado ciclos, tentar estender esses ciclos pela aresta de extensão. Este proceddimento especial deveria ser feito para as duas pontas da árvore. Isto se deve ao fato de que as arestas que ainda

poderiam ser inseridas só podem ser conectadas a uma das duas pontas da árvore já que é uma árvore geradora linearmente conectada.

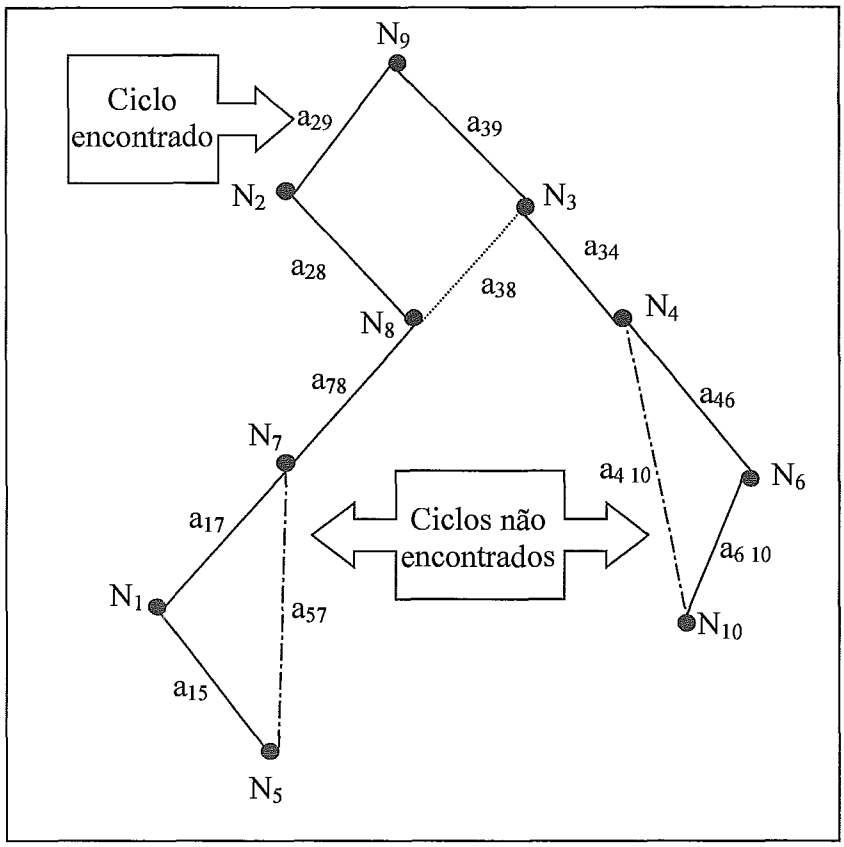


Fig. 3.14 – Critério de parada

3.6 ESTRUTURA GERAL

Nesta seção apresentaremos a estrutura geral do algoritmo, ou seja, uma visão resumida do algoritmo (sem entrar em muitos detalhes) levando em consideração as extensões propostas nas seções anteriores. Na próxima seção será apresentado o pseudo-código do algoritmo detalhado (incluindo exemplos) e para que haja um melhor entendimento do mesmo é necessário saber a estrutura geral e também todas as outras informações dadas nas seções anteriores (conceitos fundamentais, notações, definições, etc). Devemos ressaltar também que no capítulo 4 é encontrado um fluxograma detalhado sobre o algoritmo (seção 4.4.2) que pode ser visto após a leitura desta seção.

Resumidamente, a construção do grafo para a geração das partições é feita seguindo os seguintes passos:

Passo 1: Escolhe um nó qualquer para começar;

Passo 2: A seleção da próxima aresta deve ser linearmente conectada com a árvore geradora já construída e deve ter o maior valor entre todas as possibilidades;

Passo 3: Se é formado um Ciclo Primitivo então é verificada a existência do Nó de Ciclo (o que indica se um ciclo já foi formado anteriormente e que está na fase de tentar estender o ciclo):

- Passo 3.1: Se não existe Nó de Ciclo é verificada a Possibilidade de um Ciclo (Fig. 3.5) e se este existe, o ciclo é marcado como Ciclo de Afinidade e Partição Candidata. Depois selecione a próxima aresta (passo 2).
- Passo 3.2: Se existe Nó de Ciclo (o que indica que já existe um ciclo), descarte a aresta selecionada e selecione uma nova aresta (passo 2). Isto porque queremos tentar estender o ciclo já formado anteriormente. Devemos ressaltar as seguintes modificações: se a aresta selecionada forma um ciclo, então devemos verificar se o nó novo que será inserido no grafo (faz parte da aresta selecionada mas não é um nó que seja ponta do grafo) faz parte de uma partição já existente ou de uma partição candidata. Neste caso, esta aresta deve ser excluída da lista de arestas candidatas. Pode acontecer também de outro ciclo ser formado do outro lado do grafo. Neste caso, esta aresta deve ser excluída temporariamente.

Passo 4: Se não é formado um ciclo e existe uma Partição Candidata então:

- Passo 4.1: Verifique a possibilidade de extensão do ciclo (Fig. 3.7) pela nova aresta (a última selecionada no passo 2). Se houver esta possibilidade, estenda o ciclo e vá para o passo 2.
- Passo 4.2: Se não houver possibilidade de extensão do ciclo pela nova aresta, verifique a existência de Aresta de Extensão.
  - → Passo 4.2.1: Se não houver a possibilidade de extensão pela nova aresta, a nova aresta está conectada à partição candidata e não há Aresta de Extensão no momento, corte esta aresta e considere o ciclo como uma partição (Fig. 3.9). Vá para o passo 2.
  - → Passo 4.2.2: Se não houver a possibilidade de extensão pela nova aresta, a nova aresta está conectada à partição candidata e existir uma Aresta de Extensão, troque o Nó de Ciclo de lugar(para o outro nó que faz parte do Aresta Formadora de Ciclo e não é o Nó de Ciclo) e verifique a possibilidade de extensão do ciclo utilizando a Aresta de Extensão. Se não existe esta

possibilidade, corte a Aresta de Extensão e considere o ciclo como uma partição, pois assim já que não conseguimos estender através da nova aresta nem da Aresta de Extensão, isolamos a partição candidata (Fig. 3.10). Caso contrário, estenda o ciclo (que será a nova partição candidata), atualize a Aresta de Extensão e vá para o passo 2.

→ Passo 4.2.3: Se não houver possibilidade de extensão pela nova aresta pois esta não está conectada à partição candidata, simplesmente devemos retirar a aresta selecionada da liasta de arestas candidatas e inserir na lista de arestas removidas temporariamente.

3.7 PSEUDO-CÓDIGO

Nesta seção apresentaremos o pseudo-código do algoritmo detalhado (incluindo exemplos) e levaremos em consideração todas as extensões mencionadas nas seções anteriores. Para a leitura desta seção é importante ter compreendido todas as informações dadas nas seções anteriores deste capítulo e ter em mente a estrutra geral apresentada na seção anterior.

3.7.1 ALGORITMO DE FRAGMENTAÇÃO VERTICAL

O algoritmo usado para a Fragmentação Vertical, que é o foco desta dissertação, é executado para cada classe a ser verticalmente fragmentada através da função FragmentacaoVertical (Fig. 3.15).

```
função FragmentacaoVertical(Cv: classes a serem verticalmente fragmentadas,
Oproj: conjunto das operações de projeção,
T: conjunto das transações, frequência e operações acessadas)
retorna Fv: conjunto das partições das classes
Início
  cria Fv vazia                                                    (1)
  para cada classe Ck ∈ Cv do                                      (2)
  início
    U: = ConstrucaoMatrizUsoElementos(Ck,Oproj,T)                  (3)
    M: = ConstrucaoMatrizAfinidadesElementos(U)                    (4)
    particoesCk : = ConstrucaoParticaoGrafoAfinidadesElementos(Ck,M)  (5)
    Fv : = Fv + particoesCk                                        (6)
  fim
  return Fv
Fim
```

Fig. 3.15 – Algoritmo de Fragmentação Vertical

O conjunto das classes que serão verticalmente fragmentadas é apontado por Cv. Para cada classe Ck, existente em Cv, será construída a Matriz de Uso de Elementos (linha 3 da Fig. 3.15), conforme a definição encontrada na seção 3.2.4.1, utilizando as informações sobre as transações (atributos e chamadas de métodos), operações de projeção acessadas e frequências (indicadas por T e Oproj na Fig. 3.15).

A Matriz de Uso dos Elementos será utilizada para a construção da Matriz de Afinidades entre os Elementos através da função ConstrucaoMatrizAfinidadesElementos e retornará em particoesCk todas as partições (ou fragmentações) verticais da classe Ck (linha 4 da Fig. 3.15).

3.7.2 CONSTRUÇÃO DA MATRIZ DE USO DOS ELEMENTOS

A funcao ConstrucaoMatrizUsoElementos (Fig. 3.16) constrói a Matriz de Uso entre os Elementos conforme a definição apresentada na seção 3.2.4.2. Nesta matriz são disponibilizadas informações sobre os elementos utilizados em importantes transações da aplicação.

```
função ConstrucaoMatrizUsoElementos(Ck: classe a ser verticalmente fragmentada,
                                     Oproj: conjunto das operações de projeção,
                                     T: conjunto das transações, frequência e operações acessadas)
retorna U: matriz de uso entre os elementos de Ck
Início
 inicializa U(Ti,Ej) : = 0 e U(Ti,Acc) : = 0, ∀ Ti ∈ T e Ej ∈ Ck                (8)
 para cada Ti ∈ T faca                                                          (9)
  para cada Op ∈ Ti e Op ∈ Oproj faça                                           (10)
   para cada elemento Ej ∈ Ck acessado por Op faça                              (11)
   início
    acci : = numero de acessos por período de tempo de Ti tal que Op ∈ Ti       (12)
    U(Ti,Ej) : = 1                                                              (13)
    U(Ti, Acc) : = acci                                                         (14)
   fim
Fim
```

Fig. 3.16 – Construção da Matriz de Uso dos Elementos

3.7.3 CONSTRUÇÃO DA MATRIZ DE AFINIDADES ENTRE OS ELEMENTOS

A função ConstrucaoMatrizAfinidadesElementos (Fig. 3.17) constrói a matriz de afinidades entre os elementos da classe conforme a definição encontrada na seção 3.2.4.2. Os elementos da classe representam as dimensões da matriz(linhas e colunas).

```
função ConstrucaoMatrizAfinidadesElementos (U: matriz de uso dos elementos)
retorna M: matriz de afinidade entre os elementos
Início
 inicializa M(Ei,Ej) : = 0 ∀ ei, ej ∈ M                                    (15)
 para cada Ti ∈ U faça                                                     (16)
  para cada Ej ∈ Ti faça                                                   (17)
   para cada Ek ∈ Ti faça                                                  (18)
       se (U(Ti,Ej) = 1 e U(Ti,Ek) = 1) então                              (19)
     M(Ej,Ek) := M(Ej,Ek) + U(Ti,Acc)                                      (20)
Fim
```

Fig. 3.17 – Construção da Matriz de Afinidade entre os Elementos

Cada valor $M(E_i, E_j)$ da matriz de afinidades entre os elementos representa a soma das número de acessos por período de tempo (ou frequência) das operações que acessam os elementos E_i e E_j simultaneamente.

3.7.4 CONSTRUÇÃO DO GRAFO DE AFINIDADES ENTRE OS ELEMENTOS

A função ConstrucaoParticaoGrafoAfinidadesElementos[2] (Fig. 3.18) define as partições verticais da classe através da construção de um grafo de afinidades entre os atributos e métodos, chamados de elementos, que serão representados pelos nós do grafo. Cada ciclo resultante no grafo representa uma partição. Esta abordagem, baseada em grafos, agrupa os elementos em ciclos que representarão as partições verticais da classe. Durante o algoritmo, que tem como objetivo construir o grafo e encontrar os ciclos formados, cada nó do grafo contém uma grande afinidade com os nós do ciclo onde este se encontra, porém baixa afinidade em relação aos nós dos demais ciclos ou nós que não se encontram em ciclo algum.

A) Linhas 1-7 da Fig. 3.18: inicializações

Na Fig. 3.18 é mostrado o pseudo-código detalhado do algoritmo. Nesta seção iremos explicar todo o pseudo-código e mostraremos a seguir a idéia da construção do grafo de afinidades:

Chamaremos o grafo que está sendo construído de G = (N,A) onde N é o conjunto de nós do grafo e A o conjunto de arestas (linhas 1 a 3 da Fig. 3.18). Cada nó do grafo representa um elemento da classe a ser verticalmente fragmentada. Esta classe é representada por Ck. Selecionamos primeiramente qualquer elemento de Ck (passo 1 da seção 3.6) e inserimos no grafo (linha 4 da Fig. 3.18), ou seja, é inserido o elemento

[2] Os casos apontados na Fig. 3.18, como por exemplo na linha 28, serão utilizados no próximo capítulo.

selecionado no conjunto N de nós do grafo (NAVATHE (1989) mostra que o resultado é o mesmo, independente do nó inicial).

São criadas inicialmente duas listas vazias: lista de Arestas Candidatas (AC) e lista de Arestas Removidas Temporariamente (AR) (linhas 5 e 6 da Fig. 3.18). A lista de Arestas Candidatas (AC) contém as arestas que podem ser selecionadas para serem inseridas no grafo. Inicialmente são inseridas na lista AC todas as arestas a_{ij}, que conectam os nós E_i e Ej e tem valor $p(a_{ij})=M(E_i, E_j)$, representadas na Matriz de Afinidades entre os Elementos, não nulo (linha 7 da Fig. 3.18). A lista de Arestas Removidas Temporariamente (AR) contém as arestas que num certo momento não podem ser inseridas porém após a definição da partição estas devem ser consideradas novamente. Este momento é justamente quando estamos tentando estender uma partição candidata até que não haja mais possibilidades para estendê-la e considere a mesma como partição. Após a criação da partição todas as arestas que estão em AR serão inseridas em AC (simbolizado por AC+=AR) e AR ficará vazia (AR=nil). Durante o pseudo-código serão apontadas as situações onde isto ocorre. NAVATHE (1989) somente descreve que se uma aresta não pode ser utilizada esta deve ser descartada, não considerando a diferença entre as arestas que devem ser descartadas e não serão mais utilizadas como arestas candidatas e as arestas, que após a criação de uma partição, devem ser consideradas novamente (estas serão guardadas em AR). Estes dois modos diferentes de descartar uma aresta são uma contribuição desta dissertação.

B) Linhas 8-102 da Fig. 3.18: criação do grafo de afinidades e ciclos (partições)

Enquanto a lista das arestas candidatas (AC) não estiver vazia (AC = nil), correspondente a linha 8 da Fig. 3.18, são feitos diversos procedimentos detalhados a seguir. NAVATHE (1989) utilizava como critério de parada a inserção de todos os elementos da classe, porém na seção 3.2.5.4 em "Critério de parada do algoritmo" foi discutido e apresentado as razões para tal mudança, bem como um exemplo para ilustrar o problema.

```
        função ConstrucaoPartiçãoGrafoAfinidadesElementos (Ck: classe a ser verticalmente
               fragmentada, M: matriz de afinidade entre os elementos)
        retorna particoesCk: conjunto das partições da classe Ck
        Início
(1)      N = conjunto vazio de nós
(2)      A = conjunto vazio de arestas
(3)      grafo G = (N,A)
(4)      N  += qualquer elemento de Ck {passo 1 da seção 3.6}
(5)      cria lista de Arestas Candidatas AC inicialmente vazia AC=nil
(6)      cria lista de Arestas Removidas Temporariamente AR inicialmente vazia AR=nil
(7)      ∀ Ei, Ej tal que M(Ei, Ej) <> 0, insere (Ei, Ej, M(Ei, Ej)) na lista AC
(8)      enquanto (AC <> nil ) faca
(9)      Início
(10)      se (((existe aresta que possa ser conectada a Partição Candidata cp) ou
(11)          (não existe Partição Candidata cp)) ) entao {Negação de Caso –1 }
(12)      Início
(13)       repita  {Escolhe uma aresta que não faça parte de uma partição ou partição candidata - passo 2
                   da seção 3.6}
(14)         escolhe (Ei,Ej, M(Ei,Ej))= maior valor de M(Ei,Ej) existente em AC tal que Ei seja uma
(15)                                    extremidade do grafo e Ej ∉ N
(16)         aij : = aresta entre Ei e Ej
(17)             se (Ej ∈ Partição Candidata) ou ( ∃P ∈ particoesCk tal que Ej ∈ P) entao {Caso 0 }
(18)         início
(19)           AC - = aij
(20)           aij : = nil
(21)         fim
(22)       até (aij <> nil ou AC = nil ) {Fim da escolha da aresta}
(23)       se aij <> nil  então
(24)       início {passo 3 da seção 3.6}
(25)        se a aresta aij forma um ciclo em G então
(26)        início{Casos 2,3,4,5}
(27)          seja ca este ciclo
(28)          se existe outro ciclo entao{Caso 5} {passo 3.2 da seção 3.6}
(29)          início
(30)            AC -= aij
(31)            AR += aij
(32)          fim
(33)          senao
(34)           se ca satisfaz a condição possibilidade dee ciclo entao{Casos 2,3}
(35)           Início {passo 3.1 da seção 3.6}
(36)             marca ca como partição candidata
(37)             AC -= aij
(38)             A+= aij
(39)           fim
(40)           senao{ Caso 4}
(41)             AC -= aij
(42)        Fim{Casos 2,3,4,5}
(43)        Senao {aij não forma um ciclo em G – passo 4 da seção 3.6}
(44)        Início{Casos 1,6,7.1,7.2,8,9}
(45)         se existe uma partição candidata cp então{Casos 6,7.1,7.2,8,9}
(46)           se não há Possibilidade de Extensão de cp por aij então {Casos 7.1,7.2,8,9}
               {passo 4.2 da seção 3.6}
(47)           início
(48)             se aij não está conectado a uma partição candidata então {Caso 7.1}
(49)             início
(50)               AC -= aij
(51)               AR += aij
(52)             fim
```

```
(53)            Senão {passos 4.2.2, 4.2.1 e 4.1 da seção 3.6}
(54)            Início
(55)              corte a_ij
(56)              se existe aresta extensão ae então{Casos 8, 9} {passo 4.2.2 da seção 3.6}
(57)                se cp pode ser estendido por ae entao {Caso 9}
(58)                Início
(59)                  seja E_r o nó de ae que não está em cp
(60)                  cp = cp estendido por ae
(61)                  AC -=a_ij       N += E_j
(62)                  A += a_ri       A += a_ij
(63)                fim
(64)                senao{Caso 8 – cp não pode ser estendido por ae}
(65)                início
(66)                  marque cp como uma partição
(67)                  particoesCk  += cp
(68)                  AC -= a_ij       corte a_pr
(69)                  AC +=  AR    AR  = nil
(70)                 A += a_ij          N += E_j
(71)                fim
(72)              Senao{Caso 7.2} {passo 4.2.1 da seção 3.6 – não existe aresta de extensão}
(73)              Início
(74)                Marca cp como partição
(75)                ParticoesCk += cp
(76)               AC -= a_ij
(77)               AC += AR       AR  = nil
(78)               A += a_ij          N += E_j
(79)              Fim
(80)          Fim { fim dos casos 7.1,7.2,8,9}
(81)          Senao{Caso 6} {passo 4.1 da seção 3.6 – partição candidta pode ser estendida por a_ij}
(82)          Início
(83)            seja E_k o nó de ciclo de cp
(84)            cp = cp estendido por a_ij e a_kj
(85)            AC -= a_ij   AC -= a_kj
(86)            A += a_kj    A += a_ij      N += E_j
(87)          fim
(88)          Senao {Caso 1 – não há partição candidata}
(89)          Início
(90)            AC -= a_ij
(91)             A += a_ij
(92)             N += E_j
(93)          Fim
(94)        Fim
(95)      {Se existe Partição Candidata e não há mais arestas em AC que possam ser conectadas a esta}
(96)       Senao{ Caso –1 }
(97)       Início
(98)         ParticoesCk + = cp
(99)         AC += AR
(100)        AR = nil
(101)      fim
(102)     fim{enquanto}
(103)     se existe uma partição candidata cp então
(104)       particoesCk + = cp
(105)     seja c = todos os elementos de Ck que não pertencem a nenhuma partição de particoesCk
(106)     particoesCk + = c
        Fim.
```

Fig. 3.18 – Construção do Grafo de Afinidades

B.1) Linhas 10-11 e 95-102 da Fig. 3.18: verificação da existência de partição candidata e se não existe aresta em AC que possa ser conectada a mesma

Antes de ser escolhida uma aresta para ser inserida no grafo, devemos verificar se existe partição candidata e se não existe aresta na lista de arestas candidatas (AC) que possa ser conectada a esta partição (linhas 10 e 11 da Fig. 3.18), pois neste caso, como não há mais arestas que possam ser utilizadas para ser conectada à partição candidata (ou seja, só haverá arestas para serem inseridas na ponta que não foi formada a partição), esta não poderá ser mais estendida e se tornará uma partição(linhas 95 a 102 da Fig. 3.18). Este caso está exemplificado e ilustrado na seção 3.2.5.3, em "Criação de Partição", no caso (d). A criação da partição é identificada pela inserção da partição candidata em particoesCk, que guarda todas as partições criadas para Ck (linha 98 da Fig. 3.18) e que será o retorno desta função. Neste momento todas as arestas que estão inseridas em AR devem ser removidas e inseridas em AC (linhas 99 e 100 da Fig. 3.18).

Caso exista aresta a ser conectada ao ciclo identificado como partição candidata (pela qual podemos tentar estender o ciclo) ou não exista partição candidata (pois desta forma podemos conectar arestas em qualquer uma das duas pontas) devemos selecionar a aresta a ser inserida no grafo e fazer uma série de verificações (como formação ou não de ciclos, possibilidade de ciclo e possibilidade de extensão pela aresta), onde as principais idéias foram apresentadas na seção 3.2.6 em "Estrutura Geral" e estas idéias serão detalhadas a seguir.

Considerando que não existe partição candidata ou que exista aresta a ser selecionada para ser conectada a partição candidata existente, o primeiro procedimento a ser realizado é a escolha da aresta a ser inserida no grafo (linhas 13 a 22 da Fig. 3.18). As arestas do grafo são inseridas uma de cada vez selecionando (da matriz da afinidades dos elementos) uma aresta $a_{ij}=(E_i, E_j)$ que satisfaz o seguinte critério: $M(E_i,E_j)$ é o maior valor(ou peso), definido na seção 3.2.4 como $p(a_{ij})$, que não foi selecionado previamente (linhas 14 e 15 da Fig. 3.18) e ei ou ej é uma das extremidades do grafo(onde entende-se por extremidade de um grafo como sendo o nó do grafo que tem apenas uma aresta incidente a ele). Consideremos E_i uma das extremidades do grafo e E_j o novo nó a ser introduzido no grafo e que a aresta $a_{ij}=(E_i, E_j)$ foi selecionada pelo critério supracitado. Se E_j pertence a uma partição ou à partição candidata em questão (linha 17 da Fig. 3.18) esta deve ser descartada, sendo excluída da lista de arestas candidatas (linhas 19 e 20 da Fig. 3.18) e escolhida outra aresta, até que uma aresta satisfaça toda esssas condições ou

que não haja mais arestas em AC (linhas 13 a 22 da Fig. 3.18). Isto se deve ao fato de que:

-Se Ej já pertence a uma partição logo Ej não pode pertencer a outra partição e o intuito é selecionar uma aresta para verificar a possibilidade de extensão da partição candidata pela mesma. Portanto se houvesse a possibilidade de extensão por aij, Ej iria participar de duas partições e nosso intuito é dividir os elementos da classe em partições onde cada elemento só pode estar em uma partição. Além disso, na maior parte dos casos, formaria um ciclo contendo um ou mais nós dessa partição e ou até mesmo contendo uma partição inteira. Por exemplo, na Fig. 3.19 temos um exemplo onde já existem duas partições formadas pelos ciclos C(N_2,N_8,N_3,N_9) e C(N_7, N_1, N_5). Suponha que queremos selecionar uma aresta a ser conectada em N_{10} ou N_5. Se a aresta $a_{7\,10}$ for selecionada, esta aresta deveria ser descartada já que o nó N_7 já faz parte de uma partição. Além disso, podemos notar que esta aresta formaria o ciclo C(N_7, N_8, N_2, N_9, N_3, N_4, N_6, N_{10}) que contém a partição formada pelo ciclo C(N_2,N_8,N_3,N_9) e que as arestas cortadas a_{78} e a_{38} estariam incluídas neste ciclo, porém, arestas cortadas não podem pertencer a nenhum ciclo pela definição existente em 3.2.4.11. Uma aresta que

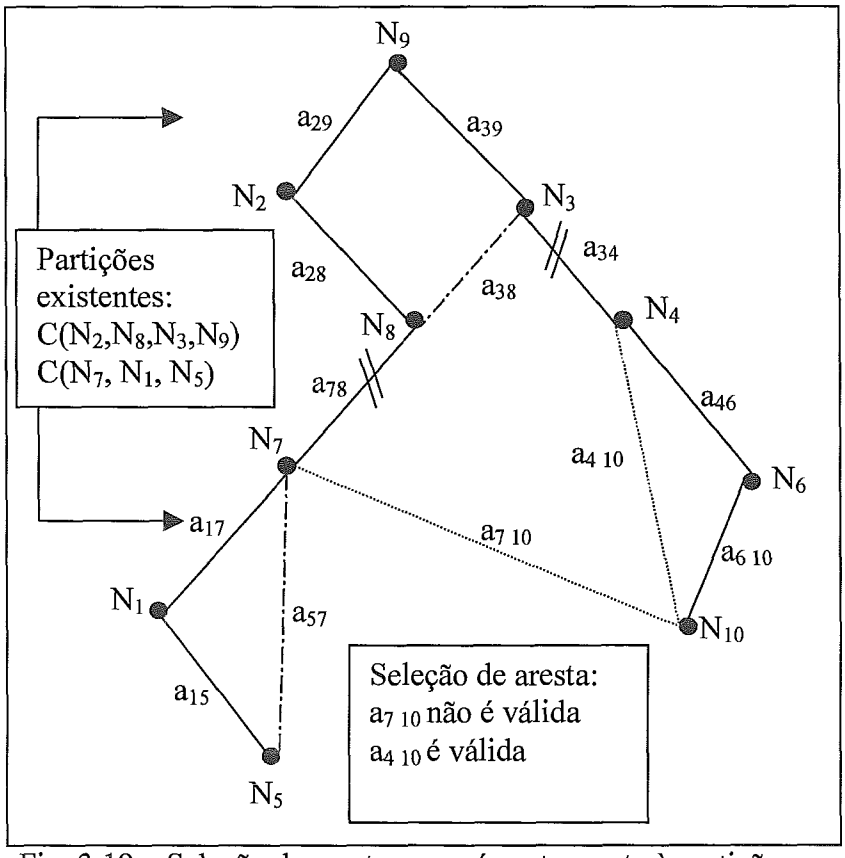


Fig. 3.19 – Seleção de aresta com nó pertencente à partição

satisfaz as condições para ser selecionada (linhas 13 a 22 da Fig. 3.18), por exemplo, é a aresta $a_{4\ 10}$ da Fig. 3.19.

-Se a aresta E_j pertence à partição candidata, a aresta a_{ij} está "dentro"do ciclo, ou seja, E_i e E_j pertecem à partição candidata e portanto forma dois subciclos da partição candidata. Veja o exemplo na Fig. 3.20 onde é escolhida a aresta a_{ij} e temos a partição candidata formada pelo ciclo C(Ep, Eq, E_i, E_j). Selecionando a_{ij} é formado dois ciclos: C(E_p, E_i, E_j) e C(E_q, E_i, E_j). Porém nós não queremos dividir a partição candidata e sim estendê-la, já que se os elementos estão numa partição candidata então há uma grande grau de afinidades entre estes e portanto mantendo-os numa mesma partição diminuiremos o custo total das consultas e chamadas de métodos que utilizam estes elementos. Um exemplo de uma aresta que satisfaz a seleção, indicada pelas linhas 13 a 22 da Fig. 3.18, está apontada na Fig. 20 pela aresta ponto-tracejada a_{qr}.

Selecionada a aresta a_{ij} (o que equivale ao passo 2 da seção 3.2.6), o próximo passo é identificar se ocorre a formação ou não de ciclo (passos 3 e 4 da seção 3.2.6).

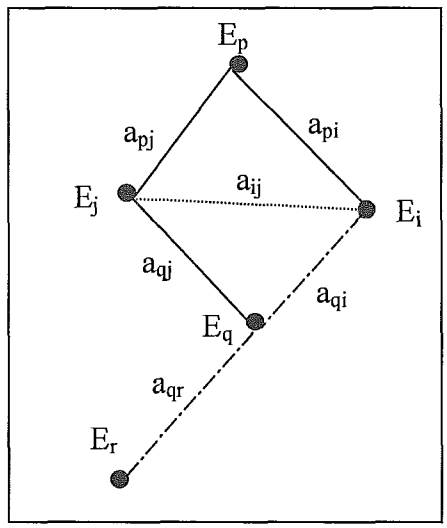


Fig. 3.20 – Seleção de aresta com nó pertencente à partição candidata

B.2) Linhas 25-42 da Fig. 3.18: aresta selecionada forma ciclo em G (passo 3 da seção 3.6)

B.2.1) Linhas: 25-32 da Fig. 3.18: formação de ciclo com outro ciclo já existente (passo 3.2 da seção 3.6)

Agora explicaremos mais detalhadamente o passo 3 da seção 3.2.6 – "Estrutura Geral": a aresta selecionada forma ciclo no grafo. Se a inclusão da aresta a_{ij}=(E_i, E_j) formar um ciclo (C1) no grafo (linha 25 da Fig. 3.18), testaremos se já existe no grafo

outro ciclo (C2). Se existir C2, a aresta a_{ij} é removida da lista AC e incluída em AR (linhas 29 e 32 da Fig. 3.18). Isto se deve ao fato que se já existe um ciclo no grafo então devemos tentar estender este ciclo antes de formar outro. Mas após a formação da partição relativa ao ciclo C2 em questão, esta aresta escolhida deve retornar a AC a fim de que seja formado o ciclo C1 novamente. Através da Fig. 3.21 podemos exemplificar esta situação. Considere que exista a partição candidata identificada pelo ciclo C2(E_2,E_8,E_3,E_9). Selecionando a aresta a_{57}, é formado o ciclo C1(E_7, E_1, E_5) e segundo (NAVATHE e RA, 1989) esta aresta é descartada (já que não podemos estender C2 por a_{57}). Porém, após a formação da partição relacionada a C2 (quando não for mais possível estender C2), a aresta a_{57} tem que retornar para AC pois senão o ciclo C1(E_7, E_1, E_5) não será mais formado. Este foi um dos motivos onde foi verificada a importância de criar a lista AR.

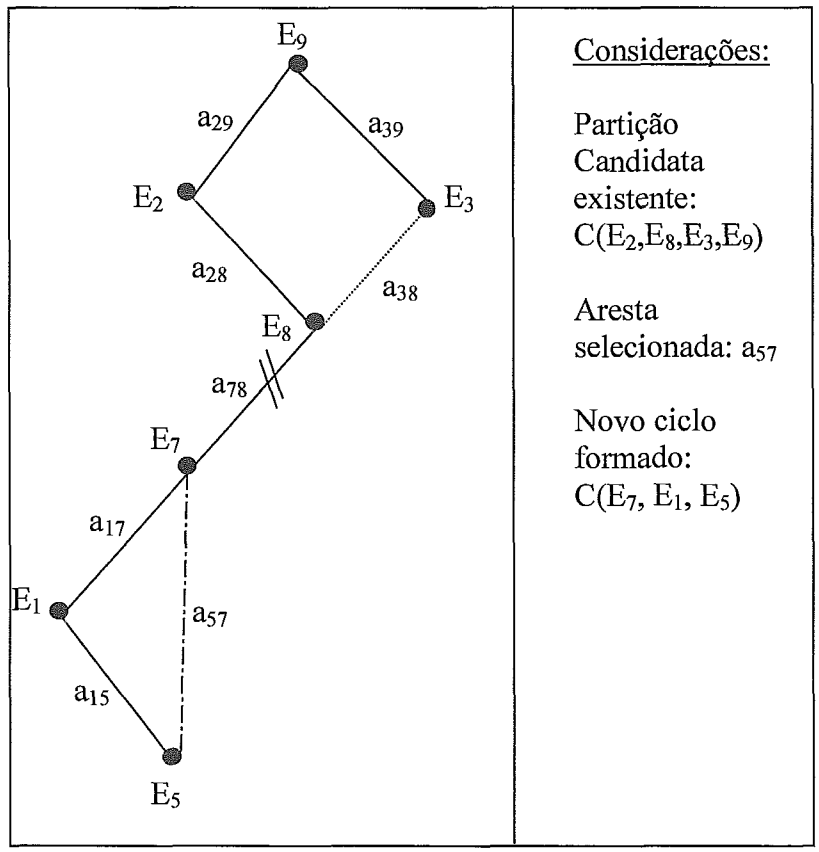


Fig. 3.21 – Aresta selecionada forma ciclo e já existe outro

B.2.2) Linhas: 33-42 da Fig. 3.18: formação de ciclo de afinidades (passo 3.1 da seção 3.6)

Se a inclusão da aresta $a_{ij}=(E_i, E_j)$ formar um ciclo no grafo (linha 25 da Fig. 3.18), testaremos se já existe no grafo outro ciclo (linha 28 da Fig. 3.18). Se não existir, então testaremos se o ciclo é um ciclo de afinidades (linha 34 da Fig. 3.18). Na seção 3.2.5.1 é encontrada toda a discussão detalhada sobre formação de ciclo de afinidades, incluindo a condição "Possibilidade de Ciclo". Podemos dizer, de acordo com (NAVATHE E RA, 1989), que um ciclo é um "ciclo de afinidades"(linha 34 da Fig. 3.18) se for satisfeita a Possibilidade de Ciclo, ou seja,

se não existe Aresta de Extensão

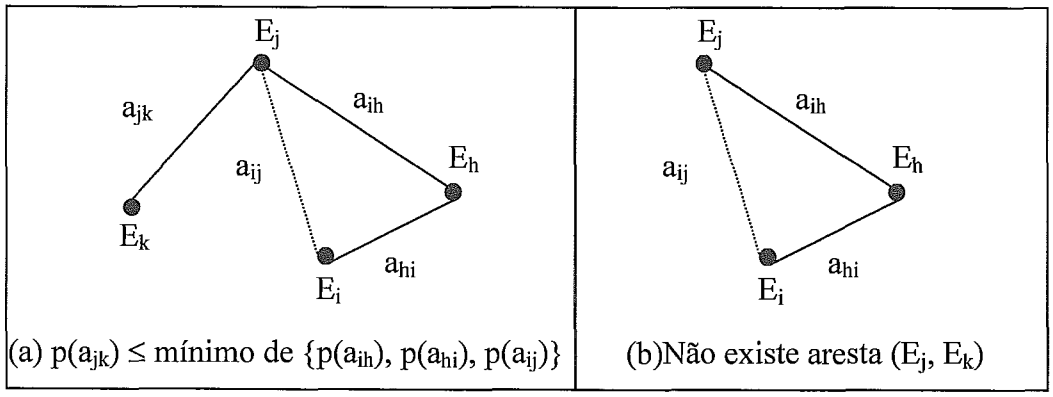
ou

p(Aresta de Extensão) ≤ p(todas as arestas do ciclo formado).

Intuitivamente, não existe aresta de extensão que deveria participar do ciclo por ter os elementos - que a compõe - uma grande afinidade em relação aos outros elementos do ciclo. No exemplo da Fig. 3.22 - (a), o ciclo formado pelas arestas a_{ij}, a_{ih} e a_{hi} será um ciclo de afinidades (formalizando as idéias acima mencionadas) se a aresta de extensão a_{jk} satisfizer a Possibilidade de Ciclo, ou seja,

$p(a_{jk}) \leq$ mínimo de $\{p(a_{ij}), p(a_{ih}), p(a_{hi})\}$

A aresta em tracejado representa a aresta selecionada $a_{ij}=(Ei, Ej)$ que formou o ciclo. No exemplo da Fig. 3.22 - (b) o ciclo formado pelas arestas a_{ij}, a_{ih} e a_{hi} também será um ciclo de afinidades já que não existe uma aresta $a_{jk}=(E_j, E_k)$ que seja Aresta de Extensão.



(a) $p(a_{jk}) \leq$ mínimo de $\{p(a_{ih}), p(a_{hi}), p(a_{ij})\}$ (b)Não existe aresta (E_j, E_k)

Fig. 3.22 – Ciclo de afinidades formado pelo ciclo $C(E_i, E_j, E_h)$

O ciclo de afinidade é então considerado como Partição Candidata (linha 36 da Fig. 3.18) e a aresta a_{ij} deve ser removida de AC (linha 37 da Fig. 3.18). Além disso, a aresta a_{ij} deve ser inserida no grafo (linha 38 da Fig. 3.18). Note que o nó E_j não precisa ser

inserido no grafo, pois como foi formado um ciclo, então E_i e E_j já faziam parte do grafo.

Contrariamente, se a inclusão da uma aresta $a_{ij}=(E_i, E_j)$ formar um ciclo no grafo porém este não satisfizer a Possibilidade de Ciclo, o ciclo não será marcado, bastando apenas retirar de AC a aresta a_{ij} (linhas 40 e 41 da Fig. 3.18).

B.3) Linhas 43-94 da Fig. 3.18: aresta selecionada não forma ciclo em G

Agora explicaremos mais detalhadamente o passo 4 da seção 3.2.6 – "Estrutura Geral":

Se a inclusão da aresta $a_{ij}=(E_i, E_j)$ não formar um ciclo no grafo (linhas 43 a 94 da Fig. 3.18), poderão ser feitos vários procedimentos diferentes baseados nas seguintes informações: existência de partição candidata, extensão por a_{ij} e extensão pela Aresta Extensão(condições nas linhas 45, 46 e 56 da Fig. 3.18).

B.3.1) Linhas 88-93: não há partição candidata

Verifique a existência de partição candidata (condição existente na linha 45 da Fig. 3.18). Se não houver (linhas 88 a 93 da Fig. 3.18), inclua a aresta a_{ij} e o nó E_j no grafo (ver exemplo na Fig. 3.23) e remova a_{ij} de AC.

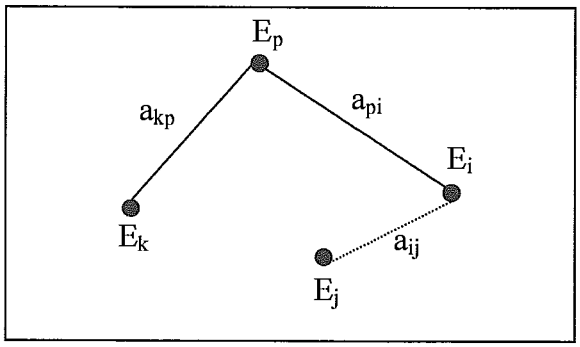


Fig. 3.23 – A aresta a_{ij} é escolhida e não há partição candidata

B.3.2) Linhas 48-52: não pode estender por aij pois não está conectada à partição (caso 4.3.2 da seção 3.6)

Se há partição candidata e esta não possa ser estendida por a_{ij} (condição existente na linha 46 da Fig. 3.18) pois a_{ij} não foi conectada à partição candidata (linha 48 da Fig. 3.18), simplesmente devemos retirar a_{ij} de AC e inserir em AR (linhas 49 a 52 da Fig. 3.18). Veja um exemplo na Fig. 3.24.

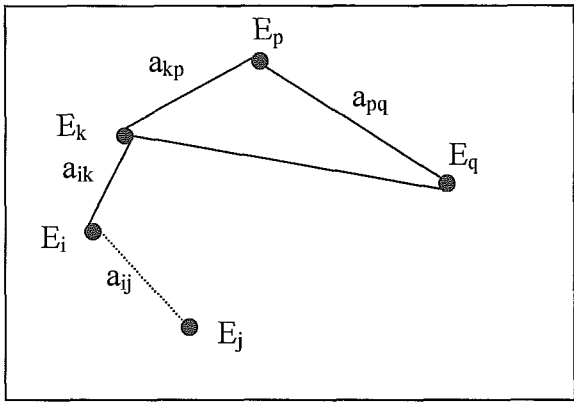


Fig. 3.24 – Não há extensão pela aresta escolhida a_{ij} pois esta não está conectda à partição candidata

B.3.3) Linhas 81 – 87: partição candidata pode ser estendida por a_{ij} (passo 4.1 da seção 3.6)

Se houver partição candidata , verifique se esta pode ser estendida, de acordo com a Fig. 3.7 que define Possibilidade de Extensão, por a_{ij} (condição existente na linha 46 da Fig. 3.18). Se puder ser estendido por a_{ij}, considerando E_k o Nó de Ciclo da partição candidata (ilustrada no exemplo da Fig. 3.25), a nova partição candidata conterá também o nó E_j e as arestas a_{ij} e a_{kj} (linhas 81 a 87 da Fig. 3.18).

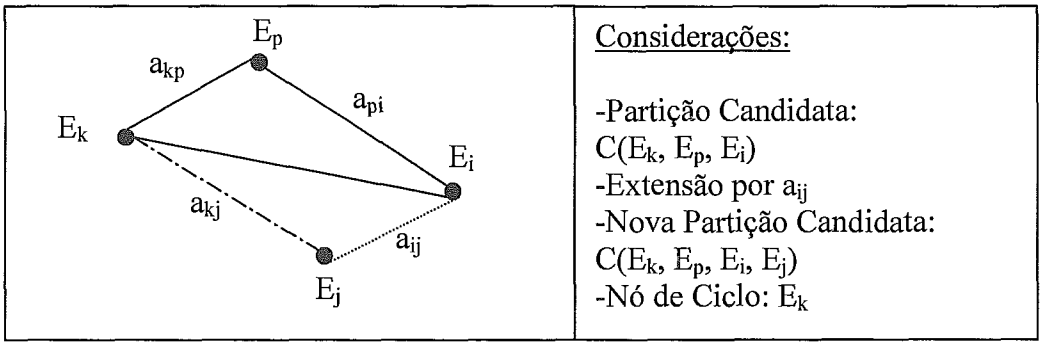
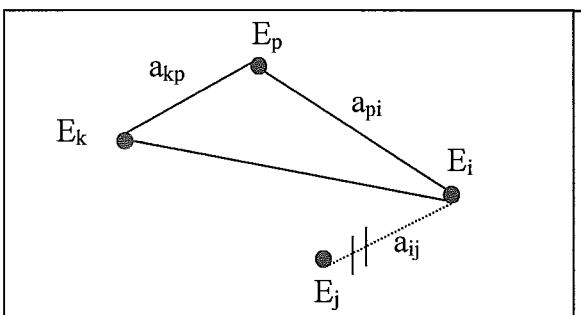


Fig. 3.25 – Extensão da partição candidata pela aresta escolhida a_{ij}

Os mesmos deverão ser incluídos no grafo (incluindo a aresta a_{kj} que foi necessária para a extensão) e as arestas a_{ij} e a_{kj} deverão ser retiradas de AC, já que foram incluídas no grafo (linhas 85 e 86 da Fig. 3.18).

B.3.4) Linhas 72-79: não pode estender por aij (conectada à partição) e não existe Aresta de Extensão (passo 4.2.1 da seção 3.6)

Se houver partição candidata, aij está conectada à partição candidata e esta não possa ser estendida por a_{ij} (condições existentes nas linhas 45, 46 e 48 da Fig. 3.18) pois não satisfaz a possibilidade de extensão, marque aij como aresta cortada(linha 55 da Fig. 3.18) e verifique se existe Aresta de Extensão (condição existente na linha 56 da Fig. 3.18). Se não existir Aresta de Extensão(ver exemplo na Fig. 3.26), então a partição candidata deverá ser marcada como partição (linhas 74 e 75 da Fig. 3.18) e aij deverá ser retirada de AC (já que esta foi inserida no grafo, mesmo com um corte) e inserida no grafo juntamente com Ej (linhas 76 e 78 da Fig. 3.18 respectivamente). Além disso, como foi criada uma partição, todas as arestas temporariamente removidas deverão ser inseridas novamente em AC e excluídas de AR (linha 77 da Fig. 3.18).



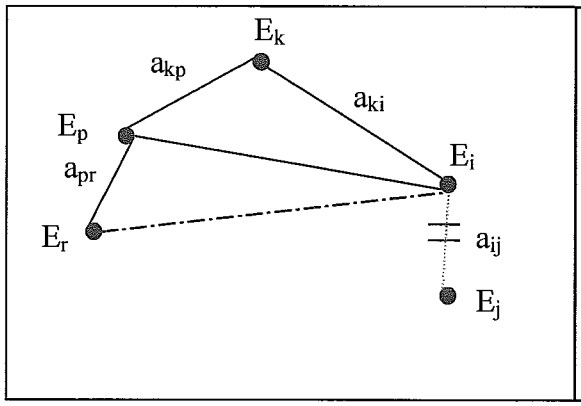
Considerações:

-Partição Candidata: $C(E_k, E_p, E_i)$
-Não há Possibilidade de Extensão por a_{ij}
-Corte em a_{ij}
-Não há Aresta de Extensão

Fig. 3.26 – Não há possibilidade de extensão da partição candidata pela aresta escolhida a_{ij} e não há Aresta de Extensão

B.3.5) Linhas 54-71: não é possivel estender por aij e é verificada a possibilidade de extensão pela Aresta de Extensão (passo 4.2.2 da seção 3.6)

Se houver partição candidata, a_{ij} está conectada à partição candidata e esta não possa ser estendida por a_{ij} (condição existente na linha 46 da Fig. 3.18) pois não satisfaz a possibilidade de extensão, marque a_{ij} como aresta cortada(linha 55 da Fig. 3.18) e verifique se existe Aresta de Extensão (condição existente na linha 56 da Fig. 3.18). Se existir Aresta de Extensão, deve ser verificada a Possibilidade de Extensão pela Aresta de Extensão. Suponhamos agora que exista.



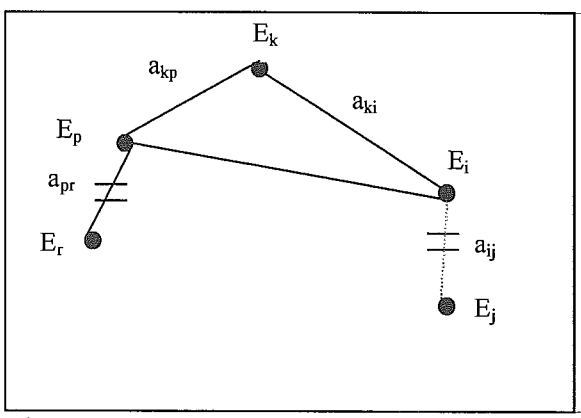
Considerações:

-Partição Candidata: $C(E_k, E_p, E_i)$
-Não há Possibilidade de Extensão por a_{ij}
-Aresta de Extensão: a_{pr}
-Nova partição candidata: $C(E_r, E_k, E_p, E_i)$

Fig. 3.27- Extensão pela Aresta de Extensão e corte em a_{ij}

Se a partição candidata puder ser estendida pela Aresta de Extensão a_{rp} (linhas 58-63 da Fig. 3.18), tal que E_p está na partição candidata e E_r não está (ver exemplo na Fig. 3.27), então a partição candidata deverá ser estendida por a_{rp} (linhas 59 e 60 da Fig. 3.18). As arestas a_{ij} e a_{ir} deverão ser retiradas de AC (já que a_{ij} foi inserida no grafo, mesmo com um corte e a_{ir} foi utilizada na extensão e incluída também no grafo) e inserida no grafo juntamente com E_j (linhas 61 e 62 da Fig. 3.18).

Caso contrário, se a partição não puder ser estendida pela Aresta de Extensão a_{rp} (linhas 64 a 71 da Fig. 3.18), corte a_{rp} e desta forma é isolada a partição candidata, tornando-se uma partição (linhas 66 e 67 da Fig. 3.18) – veja o exemplo da Fig. 3.28. As arestas a_{ij} e a_{pr} deverão ser retiradas de AC (já que ambas foram inseridas no grafo, mesmo com um corte em cada) e deverão ser inseridas no grafo juntamente com o nó E_j.



Considerações:

-Partição Candidata: $C(E_k, E_p, E_i)$
-Aresta de Extensão: a_{pr}
-Não há Possibilidade de Extensão por a_{ij} nem por a_{pr}

Fig. 3.28 - Não há Possibilidade de Extensão por a_{ij} nem pela Aresta de Extensão

C) Linhas 103-106 da Fig. 3.18: criação partições após a inclusão da última aresta

Finalmente, após a inclusão da última aresta, ou seja, quando AC estiver vazia, se existir uma partição candidata esta se tornará uma partição já que não há mais arestas para tentar estender esta partição candidata. Além disso, haverá a formação de uma partição com todos os elementos que não participam de nenhuma partição (maiores detalhes na seção 3.2.5.3 – (c).

3.8 EXEMPLO

A partir da Matriz de Afinidade entre os Elementos apontada na tabela 3.2 da seção 3.2.4.2 (utilizada por NAVATHE e RA (1989), NAVATHE *et al.* (1984)) exemplificam a construção do grafo de afinidades e obtenção das partições (Fig. 3.29), inclusive mostram também que começando pelo nó 1 ou 9 o grafo resultante é o mesmo. Apresentam também um outro exemplo com 15 transações e 20 elementos.uponha que comece pelo nó 9 (passo 1 da seção 3.6) e então seja selecionada as arestas a_{39}, a_{29} e a_{28} (passo 2 da seção 3.6).A próxima aresta a ser escolhida será a_{89} e não formará o ciclo pois não satisfaz a possibilidade de ciclo (não satisfaz a Fig. 3.5 no passo 3.1 da seção 3.6) e esta será retirada da lista de arestas candidatas.Depois a aresta a_{38} é selecionada formando um ciclo primitivo (satisfazendo o passo 3.1 da seção 3.6). O nó N_3 será marcado como Nó de Ciclo. O processo continua selecionando a aresta a_{78} e, como há a partição candidata formada pelo ciclo C(N_8, N_2, N_9, N_3) a possibilidade de extensão (Fig. 3.7) é verificada. Como não há possibilidade de extensão pela nova aresta a_{78} (aresta a ser considerada na Fig. 3.7) nem pela a_{73} (aresta formadora de ciclo na Fig. 3.7) e não há aresta de extensão no momento, é satisfeito o passo 4.2.1, o ciclo C(N_8, N_2, N_9, N_3) é marcado como partição e a aresta a_{78} é cortada. Depois são inseridas as arestas a_{57}, a_{51} e a_{17} (passo 2 da seção 3.6), formando outro ciclo que será marcado como partição candidata (passo 3.1 da seção 3.6). Como não existem mais arestas (não zeradas) que possam ser conectadas na partição candidata (pelo nó N_1) então (caso B.1 da seção 3.7.4) este ciclo será marcado com partição. Seguindo, as arestas a_{34}, a_{46}, $a_{6\,10}$ e $a_{4\,10}$ serão inseridas (passo 2 da seção 3.6) e será formado um ciclo (passo 3.1 da seção 3.6) que será marcado como partição candidata. Somente as arestas $a_{3\,10}$ e $a_{9\,10}$ ainda poderiam ser conectadas ao ciclo C(N_4, N_6, N_{10}) porém, como os nós N_3 e N_9 já fazem parte de outro ciclo, estas arestas são removidas da lista de arestas candidatas. Desta forma, não haverá mais arestas que possam ser conectadas ao ciclo e este será marcado

como partição.

Vale ressaltar que como a escolha das arestas é feita como a maior aresta a ser conectada à árvore, se houver várias arestas com o mesmo valor uma dessas serão escolhidas aleatoriamente. Desta forma haverá casos em que, se pararmos o algoritmo quando não houverem mais nós a serem conectados pode ainda faltar inserir arestas para achar os ciclos (Fig. 3.14).

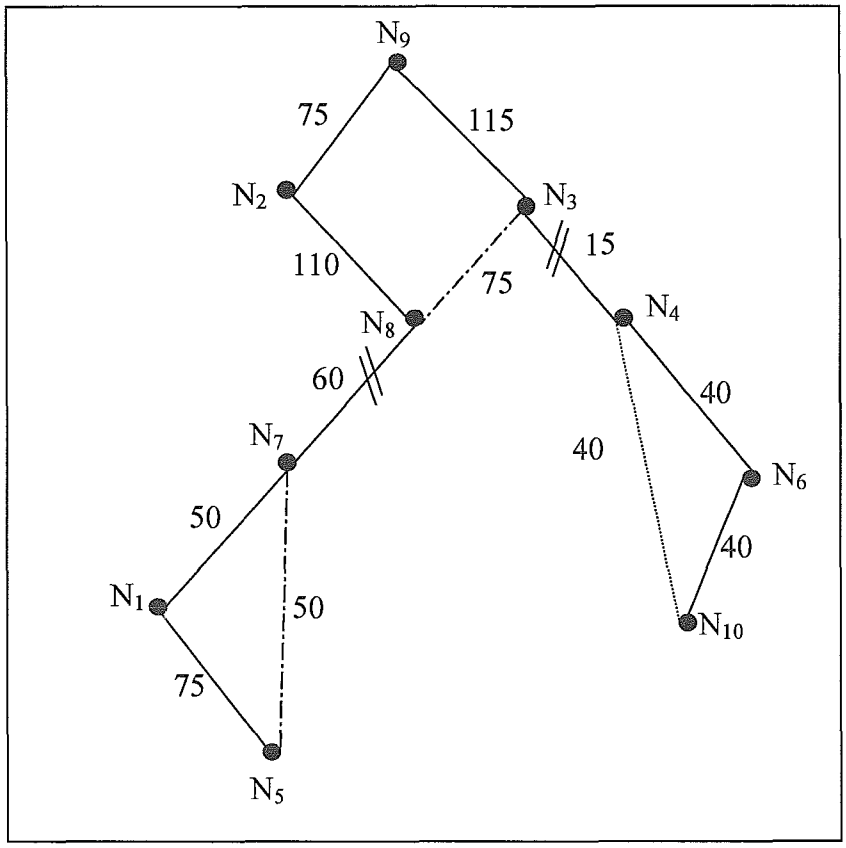


Fig. 3.29 – Resultado começando pelo nó 9 - Fonte: (NAVATHE E RA, 1989)

3.9 Considerações Gerais Sobre o Algoritmo

Um problema que se encontra em aberto é a geração de partições com menos de três elementos já que para formar um ciclo no grafo é preciso pelo menos três nós. Dependendo da aplicação, para diminuir o custo total do processamento das consultas, pode ser necessário criar partições com um ou dois nós. Este tipo de partição só ocorreria, pelo algoritmo de Fragmentação Vertical FVNB apresentado neste capítulo, quando no fim do algoritmo é formada uma partição que contém nós que não foram utilizados em outras partições (nós que têm baixa afinidade com os demais), apresentado no pseudo-código do algoritmo nas linhas 105 e 106 da Fig. 3.18.

Uma vez que são as afinidades de valor mais alto as que mais influenciam, como as maiores afinidades entre os elementos que influenciam mais no custo total de processamento da consulta (e esta representam as partições formadas), uma sugestão é fazer, após a criação das partições, uma avaliação extra de cada partição e eventualmente particioná-las novamente, podendo assim encontrar partições com um ou dois elementos.

A Avaliação Extra das Partições de uma classe Ck que foi verticalmente fragmentada é feita da seguinte maneira:

- Para cada partição P da classe Ck faça:

- Considere o valor de afinidade médio (Vm) entre os elemento de uma partição P como sendo a média aritmética dos valores de afinidades das arestas que participam de P.
- Sejam P1 e P2 partições inicialmente vazias
- Crie uma lista (L) com as arestas de P ordenada decrescentemente em relação ao peso da aresta
- Considere a_{ij} a aresta de maior peso em L (identificado por $p(a_{ij})=aff_{ij}$=MaiorAresta) e a_{pq} a de menor peso em L (identificado por $p(a_{pq})=aff_{pq}$=MenorAresta), então
- Se (MaiorAresta - MenorAresta) $\geq$ Vm então o espalhamento dos valores dos pesos das arestas é grande.
- Se (MaiorAresta + MenorAresta) $\leq 2 \times$ Vm então P2 receberá os nós da aresta identificada pela MenorAresta e P1 receberá todos os nós pertencentes a todas as arestas que participam da lista L Caso contrário, P1 receberá os nós da aresta identificada pela MaiorAresta e P2 receberá todos os nós pertencentes a todas as arestas que participam da lista L
- Para finalizar, deverá ser excluído de P2 os nós que estão também em P1 (já que em P1 se encontram os nós de maior afinidade).

A Avaliação Extra de Partições pode ainda ser melhorada e, inclusive, pode ser utilizado o Módulo de Revisão de Teoria (capítulo 4) para revisar a avaliação apresentada, utilizando as fragmentações obtidas pelo Módulo de Busca Exaustiva para

gerar exemplos positivos (fragmentações com custo baixo) e negativos (com custo alto) para o FORTE.

3.9.1 AVALIAÇÃO EXPERIMENTAL DO ALGORITMO

Através de um estudo usando o benchmark OO7 (CAREY *et al.*, 1993) podemos mostrar que o custo total da aplicação, de acordo com o modelo de custo descrito em (RUBERG, 2001), diminuirá se fizermos a Avaliação Extra da Partição supramencionada. O estudo engloba a fragmentação vertical da classe *AtomicPart*, utilizando as consultas Q1, Q2 e Q3 e as travessias (*traversal*) T1 e T2 descritas em (BAIÃO, 2001, CAREY *et al.*, 1993).

Uma versão resumida da Matriz de Uso dos Elementos (sem a coluna Tipo e sem os elementos que não são acessados por nenhuma das consultas e travessias) entre os elementos E_1, E_2 e E_3, que representam respectivamente os elementos *Id*, *to* e *buildDate*, da classe *AtomicPart*, pode ser visualizada na Fig 3.30. Na Fig. 3.31 encontramos a Matriz de Afinidades entre os Elementos de *AtomicPart* e na Fig. 3.32 encontramos o Grafo de Afinidades, lembrando que será criada uma partição extra com os elementos que não foram utilizados por nenhuma consulta nem travessia(ou seja, conterá os elementos *type*, *x*, *y*, *isRootPart* e *isPartOf*).

Transações x Elementos				Acc
	E_1	E_2	E_3	
Q1	1	0	1	100
Q2	1	0	1	50
Q3	0	0	0	10
T1	1	1	1	30
T2	0	0	0	30

Fig. 3.30 – Matriz de Uso de Elementos de *AtomicPart*

	E_1	E_2	E_3
E_1	180	30	180
E_2	30	30	30
E_3	180	30	180

Fig. 3.31 – Matriz de Afinidades entre os Elementos de *AtomicPart*

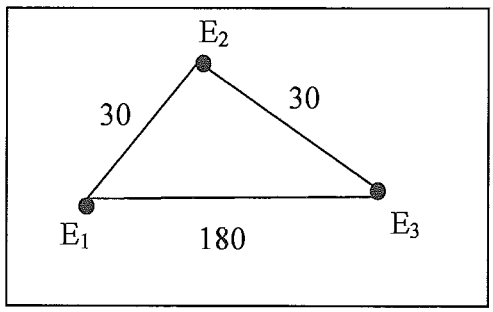


Fig. 3.32 – Grafo de Afinidades de *AtomicPart*

Temos portanto os seguintes resultados:

-Serão criadas duas partições (P1 e P2)

-P1 será a projeção dos atributos que não são utilizados nas consultas e travessias, ou seja, a projeção de *type*, *x*, *y*, *isRootPart* e *isPartOf* de *AtomicPart*

-P2 será a projeção dos atributos *Id*, *to* e *BuildDate* de *AtomicPart*

O custo total (RUBERG, 2001) para o processamento com as fragmentações acima mencionadas é de 782.727,225 e constatamos que ainda teríamos 9,817% de todas as fragmentações possíveis para AtomicPart com custo menor, mostrando que ao mesmo tempo que o algoritmo é eficaz ainda temos como melhorá-lo. Este custo chamamos de custo heurístico.

Para esta avaliação foi utilizado o Módulo Busca Exaustiva (Fig.1.1 do capítulo 1), composto de três módulos: Geração dos Fragmentos, Avaliação dos Fragmentos e Função de Custo. Através do Módulo Função de Custo, foi calculado o custo heurístico. O Módulo Geração dos Fragmentos gera, utilizando busca exaustiva, todas as possíveis fragmentações da classe a ser verticalmente fragmentada, que no caso foi *AtomicPart*. Foi então calculado o custo de todas essas fragmentações pelo Módulo Avaliação dos Fragmentos utilizando o Módulo Função de Custo.

Utilizando a Avaliação Extra das Partições sugerida na seção anterior, a partição P2 seria dividida em P2 e P3, onde:

-P2 será a projeção dos atributos *Id* e *BuilDate* de *AtomicPart*

-P3 será a projeção do atributo *to*

O custo total (RUBERG, 2001) para o processamento com estas novas fragmentações baixaria para 752.757.225 e portanto, teríamos apenas 1,253% de todas as fragmentações possíveis de AtomicPart abaixo deste valor. No entanto, observando tais fragmentações (com valor de custo total menores que 752.757.225) pode ser verficado que em todas são considerados fragmentos distintos para *Id, BuilDate* e *to* de *AtomicPart* , ou seja, é criado um fragmento só com *Id,* outro somente com *BuilDate*, outro só com *to* de *AtomicPart* e outros fragmentos com a combinação dos atributos restantes (*type*, *x*, *y*, *isRootPart* e *isPartOf).* Este custo menor se deve ao fato de que a funçao de custo utilizada para a avaliação de RUBERG (2001) é simplificada e não considera a sobrecarga (*overhead*) de gerência do SGBD sobre um grande número de fragmentos o que torna essas alternativas de fragmentação desvantajosas.

Capítulo 4

UMA ABORDAGEM BASEADA EM CONHECIMENTO PARA REFINAMENTO DO ALGORITMO DE FRAGMENTAÇÃO VERTICAL

Este capítulo apresenta uma estratégia baseada em conhecimento para refinamento do Algoritmo de Fragmentação Vertical (VFNBC) apresentado no capítulo anterior.

4.1 MOTIVAÇÃO

O algoritmo de fragmentação vertical baseado em heurísticas apresentado no capítulo 3 produz resultados com bom desempenho como mostrados em (BAIÃO, 2001). Entretanto, seria extremamente interessante continuar a melhorar estes resultados, descobrindo novas heurísticas para o problema de fragmentação vertical, e incorporá-las ao algoritmo. Contudo, isto requereria uma análise detalhada do desempenho de cada resultado experimental novo na literatura, e as modificações no algoritmo deveriam ser feitas manualmente. Adicionalmente, a formalização de novas heurísticas e sua incorporação ao algoritmo já existente, de tal forma a manter a consistência das heurísticas definidas previamente, mostrou-se um problema complexo, e de dificuldade crescente.

Por conseguinte, esta seção propõe uma abordagem baseada em conhecimento para melhorar o VFNBC através da técnica de Aprendizado de Máquina Relacional chamada Revisão de Teoria (LAVRAC e DZREROSKI, 1994, RICHARDS e MOONEY, 1995, WROBEL, 1996). Foram estendidas as idéias propostas em (BAIÃO, 2001), onde é mostrada a eficácia desta abordagem baseada em conhecimento para melhorar uma versão anterior de um algoritmo existente de análise através de um experimento utilizando o benchmark 007 (CAREY *et al.*, 1993). Em BAIÃO (2001), o processo de Revisão de Teoria modificou automaticamente a versão anterior do algoritmo da análise e produziu uma nova versão, onde foi obtido um esquema da fragmentação com desempenho melhor.

O objetivo é incorporar automaticamente no VFNBC, apresentado no capítulo anterior, as mudanças requeridas de forma a obter os melhores esquemas de fragmentação vertical que possam ser encontrados de acordo com resultados experimentais, e consequentemente refletir as novas heurísticas implícitas em tais resultados. Este processo da revisão representará então um refinamento de nosso conjunto inicial de regras, descobrindo assim um novo conjunto de regras que representarão o algoritmo revisado.

Alguns pesquisadores têm trabalhado em aplicar técnicas da aprendizagem de máquina para resolver problemas no contexto de Banco de Dados (apresentados na

seção 2.3). Todavia, considerar a fragmentação vertical da classe como uma aplicação para a Revisão de Teoria é uma abordagem nova na área.

Na próxima seção é explicado o motivo da escolha da aplicação de Revisão de Teoria para refinar o algoritmo VFNBC.

4.2 Redes Neurais X Programação de Lógica Indutiva

Foi considerada primeiramente por BAIÃO (2001) a idéia de usar redes neurais baseadas em conhecimento para revisar a base de conhecimento. Há, na literatura, muitas abordagens para usar redes neurais no processo de Revisão de Teoria usando regras proposicionais, tais como KBANN (TOWELL e SHAVLIK, 1994) e CIL2P (GARCEZ e ZAVERUCHA, 1999). Entretanto, devido à existência de símbolos funcionais ("function symbols") no algoritmo de análise (tal como listas), BAIÃO (2001) decidiu que o mesmo não poderia ser expressado através de regras proposicionais. Como o algoritmo VFNBC também utiliza símbolos de função é necessária uma linguagem mais expressiva, tal como cláusulas de Horn de primeira ordem. Já que o refinamento de teoria de primeira ordem em redes neurais ainda é um problema de pesquisa em aberto (BASILIO *et al.*, 2001), foi decidido trabalhar com uma técnica da aprendizagem de máquina relacional - Programação em Lógica Indutiva (ILP) (LAVRAC e DZREROSKI, 1994, MUGGLETON e DE RAEDT, 1994).

De acordo com MITCHELL (1997), o processo de ILP pode ser visto como automaticamente deduzir programas em Prolog através de exemplos e, possivelmente, da base de conhecimento inicial. Em (MITCHEL, 1997), indicou-se que os algoritmos de aprendizagem de máquina que usam base de conhecimento, que combinam mecanismos indutivos com analíticos, obtem os benefícios de ambas as abordagens: melhor acurácia na generalização, um número menor de exemplos de treinamento são requeridos e capacidade de explicação dos resultados obtidos.

Pelo fato de que cláusulas de primeira ordem de Horn possam ser interpretadas como programas na linguagem lógica de programação Prolog, o processo de Revisão de Teoria para o aprendizado pode ser chamado de Programação em Lógica Indutiva.

4.3 A TÉCNICA DE REVISÃO DE TEORIA

A abordagem baseada em conhecimento que será usada, proposta por BAIÃO (2001) para o refinamento do PDBDOO, implementa uma técnica de inteligência artificial denominada revisão de teorias (WROBEL, 1996). O processo de revisão de teorias é responsável por alterar automaticamente o algoritmo original (denominado teoria inicial, ou conhecimento preliminar) de forma a adequá-lo para a geração do melhor esquema de fragmentação vertical encontrado pela busca. O resultado do procedimento de refinamento é o algoritmo alterado (denominado teoria revisada) capaz de obter a solução ótima que lhe foi apresentada como entrada.

A técnica de revisão de teorias tem o objetivo de encontrar uma modificação mínima em uma teoria inicial (ou conhecimento preliminar) de forma a classificar corretamente um conjunto de exemplos de treinamento. A descrição formal do processo é apresentada na Fig. 4.1:

Dados:	um conceito-alvo C
	um conjunto P com exemplos positivos de C
	um conjunto N com exemplos negativos de C
	uma linguagem de hipóteses L
	uma teoria inicial T expressa em L descrevendo C
Encontre:	uma teoria revisada TR expressa em L que corresponda à modificação mínima de T tal que TR é correta nos exemplos de P e de N

Fig. 4.1: Especificação formal do processo de revisão de teorias

Fonte: (BAIÃO, 2001)

Uma teoria é um conjunto de cláusulas de programa definidas. Uma cláusula de programa definida é uma cláusula da forma:

$$\alpha \leftarrow \beta 1 \ldots \beta n \qquad (1)$$

onde α, $\beta 1 \ldots \beta n$ são fórmulas atômicas (CASANOVA et al, 1987, LLOYD, 1987).

Um conceito-alvo é um predicado em uma teoria para o qual existam exemplos no conjunto de treinamento.

Um exemplo é uma instanciação de um conceito-alvo. Por exemplo, um exemplo do conceito "node", que representa um nó do grafo, é:

$$\text{node}(1) \qquad (2).$$

Cada exemplo i possui um conjunto de fatos associados F_i, o qual reúne todos os exemplos de um conceito em um conjunto de treinamento. Um exemplo positivo deve ser derivável da teoria e de seus fatos associados, enquanto que exemplos negativos não.

Para representar, por exemplo, o ciclo mostrado na Fig. 4.2, no domínio do VFNBC, será necessário o conjunto de fatos indicados na Tabela 4.1. A modelagem necessária para a revisão de teoria utilizando o sistema FORTE será apresentada na seção 4.4.3.

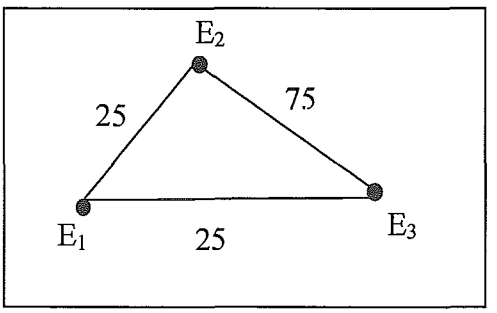


Fig. 4.2 – Exemplo de Ciclo

Tabela 4.1 – Conjunto de fatos para representar um ciclo

Definição dos identificadores dos nós (*node*) e das arestas (*edge*), peso da aresta (*weight*) e nós acessados por cada aresta (*edgeAccess*)			
node(1).	edge(edge12).	weight(edge12,25).	edgeAccess(edge12,1,2).
node(2).	edge(edge13).	weight(edge13,25).	edgeAccess(edge13,1,3).
node(3).	edge(edge23).	weight(edge23,75).	edgeAccess(edge23,2,3).
Identificação das arestas cortadas (*cut*) (0 – não há corte; 1- há corte)			
cut(edge12,0). cut(edge13,0). cut(edge23,0).			
Definição do identificador de ciclo (*cycle*) e nós pertencentes ao ciclo (*nodesAccessByCycle*)			
cycle(c1). nodesAccessByCycle(c1,[1,2,3]).			

A corretude de uma teoria é definida da seguinte maneira: dados um conjunto P de exemplos positivos e um conjunto N de exemplos negativos, uma teoria T é correta com relação ao conjunto de exemplos se e somente se:

$$\forall p \in P: T \cup Fp \models p \qquad (3)$$

$$\forall n \in N: T \cup Fn \not\models n \qquad (4)$$

onde em (3) cada exemplo positivo p em P possuiu um conjunto de fatos associados indicados por Fp e este exemplo p é derivável da teoria T e seus fatos associados Fp. Já em (4), para cada exemplo negativo n em N há um conjunto de fatos associados Fn e este exemplo n não é derivável da teoria T e seus fatos associados Fn.

O processo de revisão de teorias aplica um conjunto de modificações na teoria inicial com o objetivo de obter uma teoria revisada correta. As modificações realizadas em uma teoria resultam da aplicação de operadores de revisão (de especialização e generalização), os quais fazem pequenas modificações sintáticas na teoria. Uma teoria revisada correta obtida através de uma modificação mínima da teoria inicial é alcançada minimizando o número de operadores aplicados. Por requerer um conjunto mínimo de modificações, o processo assume implicitamente que a teoria inicial encontra-se aproximadamente correta e, portanto, a teoria revisada deve ser semanticamente e sintaticamente tão parecida quanto possível com a teoria inicial.

Trabalhos anteriores (BRUNK e PAZZANI, 1995, BRUNK, 1996) apresentaram uma análise e comparações detalhadas entre os diversos sistemas de revisão de teoria existentes na literatura. Esta análise incluiu sistemas como o FORTE (RICHARDS e MOONEY, 1995), A3 (WOGULIS, 1994) e PTR+ (KOPPEL et al., 1994). As comparações apresentadas resultaram em uma classificação e uma metodologia de avaliação de sistemas de revisão de teorias quanto à capacidade de identificação e localização de erros na teoria inicial, independentemente da capacidade de consertá-los.

De acordo com BRUNK (1996), a análise e comparação do sistema FORTE (*First Order Revision of Theories from Examples*) frente aos outros sistemas demonstraram uma maior abrangência no seu espaço de busca de teorias revisadas, em diferentes domínios, desta forma obtendo teorias revisadas potencialmente melhores, ou seja, com maior acurácia. Em decorrência de ter um espaço de busca maior, o sistema FORTE trata um número maior de pontos de revisão, pois gera e avalia mais candidatos durante a busca.

No contexto do PDBDOO, BAIÃO (2001) escolheu o sistema FORTE para a realização do procedimento de refinamento do algoritmo de análise (e mostra sua

eficiência) levando em consideração os resultados mostrados em BRUNK (1996). Portanto, nesta dissertação seguiremos esta escolha para o procedimento de refinamento do algoritmo de fragmentação vertical.

O sistema FORTE automaticamente refina um conjunto de regras que receberá como entrada uma implementação baseada em regras de primeira ordem (i.e., uma implementação em Prolog) do algoritmo VFNBC (encontrado no apêndice A) e um conjunto de exemplos e fatos associados.

Mais detalhes sobre o sistema FORTE podem ser encontrados em (RICHARDS e MOONEY, 1995, BRUNK, 1996, BAIÃO *et al.*, 1999) e a sua utilização para revisar o algoritmo de análise do PDBDOO pode ser encontrado em (BAIÃO, 2001).

Nas próximas seções detalharemos as fases da abordagem baseada em conhecimento para o refinamento do algoritmo de fragmentação vertical VFNBC, denominada "Revisão de Teorias de VFNBC" que indicaremos por TRVFNBC (Theory Revision on the Vertical Fragmentation algorithm from Navathe, extended by Baião and Cruz).

A abordagem é composta das seguintes fases: definição e construção da teoria inicial do domínio (apresentada na seção 4.4) e escolha do conjunto de exemplos (seção4.5). Na seção 4.4 será apresentada a metodologia para a construção do programa em Prolog (4.4.1), a visão geral da construção do programa seguindo esta metodologia (4.4.2), a modelagem dos dados de entrada a fim de representar toda a informação requerida pelo sistema (teoria inicial de domínio e conjunto de exemplos) de forma adequada para o sistema FORTE (4.4.3) e a divisão da Teoria Inicial de Domínio, conforme requerida pelo sistema FORTE, como Teoria Fundamental de Domínio (predicados que não devem ser revisados) e Teoria Inicial (predicados que devem ser revisados). No apêncice C é encontrado um exemplo desses arquivos para a revisão de aresta de extensão.

4.4 Definição e construção da teoria inicial do domínio

O objetivo desta fase é extrair e explicitar todo o conhecimento existente sobre o problema, e representá-lo de forma adequada. O conhecimento a ser extraído pode estar tanto sob o domínio de especialistas no assunto, quanto implícito em resultados experimentais de trabalhos apresentados anteriormente na literatura. No escopo do

presente trabalho, este conhecimento é o algoritmo de fragmentação vertical VFNBC exposto no capítulo 3.

Todo este conhecimento extraído deve então ser estruturado e representado de forma adequada à realização do processo de revisão de teorias. O algoritmo VFNBC foi então implementado como um conjunto de cláusulas de Horn de primeira ordem (i.e., um programa Prolog) e o exemplo dado em (NAVATHE e RA, 1989) como um caso de teste ("test case").

4.4.1 METODOLOGIA PARA A CONSTRUÇÃO DO VFNBC EM PROLOG

Nesta sub-seção é apresentada a metodologia seguida para criar o programa em Prolog referente ao VFNBC, que inclusive pode ser seguida para a elaboração de outros programas em Prolog que se tenha um pseudo-código do algoritmo bem detalhado.

4.4.1.1 Pseudo-código X Prolog

Nesta seção é feito um breve mapeamento entre comandos básicos existentes em pseudo-códigos com cláusulas em Prolog.

Devemos lembrar primeiramente que Prolog trabalha com cláusulas de Horn de primeira ordem, ou seja, um subconjunto de lógica de primeira ordem, onde cada cláusula só pode conter no máximo um literal positivo na cabeça da cláusula (máximo um consequente positivo).

4.4.1.1.1 "Ifs" aninhados

Podemos ter em Prolog, por exemplo, a seguinte cláusula:

$$p(X) \leftarrow q(X), r(a) \tag{5}$$

onde p, q e r são predicados, X é uma variável e a uma constante. Podemos pensar que é equivalente a

$$\text{se } (q = X \text{ e } r = \text{'a'}) \text{ então } p=X \tag{6}$$

Desta forma, para o pseudo-código

$$\text{se } (q = X \text{ e } r = \text{'a'}) \text{ então } p = X \tag{7}$$
$$\text{senao } p = \text{'b'}$$

temos o equivalente em Prolog:

$$p(X) \leftarrow q(X), r(a), !. \tag{8}$$
$$p(b). \tag{9}$$

onde ! representa um *cut* em Prolog e indica que será feita uma redução no espaço de busca (LLOYD, 1987), ou seja, se for provada a cláusula em (8) não será tentado provar a cláusula (9), e sendo assim a idéia é equivalente a (7).

Se ao invés de "e" no pseudo-código (7) fosse preciso representar um "ou", ou seja,

se (q = X ou r = 'a') então p = X (10)
senao p = 'b'

podemos então construir o equivalente em Prolog como:

p(X) ← q(X), !. (11)
p(X) ← r(a), !.
p(b).

Utilizando as idéias supramencionadas podemos facilmente traduzir "ifs" aninhados de pseudo-código para Prolog.

4.4.1.1.2 Função recursiva

Uma função recursiva pode ser facilmente representada em Prolog, como por exemplo, para calcular o fatorial de um número:

fat(0,1):-!. (12)
fat(A,Result):- B is A-1, fat(B,R2), Result is R2 * A. (13)

No exemplo, em (12) e (13), é chamado recursivamente o predicado fat. Em (13) o fatorial de um número A é representado como sendo o fatorial de A-1 multiplicado por A, até que seja passado A = 0, onde seu fatorial é 1 por (12) e parando a recursividade. No final o fatorial de A é retornado em Result.

4.4.2 VFNBC EM PROLOG

Nesta sub-seção veremos a construção do programa em Prolog. A modelagem dos dados de entrada será apresentada na próxima sub-seção. Foram usadas as idéias expostas na sub-seção anterior para fazer um mapeamento do pseudo-código apresentado na Fig. 4.6 (a mesma que a Fig. 3.17 do capítulo anterior, somente copiado neste capítulo para facilitar o entendimento).

A estrutura geral do programa é encontrada na Fig. 4.3 e o programa em Prolog pode ser encontrado no apêndice A.

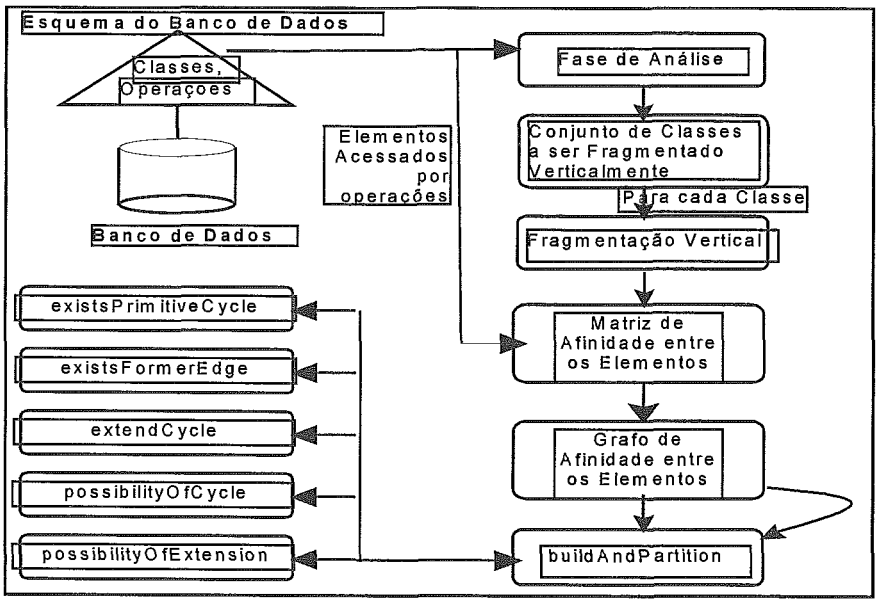


Fig. 4.3 - Estrutura geral da implementação em Prolog para a FV da classe

O predicado buildAndPartition é chamado recursivamente até que todas as arestas sejam consideradas para formar os fragmentos da classe. Algumas destas cláusulas são ilustradas na Fig. 4.4 e Fig. 4.5. Este conjunto de regras constitui um ponto inicial muito bom para que o processo de ILP possa obter o algoritmo revisado. Verificamos no capítulo 3, seção 3.9, que o experimento de fragmentação vertical na classe *AtomicPart* pertencente ao *benchmark* OO7 (CAREY *et al.*, 1993) mostrou que apenas 10% de todas as possíveis fragmentações verticais para tal classe tem o custo total de processamento (RUBERG, 2001) abaixo do custo da fragmentação encontrada por VFNBC e que tais situações são casos extremos em que o *overhead* de gerência do SGBDs sobre um grande número de fragmentos pode não ser conveniente.

```
findPartition(FirstNode,N,ListPartition):-
retractall(partition(_)),
  retractall(cut(_,_)),
  retractall(candidateNewEdge(_,_,_)),
  retractall(removedProvisorilyEdge(_,_,_)),
  assert(cut(0,0)),
  createListNewEdge(ListCandidateNewEdge),
  createCandidateNewEdge(ListCandidateNewEdge),
  buildAndPartition([FirstNode],FirstNode,
  0,0,0,0,0,0,[],[],[],N,FirstNode),
  findall(P,partition(P),ListPartition_dup),
  list_to_set(ListPartition_dup, ListPartition),
```

Fig. 4.4 - Ponto inicial do FVNB implementado em Prolog VFNBC.

```
buildAndPartition(Tree,Begin,End,CycleNode,
NodeCompletingEdge,WeightCompletingEdge,
NodeFormerEdge,WeightFormerEdge,PrimitiveCycle,
CandidatePartition,Partition,N,Node):-
  selectNewEdge(NewNode,BiggestEdge,NodeConnected,
   Begin,End),
  adjustLimits(NewNode,Begin,End,NewBegin,NewEnd,
   NodeConnected),
  refreshTree(Tree,NewNode,NewBegin,NewEnd,NewTree),
  notExistsPrimitiveCycle(NewNode,NodeConnected,
   CycleNode,Tree,Begin,End,NewPrimitiveCycle,
   NewNodeCompletingEdge,NewWeightCompletingEdge),
  CandidatePartition==[],
  removeCandidateNewEdge(NewNode,NodeConnected,
   BiggestEdge),
  T is N-1,
  buildAndPartition(NewTree,NewBegin,NewEnd,CycleNode,
   NodeCompletingEdge,WeightCompletingEdge,
   NodeFormerEdge,WeightFormerEdge,PrimitiveCycle,
   CandidatePartition,Partition,T,NewNode),!.
```

Fig. 4.5 - Uma das regras utilizadas para a construção do grafo e busca da partição

Nesta sub-seção daremos ênfase à contrução em Prolog da função apresentada na Fig. 4.3 como buildAndPartition, apresentado detalhadamente na Fig.4.6 através da função ConstruçãoPartiçãoGrafoAfinidadesElementos. Cada caso, numerado de 1 a 9, apontado na Fig. 4.6 representa um "if"aninhado e na linha 13 da mesma figura pode ser visto que encontramos um comando de repetição (*repeat*). O predicado buildAndPartition é chamado recursivamente até que todas as arestas sejam consideradas para formar os fragmentos da classe. Ele será chamado recursivamente, sempre passando nos seus atributos o estado em que se encontra a construção do grafo, tais como a árvore geradora linearmente conectada obtida até o momento, a aresta de extensão, partição candidata e partição.

Para cada caso identificado na Fig. 4.6 há uma cláusula com o predicado builAndPartition na cabeça com corpo diferente, de acordo com o caso que este está representando.

```
        função ConstrucaoPartiçãoGrafoAfinidadesElementos(Ck: classe a ser verticalmente
               fragmentada, M: matriz de afinidade entre os elementos)
        retorna particoesCk: conjunto das partições da classe Ck
        Início
 (1)     N = conjunto vazio de nós
 (2)     A = conjunto vazio de arestas
 (3)     grafo G = (N,A)
 (4)     N  += qualquer elemento de Ck
 (5)     Cria lista de Arestas Candidatas AC inicialmente vazia AC=nil
 (6)     Cria lista de Arestas Removidas Temporariamente AR inicialmente vazia AR=nil
 (7)     ∀ Ei, Ej tal que M(Ei, Ej) <> 0, insere (Ei, Ej, M(Ei, Ej)) na lista AC
 (8)     enquanto (AC <> nil ) faca
 (9)     Início
(10)      se (((existe aresta que possa ser conectada a Partição Candidata cp) ou
(11)          (não existe Partição Candidata cp)) ) entao {Negação de Caso –1 }
(12)      Início
(13)       repita  {Escolhe uma aresta que não faça parte de uma partição ou partição candidata}
(14)         escolhe (Ei,Ej, M(Ei,Ej))= maior valor de M(Ei,Ej) existente em AC tal que Ei seja uma
(15)                                     extremidade do grafo e Ej seja um novo nó a ser inserido
(16)         aij : = aresta entre Ei e Ej
(17)            se (Ej ∈ Partição Candidata) ou ( ∃P ∈ particoesCk tal que Ej ∈ P) entao {Caso 0 }
(18)         Início
(19)           AC - = aij
(20)           aij : = nil
(21)         Fim
(22)       até (aij <> nil ou AC = nil ) {Fim da escolha da aresta}
(23)       se aij <> nil  então
(24)       Início
(25)         se a aresta aij forma um ciclo em G então
(26)         início{Casos 2,3,4,5}
(27)           seja ca este ciclo
(28)           se existe outro ciclo entao{Caso 5}
(29)           Início
(30)              AC -= aij
(31)              AR += aij
(32)           Fim
(33)           senao
(34)             se ca pode ser um ciclo de afinidades entao{Casos 2,3}
(35)             Início
(36)               marca ca como partição candidata
(37)               AC -= aij
(38)               A+= aij
(39)             Fim
(40)             senao{ Caso 4}
(41)               AC -= aij
(42)         Fim{Casos 2,3,4,5}
(43)         Senao
(44)         Início{Casos 1,6,7.1,7.2,8,9}
(45)           se existe uma partição candidata cp então{Casos 6,7.1,7.2,8,9}
(46)             se cp não pode ser estendido por aij então {Casos 7.1,7.2,8,9}
(47)             Início
(48)               se aij não está conectado a uma partição candidata então {Caso 7.1}
(49)               Início
(50)                 AC -= aij
(51)                 AR += aij
(52)               fim
(53)               Senão
(54)               Início
(55)                 corte aij
(56)                 se existe aresta extensão ae então{Casos 8, 9}
```

```
(57)              se cp pode ser estendido por ae entao {Caso 9}
(58)              Início
(59)                seja E_r o nó de ae que não está em cp
(60)                cp = cp estendido por ae
(61)                AC -=a_ij      N += E_j
(62)                A += a_ri      A += a_ij
(63)              Fim
(64)              senao{Caso 8}
(65)              Início
(66)                marque cp como uma partição
(67)                particoesCk  += cp
(68)                AC -= a_ij
(69)                AC += AR    AR = nil
(70)                A += a_ij        N += E_j
(71)              Fim
(72)            Senao{Caso 7.2}
(73)            Início
(74)              Marca cp como partição
(75)              ParticoesCk += cp
(76)              AC -= a_ij
(77)              AC += AR      AR = nil
(78)              A += a_ij        N += E_j
(79)            Fim
(80)          Fim { fim dos casos 7.1,7.2,8,9}
(81)          Senao{Caso 6}
(82)          Início
(83)            seja E_k o nó de ciclo de cp
(84)            cp = cp estendido por a_ij e a_kj
(85)            AC -= a_ij   AC -= a_kj
(86)            A += a_kj    A += a_ij      N += E_j
(87)          Fim
(88)        Senao {Caso 1}
(89)        Início
(90)          AC -= a_ij
(91)            A += a_ij
(92)            N += E_j
(93)        Fim
(94)      Fim
(95)    {Se existe Partição Candidata e não há mais arestas em AC que possam ser conectadas a esta}
(96)     Senao{ Caso –1 }
(97)     Início
(98)       ParticoesCk + = cp
(99)       AC += AR
(100)      AR = nil
(101)    fim
(102)    fim{enquanto}
(103)    se existe uma partição candidata cp então
(104)      particoesCk + = cp
(105)    seja c = todos os elementos de Ck que pertencem a nenhuma partição de particoesCk
(106)    particoesCk + = c
       Fim.
```

Fig. 4.6 – Construção do Grafo de Afinidades

Um fluxograma com todos os casos está representado na Fig. 4.7 - (a) e partes são detalhadas na Fig.4.7 – (b), quando há a formação de ciclo e Fig.4.7 (c), quando não há formação de ciclo. Todos os caso são detalhados a seguir.

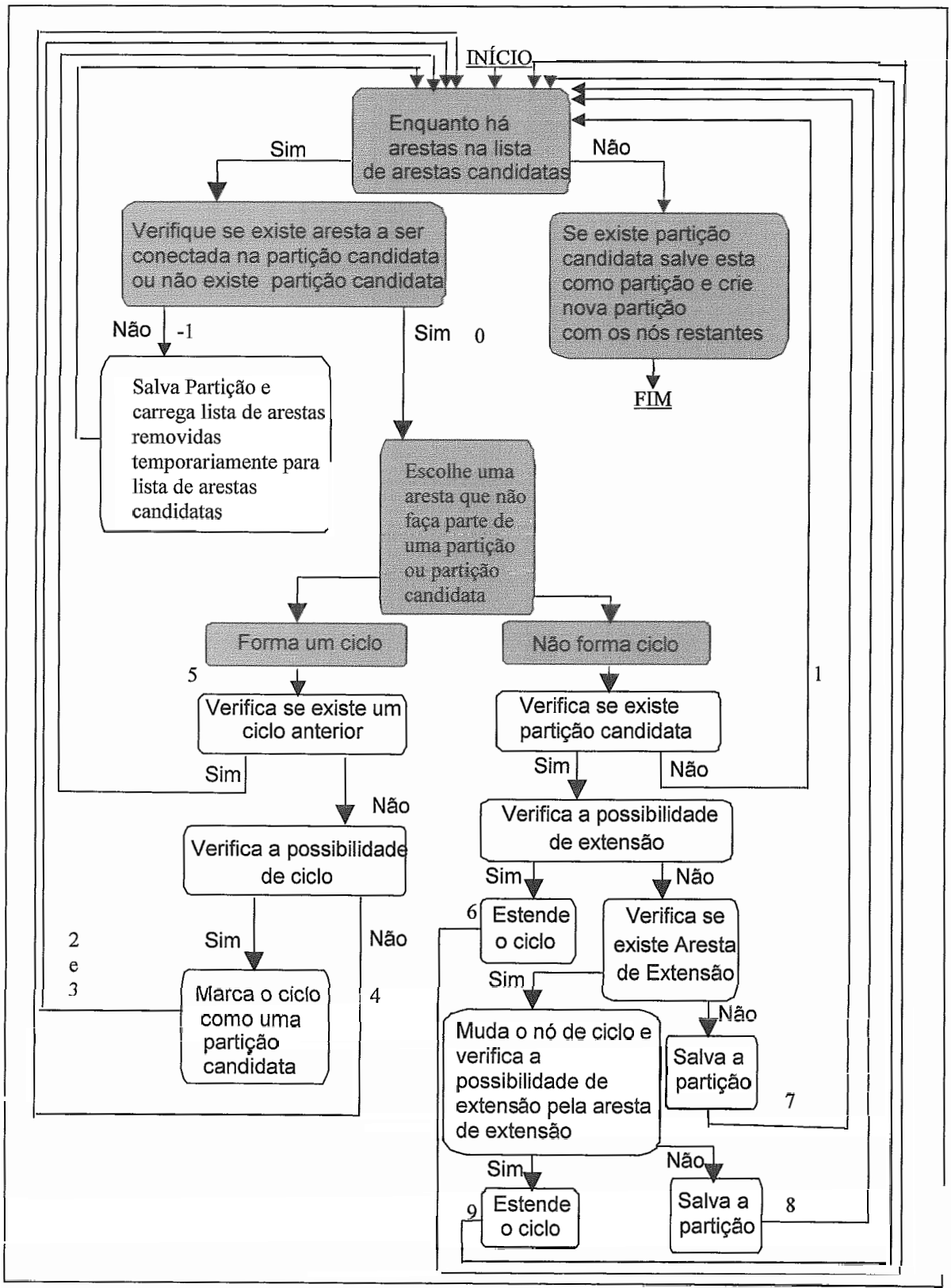


Fig. 4.7 - (a) – Fluxograma Geral

Devemos ressaltar primeiramente que escolher uma nova aresta, nos casos abaixo relacionados com exceção do caso 0, significa verificar se há aresta na lista de arestas candidatas (indicadas nas **Fig. 4.7 – (a), (b) e (c)) e seguir todos os passos posteriores.**

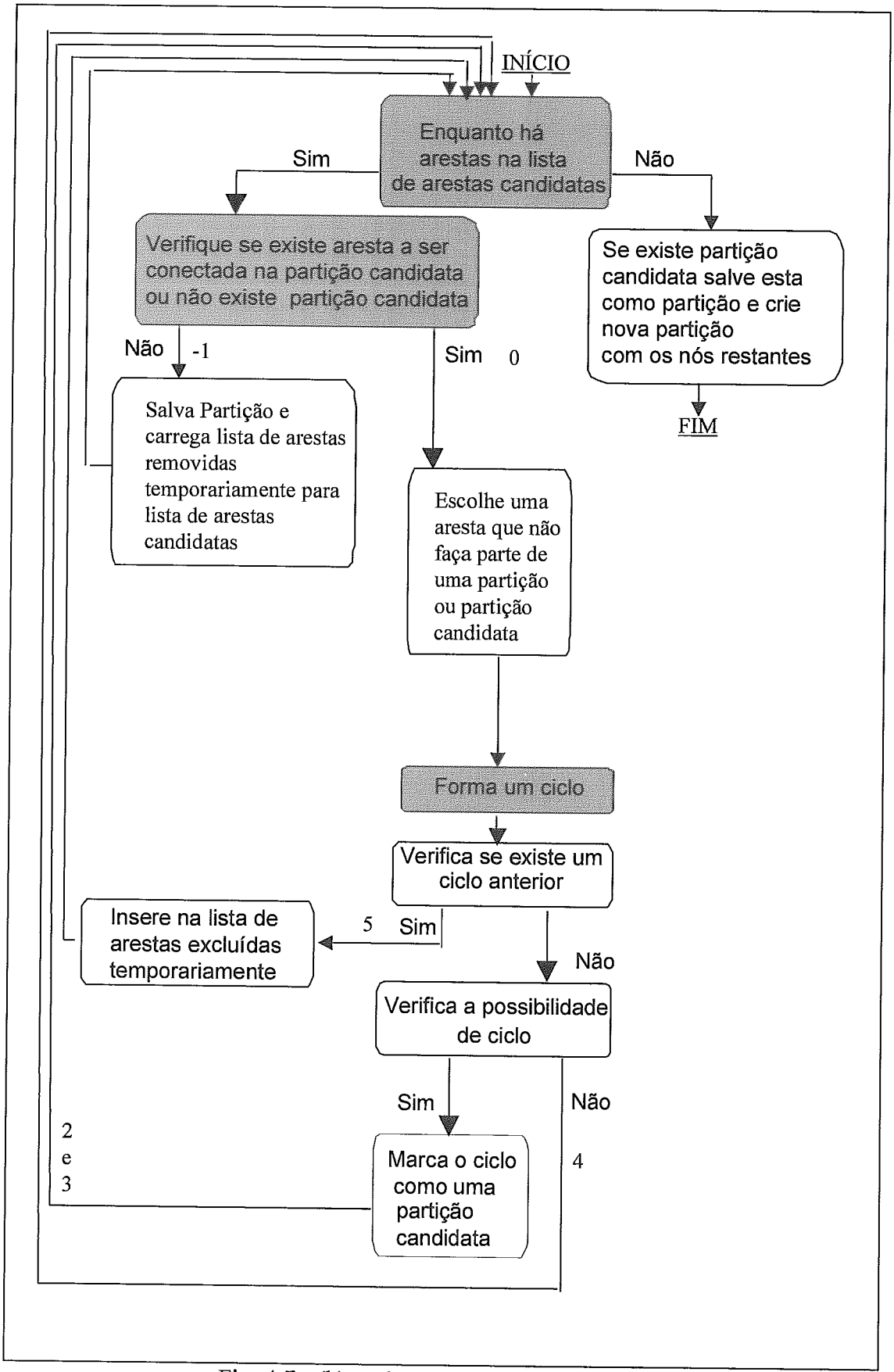


Fig. 4.7 - (b) – Fluxograma da Formação de Ciclo

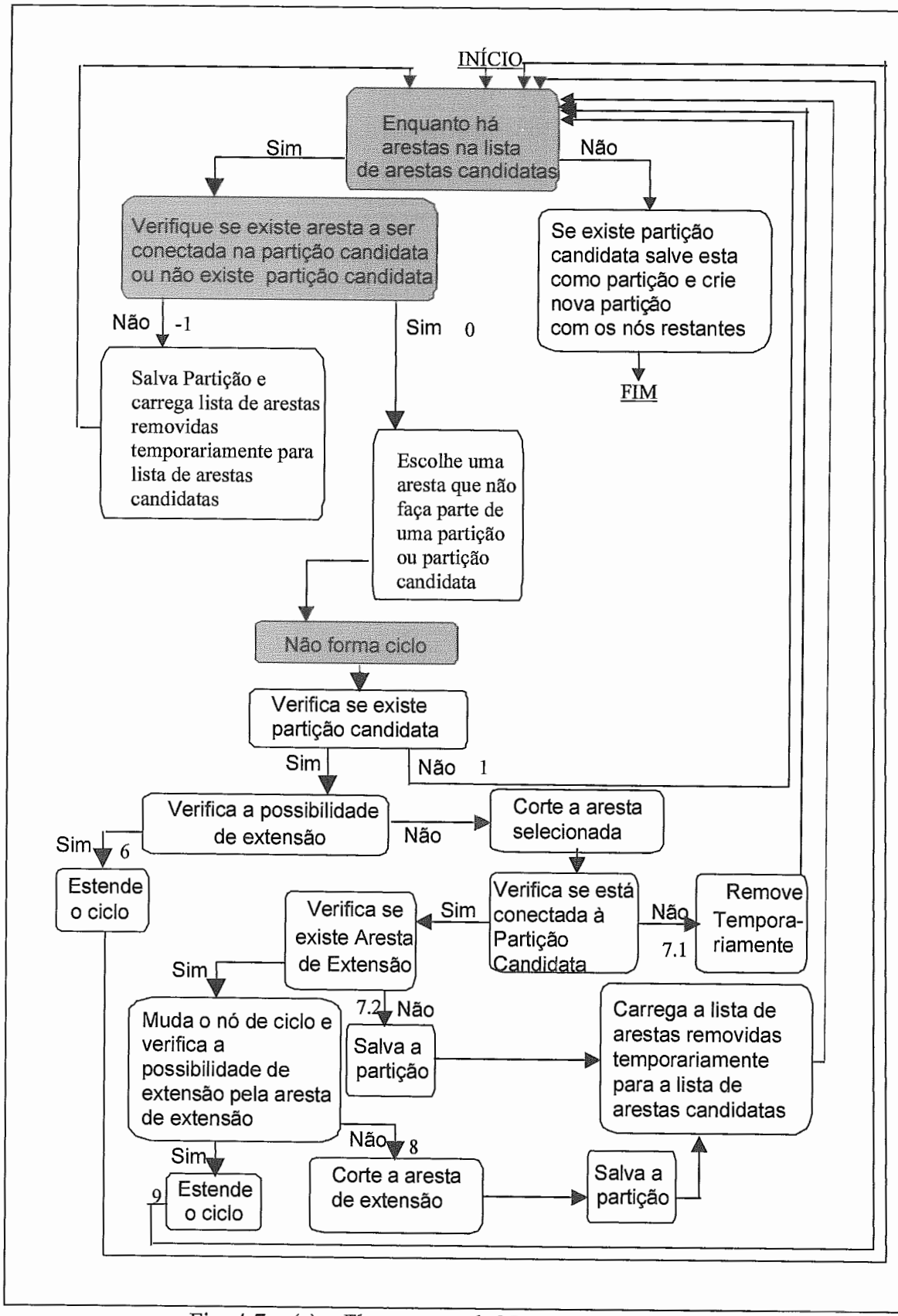


Fig. 4.7 – (c) – Fluxograma de Não Formação de Ciclo

- Casos –1: Se existe partição candidata e não há mais arestas na lista de arestas candidatas que possam ser conectadas a esta então a partição candidata se tornará partição (já que não teremos mais arestas para tentar estender a partição candidata).
- Caso 0: Escolha uma aresta que não faça parte de uma partição candidata ou partição, ou seja, o nó que será conectado a uma das pontas da árvore encontrada até o momento não pode pertencer a partição candidata ou uma partição já formada anteriormente.
- Caso 1: A aresta escolhida não forma ciclo, não existe partição candidata, retira da lista de candidatos a nova aresta inserida e escolhe a próxima aresta.
- Caso 2: A aresta escolhida forma ciclo, não existe ciclo anterior, satisfaz a Possibilidade de Ciclo e marca este como Partição Candidata. Neste caso satisfaz a parte 2 da Possibilidade de Ciclo: p(Aresta de Extensão)<= (todas as arestas do ciclo).
- Caso 3: A aresta escolhida forma ciclo, não existe ciclo anterior, satisfaz a Possibilidade de Ciclo e marca este como Partição Candidata. Neste caso satisfaz a parte 1 da Possibilidade de Ciclo: não existe Aresta de Extensão.
- Caso 4: A aresta escolhida forma ciclo, não existe ciclo anterior, não satisfaz a Possibilidade de Ciclo (p(former edge) ≥ (alguma aresta do ciclo)), retira da lista de candidatos a nova aresta e escolhe próximo nó.
- Caso 5: A aresta escolhida forma ciclo, existe ciclo anterior, retira provisoriamente a aresta da lista de candidatos a nova aresta e escolhe o próximo nó.
- Caso 6: A aresta escolhida não forma ciclo, existe Partição Candidata, existe Possibilidade de Extensão, estende o ciclo, retira da lista de candidatos a nova aresta e escolhe o próximo nó.
- Caso 7.1: A aresta escolhida não forma ciclo, existe Partição Candidata, não existe Possibilidade de Extensão pois o nó escolhido não está conectado a um nó que faz parte da Partição Candidata, remove temporariamente a aresta como candidata e escolhe o próximo nó.
- Caso 7.2: A aresta escolhida não forma ciclo, existe Partição Candidata, não existe Possibilidade de Extensão, não existe Aresta de Extensão, salva a partição, retira da lista de candidatos a nova aresta, carrega novamente as arestas

que foram removidas temporariamente para a lista de arestas candidatas e escolhe o próximo nó.

- Caso 8: A aresta escolhida não forma ciclo, existe Partição Candidata, não existe Possibilidade de Extensão, existe Aresta de Extensão, troca o Nó de Ciclo e tenta estender pelo Aresta de Extensão. Não há Possibilidade de Extensão pela Aresta de Extensão, salva a partição, retira da lista de candidatos a nova aresta, carrega novamente as arestas que foram removidas temporariamente para a lista de arestas candidatas e escolhe o próximo nó.
- Caso 9: A aresta escolhida não forma ciclo, existe Partição Candidata, não existe Possibilidade de Extensão, existe Aresta de Extensão, troca o Nó de Ciclo e tenta estender pelo Aresta de Extensão. Há Possibilidade de Extensão pelo Aresta de Extensão, estende o ciclo, retira da lista de candidatos a nova aresta, acha o novo Aresta de Extensão e escolhe o próximo nó.

Cada caso mencionado acima foi explicado com detalhes no capítulo 3. No apêndice B é encontrada uma tabela para cada caso indicando o que deve ser feito em cada termo de buildAndPartition (quais se deve ser atualizado quando for feita a chamada recursiva ou passado com o mesmo valor que recebeu na chamada). Além disso são identificadas também as chamadas para retirada da aresta da lista de arestas candidatas e os momentos quando a lista de arestas removidas temporariamente devem ser passadas para a lista de arestas candidatas.

4.4.3 Modelagem dos Dados de Entrada

O sistema FORTE foi o escolhido para o módulo de revisão de teorias no escopo desta dissertação. No entanto, para que esse sistema pudesse ser utilizado foi necessário realizar a axiomatização do problema, a fim de modelar e representar toda a informação requerida pelo sistema (teoria inicial de domínio e conjunto de exemplos) de forma adequada.

É necessária uma modelagem dos dados de entrada bem específica, já que é preciso identificar especificamente cada estrutura utilizada no grafo, tal como as arestas, os nós e os ciclos. Estas e outras estruturas serão apresentadas nesta seção.

Primeiramente poderíamos pensar em modelar a Matriz de Afinidades entre os Elementos (definida na seção 3.3.2), como por exemplo, através do predicado matriz da forma

$$\text{matriz}(E_i,E_j,\text{aff}_{ij}) \qquad (14)$$

ou seja, o valor de afinidade entre os elementos E_i e E_j, que representam os nós, é dado por aff_{ij}, que representa o peso da aresta entre os dois nós E_i e E_j. Porém desta forma, não teríamos como diferenciar de forma adequada para o sistema FORTE os tipos dos termos do predicado que são nós dos que são pesos de aresta.

Foram então criados predicados para identificar os nós, as arestas, os pesos das arestas, os ciclos, cortes de arestas e a árvore geradora linearmente conectada. Além disso, para que possamos definir para o sistema FORTE que podem ser utilizados predicados built-in para ser feita a revisão de teoria, tais como <. > e =, devemos criar um predicado para cada.

Nas próximas sub-seções são definidos os predicados utilizados na revisão do algoritmo VFNBC.

4.4.3.1 Identificação do nó

Cada nó é identificado pelo predicado

node(idNodei). (15)

onde idNodei é um identificador único do nó i. Exemplos:

node(1). node(2). node(3). (16).

4.4.3.2 Identificação da aresta

Cada aresta é identificada pelo predicado

edge(idEdgeij). (17)

onde idEdgeij é um identificador único da aresta entre os nós i e j definidos conforme (15). Exemplo:

edge(edge12). (18)

será a identificação da aresta entre os nós 1 e 2 definidos nos exemplos em (16).

4.4.3.3 Nós acessados pela aresta

Os nós que são acessados por cada aresta são identificados pelo predicado

edgeAccess(idEdgeij, idNodei, idNodej) (19)

onde idEdgeij é definido por (17) e idNodei e idNodej são definidos por (15). Exemplo:

edgeAccess(edge12, 1, 2). (20)

Desta forma podemos identificar que a aresta edge12 definida em (18) liga o nó 1 ao 2, definidos em (16).

Já que o grafo utilizado no VFNBC não é direcionado, devemos garantir que se a aresta identificada por idEdgeij liga o nó i ao j, o nó j é ligado ao i também. Para isto, foi criado o predicado os predicados mostrados na Fig. 4.8.

```
edgeAccess(idEdgeij, idNodei, idNodej).
edgeAccess(idEdgeij, idNodej, idNodei).
```

Fig. 4.8

4.4.3.4 Identificação do peso

Cada aresta contém um valor ou peso, definido na seção 3.4.3 como p(a), e que contém o valor de afinidade, oriundo da Matriz de Afinidades entre os Elementos, entre os dois elementos (representados pelos nós) da aresta. Este peso será identificado pelo predicado

weight(idEdgeij,valor). (20)

onde idEdgeij é definido por (18) e o valor é dado por $p(a)=aff_{ij}$. Exemplo:

weight(edge12,25). (21)

onde edge12 é definida em (18) e $aff_{12}=25$ (conforme tabela 3.2 do capítulo anterior).

4.4.3.5 Identificação do ciclo

Cada ciclo contém uma identificação, representada pelo predicado

cycle(idCycleck). (22)

onde idCycleck será o identificador do ciclo Ck. Exemplo:

cycle(c1). (23)

identifica o ciclo C1.

4.4.3.6 Nós do ciclo

Após a criação da identificação do ciclo, podemos descrever os nós que compõe este ciclo através do predicado

nodesAccessByCycle(idCyclek, [idNode1,ideNode2,...,idNodei]). (24)

onde idNode1,ideNode2,...,idNodei identifica respectivamente os nós 1 até i que compõe o ciclo identificado por idCycleck, definido conforme (22). Exemplo:

nodesAccessByCycle(c1,[1,2,3]). (25)

identifica que o ciclo C1, definido em (23), é composto pelos nós 1, 2 e 3 definidos em (16).

4.4.3.7 Identificação da árvore geradora linearmente conectada

A árvore geradora linearmente conectada, que utilizamos durante a construção do grafo de afinidades do VFNBC (descrita no capítulo anterior), é identificada por

$$\text{tree(idTreeq).} \tag{26}$$

onde idTreeq denota a árvore q. Exemplo:

$$\text{tree(t1).} \tag{27}$$

representa o identificador da árvore 1.

4.4.3.8 Aresta cortada

Como o FORTE tem limitação com relação ao aprendizado dee cláusulas normais (que tenham operador de negação por falha, ouseja, not no corpo da cláusula), identificamos a aresta cortada (definida no capítulo anterior na seção 3.4.9) é identificada por

$$\text{cut(idEdgeij, booleano).} \tag{28}$$

onde idEdgeij é o identificador da aresta definida por (17) e booleano será definido como:

$$\text{booleano} = \begin{cases} 0, \text{ se a aresta idEdgeij não for uma aresta cortada} \\ 1, \text{ caso contrário} \end{cases}$$

Exemplo:

$$\text{cut(edge12,1).} \tag{29}$$

identifica que a aresta definida conforme (18) é uma aresta cortada.

Esta aresta cortada poderia ser representada também por cut(edgeij) e notCut(edgeij):=\+cut(edgeij). Esta seria uma representação equivalente que também poderia ser utilizada.

4.4.3.9 Predicados embutidos

Ao invés de serem usados predicados embutidos (isto é, pré-definidos pelas bibliotecas do interpretador Prolog que foi utilizado), tais como >, < e =, devem ser criados predicados para cada um. Os predicados utilizados durante a Revisão de Teoria estão na tabela 4.2.

Tabela 4.2 – Predicados embutidos utilizados

equal(X,Y) :- X == Y.	not_equal(X,Y) :- X \== Y.
great(X,Y) :- X > Y.	not_great(X,Y) :- \+ great(X,Y).
less(X,Y) :- X < Y.	not_less(X,Y) :- \+ less(X,Y).

4.4.4 Divisão da Teoria Inicial de Domínio

A teoria inicial de domínio deve ser dividida em dois arquivos: a "teoria fundamental de domínio" (*fundamental domain theory*, arquivo .FDT), que contém predicados que não devem ser revisados pelo FORTE, e a "teoria inicial" (*initial theory*, arquivo .THY), que contém os predicados que devem ser revisados. Ambos serão apresentados nas próximas sub-seções. Um exemplo desses arquivos para a revisão da existência de aresta de extensão é encontrado no apêndice C (seções C.1 e C.2).

4.4.4.1 Teoria Fundamental de Domínio

A teoria fundamental de domínio (arquivo .fdt) é composta dos predicados da teoria de domínio inicial que não devem ser revisados pelo sistema FORTE, ou seja, a parte do algoritmo VFNBC que é assumida previamente como correta. Por exemplo, predicados que definem que se uma aresta conecta um nó i ao j então o nó j também é conectado a i (Fig. 4.9).

```
edgeAccess(idEdgeij, idNodei, idNodej).
edgeAccess(idEdgeij, idNodej, idNodei).
```

Fig. 4.9 – Exemplo de predicado da teoria fundamental

Os predicados built-in que podem ser utilizados na revisão, tais como os apresentados na seção 4.4.3.9, também fazem parte da Teoria Fundamental de Domínio.

Desta forma, quando for feita a revisão da teoria que estará no arquivo .thy (apresentado na próxima sub-seção) estes predicados existentes no .fdt serão utilizados (por exemplo, podem ser inseridos no corpo de uma cláusula existente no arquivo .thy). Por isso devemos colocar os predicados built-in que queremos que sejam utilizados na revisão, senão estes poderão estar no corpo de uma cláusula existente no arquivo .thy porém este não será retirado e será inserido outro predicado built-in que não esteja no arquivo .fdt.

4.4.4.2 Teoria Inicial para ser Revisada

Por outro lado, a teoria inicial é representada pelos predicados da teoria de domínio inicial que devem ser revisados pelo FORTE. Deve ser um programa em Prolog puro, e todas as relações e atributos devem ser precedidos por uma referência explícita ao módulo da teoria fundamental de domínio. Por exemplo, para fazer a revisão do predicado para verificar a existência de uma aresta de extensão, deveríamos colocá-lo no arquivo .thy como apresentado na Fig. 4.10.

```
existsFormerEdge(NewNode,NodeConnected,IdTree,IdFormerEdge):-
        fdt: nodesAccessByTree(IdTree,Tree),
        findFormerEdge(NewNode,NodeConnected,Tree,IdFormerEdge !.
findFormerEdge(NewNode,NodeConnected,Tree,IdFormerEdge):-
        fdt:member(NewNode,Tree),
        fdt:position(P,NewNode,Tree),
        fdt:lastList(End,Tree),
        NodeConnected==End,
        Q is P-1,
        fdt:position(Q,NodeFormerEdge,Tree),
        fdt:existEdgeAccess(IdFormerEdge,NewNode,NodeFormerEdge),!.
findFormerEdge(NewNode,NodeConnected,[H|TailTree],IdFormerEdge):-
        fdt:member(NewNode,[H|TailTree]),
        fdt:position(P,NewNode,[H|TailTree]),
        fdt:lastList(End,[H|TailTree]),
        NodeConnected==H,
        Q is P+1,
        fdt:position(Q,NodeFormerEdge,[H|TailTree]),
        fdt:existEdgeAccess(IdFormerEdge,NewNode,NodeFormerEdge), !.
```

Fig. 4.10 – Teoria Inicial: existência da aresta de extensão

4.5 ESCOLHA DO CONJUNTO DE EXEMPLOS

Outra informação essencial ao sistema FORTE durante o processo de revisão de teorias é o conjunto de exemplos. A representação de um exemplo no FORTE é um termo Prolog com 4 argumentos:

example(PositiveInstances, NegativeInstances, Objects, Facts) (30)

onde *PositiveInstance* (*NegativeInstance*) é uma lista de fatos positivos (negativos) do conceito a ser aprendido, *objects* são a representação do contexto da aplicação (no problema de fragmentação verticalf usando o VFNBC, *objects* são representados pelas características do grafo que devem ser identificadas (que precisaram de identificação única durante a modelagem na seção 4.4.3), como nós (*node*), arestas (*edge*), a árvore (*tree*) e pesos (*weight*)), e *facts* são fatos previamente conhecidos, representados como fatos dos predicados da teoria fundamental de domínio.

Para a abordagem TRVFNBC, as instâncias positivas representam boas fragmentações verticais para uma determinada classe que em conjunto levam à definição a fragmentação vertical que tenha bom desempenho, e que por isso deve ser o resultado do VFNBC revisado. Para TRVFNBC usaremos fragmentações verticais que tem o custo total de processamento (RUBERG, 2001) menor que a fragmentação dada por VFNBC para a classe a ser fragmentada verticalmente (ou menor que um certo valor – ou faixa – abaixo do custo dado por VFNBC). Já as instâncias negativas correspondem à fragmentações verticais ruins, ou seja, com custo total de processamento (RUBERG, 2001) maior que a fragmentação dada por VFNBC (ou maior que um certo valor – ou faixa – acima do custo dado por VFNBC) pois estas levam a fragmentações verticais com desempenho ruim.

Somente para mostrar um exemplo possível sobre a existência de aresta de extensão (Fig.4.10) numa determina árvore, de acordo com os predicados definidos anteriormente nos arquivos FDT e THY, temos o exemplo encontrado na Fig. 4.11 (ver apêndice C.3) .

No exemplo da Fig. 4.12, a instância positiva é dada pelo termo

[existsFormerEdge(3,1,t1,edge34)] (31)

e a instância negativa por

[existsFormerEdge(2,4,t1,edge12)] (32)

O exemplo (31) representa que quando foi inserida a aresta formada pelos nós 3 e 1, onde o nó 1 foi a ponta da árvore t1 onde foi inserida a aresta, foi formado um ciclo e a aresta extensão correspondente é dada por edge34.

Os objetos são dados pelo conjunto de arestas (edge), nós (node) e árvore (tree). Repare que em edge é definido se há corte na aresta ou não por 0 ou 1 após seu identificador. Os fatos são definidos pelas relações existentes entre os objetos (por exemplo, através do predicado edgeAccess é identificado os nós acessados por uma aresta, como a aresta edge12 que acessa os nós 1 e 2). Os exemplos do TRVFNBC são passados ao FORTE em um arquivo de dados (.DAT). O arquivo DAT contém o conjunto de exemplos e também define parâmetros de execução para direcionar o processo de aprendizado do FORTE.

```
Example(     /* Primeiro Exemplo */
      [ existsFormerEdge(3,1,t1,edge34)],
      [ existsFormerEdge(2,4,t1,edge12)],
      [ edge([ [edge12,1],
               [edge13,0],
               [edge14,0],
               [edge15,0],
               [edge23,0],
               [edge24,0],
               [edge25,0],
               [edge34,0],
               [edge35,0],
               [edge45,1]
                ]),
         node([ [1],
                [2],
                [3],
                [4],
                [5]
               ]),
```

```
            tree([ [t1],
                   [t2],
                   [t3]
                 ])
      ], facts([ edgeAccess(edge12,1,2),
            edgeAccess(edge13,1,3),
            edgeAccess(edge14,1,4),
            edgeAccess(edge15,1,5),
            edgeAccess(edge23,2,3),
            edgeAccess(edge24,2,4),
            edgeAccess(edge25,2,5),
            edgeAccess(edge34,3,4),
            edgeAccess(edge35,3,5),
             edgeAccess(edge45,4,5),

          nodesAccessByTree(t1,[1,2,3,4]),
         ])
        ).
```

Fig. 4.11 – Conjunto de Exemplos: existência de aresta de extensão

Capítulo 5

CONCLUSÕES

Este capítulo encerra a apresentação desta tese, apresenta as conclusões e contribuições mais relevantes que foram obtidas durante o seu desenvolvimento, além de apontar direções para trabalhos futuros.

O objetivo deste trabalho é tratar o problema da fragmentação vertical no PDBDOO. Neste sentido, esta dissertação propõe extensões para um algoritmo existente na literatura, além de apresentar uma abordagem baseada em conhecimento para tratar o problema. Foi utilizado um Algoritmo de Fragmentação Vertical (VFNB), sendo propostas modificações para sua melhoria (utilizando inclusive uma notação mais adequada do que a de (NAVATHE e RA, 1989)). O algoritmo VFNB modificado será identificado por VFNBC onde o C indica a modificação apresentada nesta dissertação, ou seja, por CRUZ. Algumas das modificações são apresentadas a seguir.

Foi criada nova definição para aresta de extensão onde foi feita a seguinte mudança:

- Definição de NAVATHE (1989)

  A Aresta de Extensão é definida por NAVATHE (1989) como a aresta que foi selecionada entre o último Corte e o Nó de Ciclo.

- Extensão da definição de NAVATHE (1989)

  Nesta dissertação foi estendida a definição de uma aresta de extensão de NAVATHE (1989) para uma aresta previamente selecionada, não cortada, conectada diretamente ao ciclo. Esta definição abrange as arestas conectadas ao ciclo mas que não estão conectadas a uma aresta cortada.

  Esta mudança obtém melhores fragmentos, como pode ser visto na seção 3.4.10.

Foram apontados diferentes procedimentos que devem ser feitos quando a aresta selecionada não tiver potencial de extensão de um ciclo já existente. Segundo (NAVATHE, 1989) esta deveria ser descartada. Porém, se quando selecionada, por exemplo, for formado outro ciclo, esta deve ser descartada; mas, após a criação da partição respectiva ao ciclo que estávamos tentando estender, a aresta descartada deve ser considerada novamente. Porém, se a aresta selecionada, por exemplo, contiver um nó de uma partição, esta não deve mais ser considerada até o fim da construção do grafo de afinidades (maiores detalhes na seção 3.5.2).

Foi verificada a criação de partição se não houver mais arestas para serem conectadas à partição, o que não era mencionado no trabalho original (seção 3.5.3).

Uma importante modificação foi a do critério de parada do algoritmo (seção 3.5.4). O processo continuará até que a lista de arestas candidatas esteja vazia. Segundo (NAVATHE e RA, 1989) o processo continuaria até serem inseridos todos os nós. Durante a implementação do algoritmo constatamos que, se o algoritmo parasse quando acabassem os nós, alguns ciclos poderiam não ser achados, já que para formar os ciclos

ainda precisariam ser inseridas arestas, inclusive no exemplo apresentado por (NAVATHE e RA, 1989).

Na seção 4.4 são apresentadas a correção do pseudo-código de (NAVATHE e RA, 1989) e a inclusão das modificações para sua melhoria apresentadas nesta dissertação, bem como metodologia para a construção do programa e modelagem dos dados de entrada. Além disso, no apêndice A é apresentada a implementação em Prolog do VFBN, que é uma das principais contribuições desta dissertação.

Outra contribuição que devemos ressaltar é o ganho de custo (apresentado na seção 3.9) obtido em experimentos com o benchmark OO7 (CAREY et al., 1993), quando foi apresentada uma alternativa para "quebrar" os ciclos, ou seja, as partições encontradas pelo VFNB a fim de considerar ciclos com um ou dois nós através da Avaliação Extra das Partições (seção 3.9).

A partição obtida para a fragmentação vertical da classe AtomicPart do OO7 teve o custo total (RUBERG, 2001) para o processamento de 782.727,225 e constatamos que ainda teríamos 9,817% de todas as fragmentações possíveis para AtomicPart com custo menor, mostrando que, ao mesmo tempo que o algoritmo é eficaz, ainda temos como melhorá-lo. Utilizando a Avaliação Extra de Partições, a partição foi dividida em duas, diminuindo o custo para 752.757.225. Portanto, teríamos apenas 1,253% de todas as fragmentações possíveis de *AtomicPart* abaixo deste valor. No entanto, observando tais fragmentações (com valor de custo total menor que 752.757.225), pode ser verficado que em todas são considerados fragmentos distintos para *Id*, *BuilDate* e *to* de *AtomicPart* , ou seja, é criado um fragmento só com *Id*, outro somente com *BuilDate*, outro só com *to* de *AtomicPart* e outros fragmentos com a combinação dos atributos restantes (*type*, x, y, *isRootPart* e *isPartOf*). Este custo menor se deve ao fato de que a função de custo utilizada para a avaliação de RUBERG (2001) é simplificada e não considera a sobrecarga (overhead) de gerência do SGBD sobre um grande número de fragmentos, o que tornaria essas alternativas de fragmentação desvantajosas em uma avaliação experimental.

A Avaliação Extra de Partições pode ainda ser melhorada e, inclusive, pode ser utilizado o Módulo de Revisão de Teoria para revisar a avaliação apresentada, utilizando as fragmentações obtidas pelo Módulo de Busca Exaustiva para gerar exemplos positivos (fragmentações com custo baixo) e negativos (com custo alto) para o FORTE.

O Algoritmo de Fragmentação Vertical foi implementado como um conjunto de regras e usado como conhecimento preliminar para a descoberta de um novo algoritmo revisado através do uso de Revisão de Teoria (WROBEL, 1996). Seguindo estas idéias, é feito um refinamento do algoritmo inicial (representado como um conjunto de regras em Prolog), descobrindo assim um novo conjunto de regras (em Prolog) que representam o algoritmo revisado. Este novo conjunto de regras representará o VFNB revisado, que irá propor esquemas de fragmentação ótimos (ou perto do ótimo) com melhor desempenho.

Foram apresentadas as principais idéias embutidas nesta nova abordagem para o refinamento do VFNB usando um método de aprendizagem de máquina - Programação em Lógica Indutiva (ILP). Esta abordagem realiza um processo de revisão/descoberta de conhecimento utilizando o conjunto de regras como conhecimento preliminar. O objetivo final é descobrir ("aprender") novas heurísticas a serem consideradas no processo de fragmentação vertical.

É importante ressaltar que não foi pretendido aqui obter o melhor AFV possível e sim apontar o processo de revisão de um algoritmo do PDBDOO utilizando técnicas de Descoberta de Conhecimento.

Um importante trabalho futuro é o uso da mesma abordagem de aprendizagem indutiva em outras fases do Projeto de Distribuição (tais como a fase do alocação), assim como no próprio Projeto de Banco de Dados, e utilizar outros modelos dos dados (relacional ou dedutivo). Além disso, o esquema de fragmentação resultante obtido de nosso algoritmo revisado pode ser aplicado para fragmentar a base de dados usada em (PROVOST e HENNESSY, 1996), que propõe o uso de Banco de Dados Distribuídos para melhorar algoritmos de Mineração de Dados.

REFERÊNCIAS

BAIÃO, F., 2001, "Uma Metodologia e Algoritmos para o Projeto de Distribuição de Bases de Dados usando Revisão de Teorias", Tese de D.Sc., versão estendida em Relatório Técnico ES-565/01, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, Brasil, Rio de Janeiro

BAIÃO, F., MATTOSO, M., ZAVERUCHA, G., 1998a, "A Knowledge-Based Perspective of the Distributed Design of Object Oriented Databases". In : *Proceedings of the Int. Conf. on Data Mining 1998,* WIT Press, Rio de Janeiro, Brazil , pp. 383-400

BAIÃO, F., MATTOSO, M., ZAVERUCHA, G., 1998B, "Towards an Inductive Design of Distributed Object Oriented Databases". In: *Proceedings of the Third IFCIS Conference on Cooperative Information Systems (CoopIS'98),* IEEE CS Press, Nova York, EUA, Agosto, pp. 88-197

BAIÃO, F., MATTOSO, M., ZAVERUCHA, G., 1999, "A Theory Refinement Approach to the Design of Distributed Object Oriented Databases". Relatório Técnico ES-493/99, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, Brasil, Rio de Janeiro

BAIÃO, F., MATTOSO, M., ZAVERUCHA, G., 2000, "Horizontal Fragmentation in Object DBMS: New Issues and Performance Evaluation". In: *Proceedings of the 19th IEEE International Performance, Computing and Communications Conference (IPCCC 2000)*, IEEE CS Press, Phoenix, pp.108-114

BAIÃO, F., MATTOSO, M., ZAVERUCHA, G., 2001, "A Distribution Design Methodology for Object DBMS", submetido em Agosto 2000; manuscrito revisado enviado em Novembro 2001 à *International Journal of Distributed and Parallel Databases*, Kluwer Academic Publishers

BASÍLIO, R., ZAVERUCHA, G., BARBOSA, V., 2001, "Learning Logic Programs with Neural Networks". In : *Proceedings of the 11th Int. Conf. on Inductive Logic Programming (ILP),* Lectures Notes in Artificial Intelligence, Vol. 2157. Springer-Verlag, Strasbourg, France, pp. 15-26

BELLATRECHE, L., KARLAPALEM, K., BASAK, B., 1998, "Query-Driven Horizontal Class Partitioning in Object-Oriented Databases". In: *Proceedings of the 9th International Conference on Databases and Expert Systems (DEXA'98),* Lecture Notes in Computer Science, vol. 1460, Vienna, Áustria, pp. 692-701

BELLATRECHE, L., KARLAPALEM, K., SIMONET, A., 2000, "Algorithms and Support for Horizontal Class Partitioning in Object-Oriented Databases", *International Journal of Distributed and Parallel Databases*, Kluwer Academic Publishers, vol. 8(2), pp. 155-179

BELLATRECHE, L., SIMONET, A., SIMONET, M., 1996, "Vertical Fragmentation in Distributed Object Database Systems with Complex Attributes and Methods". In: *Proceedings of the "7th International Workshop on Database and Expert Systems Applications" (DEXA'96)*, IEEE Computer Society, Zurich, Suíça, pp. 15-21

BLOCKEEL, H, DE RAEDT, L., 1996, "Inductive Database Design", In: RAS, Z., MICHALEWICZ, M. (eds.), *Proceedings of the 10th International Symposium on Methodologies for Intelligent Systems* (ISMIS'96), LNAI 1079, pp. 376-385, Springer-Verlag

BLOCKEEL, H, DE RAEDT, L., 1998, "IsIdd: an Interactive System for Inductive Database Design", *Applied Artificial Intelligence*, 12(5), pp. 385-420

BRUNK, C., 1996, "An Investigation of Knowledge Intensive Approaches to Concept Learning and THEORY Refinement", PhD Thesis, University of California, Irvine, EUA

CAREY, M., DEWITT, D., NAUGHTON, J., 1993, "The OO7 Benchmark". In: *Proceedings of the 1993 ACM SIGMOD,* vol. 22(2), Washington DC, pp. 12-21

CATTEL, R. *et al.*, 2000, "The Object Data Standard: ODMG 3.0", Morgan Kaufmann Publishers Inc., São Francisco, EUA

CHEN, Y., SU, S., 1996, "Implementation and Evaluation of Parallel Query Processing Algorithms and Data Partitioning Heuristics in Object Oriented Databases", *International Journal of Distributed and Parallel Databases*, Kluwer Academic Publishers, vol. 4(2), pp. 107-142

CORNELL, D., YU, P., 1987, "A Vertical Partitioning Algorithm for Relational Databases". In: *Proceedings of the 3rd International Conference on Data Engineering (ICDE'87),* Los Angeles, EUA, pp. 30-35

CRUZ, F., BAIÃO, F., MATTOSO, M., ZAVERUCHA, G., 2002, "Towards a Theory Revision Approach for the Vertical Fragmentation of Object Oriented Databases". A ser publicado em *Proceddings of the XVI Brazilian Symposium on Artificial Intelligence (SBIA'02),* Lectures Notes in Artificial Intelligence, Springer-Verlag, Recife, Rio de Janeiro

EISENBERG, A., MELTON, J., 1999, "SQL 1999, formerly known as SQL 3". In: *Proceedings of the 1999 ACM SIGMOD*, vol. 28(1), pp. 131-138

EZEIFE, C., BARKER, K., 1998, "Distributed Object Based Design: Vertical Fragmentation of Classes", *International Journal of Distributed and Parallel Databases*, Kluwer Academic Publishers, vol. 6(4), pp. 317-350

GARCEZ, A. S., ZAVERUCHA, G., 1999, "The Connectionist Inductive Learning and Logic Programming System". *Applied Intelligence Journal*, Vol. 11(1), pp. 59-77

GETOOR, L., FRIEDMAN, N., KOLLER, D., TASKAR, B., 2001a, "Probabilistic Models of Relational Structure", In: *Proceedings of the Int. Conf. On Machine Learning*, Williamstown, MA

GETOOR, L., TASKAR, B., KOLLER, D., 2001b, "Selectivity Estimation using Probabilistic Models", In: *Proceedings of the 2001 ACM SIGMOD.* Santa Barbara, Califórnia, USA, pp. 461-472

KARLAPALEM, K., NAVATHE, S., MORSI, M., 1994, "Issues in Distribution Design of Object-Oriented Databases". In: ÖZSU, M. *et al.* (eds.), *Distributed Object Management*, Morgan Kaufmann Publishers Inc., São Francisco, EUA, pp. 148-164

KHOSHAFIAN, S., COPELAND, G., 1986, "Object Identity". In: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, pp. 406-416 – also In: SIGPLAN Notices, vol. 21(11)

LAVRAC, N., DZREROSKI, S., 1994, "Inductive Logic Programming: Techniques and Applications", Ellis Horwood

LLOYD, J., 1987, "Foundations of Logic Programming", 2nd extend edition, Berlin, Springer-Verlag

MALINOWSKI, E., 1996, "Fragmentation Techniques for Distributed Object-Oriented Databases", MSc. Thesis, University of Florida, EUA

MCCORMICK, W., SCHWEITZER, P., WHITE, T., 1972, "Problem Decomposition and Data Reorganization by a Clustering Technique", *Operational Research*, vol. 20(5), pp. 993-1009

MESQUITA, E. J. S., FINGER, M., 1998, "Projeto de Dados em Bancos de Dados Distribuídos". In: *XIII Simpósio Brasileiro de Banco de Dados (SBBD98)*, pp. 87-102

MITCHELL, T., 1997, "Machine Learning", McGraw-Hill Companies Inc.

MOLINA, H., HSU, M., 1995, "Distributed Databases". In: W. KIM (ed.), *Modern Database Systems*, ACM Press, pp. 484-485

MUGGLETON, S., DE RAEDT, L., 1994, "Inductive logic programming: Theory and methods", *Journal of Logic Programming*, Vol. 19(20), pp. 629-679

NAVATHE, S., CERI, S., WIEDERHOLD, G., DOU, J., 1984, "Vertical Partitioning Algorithms for Database Design", *ACM Transactions on Database Systems*, vol. 9(4), pp. 680—710

NAVATHE, S., KARLAPALEM, K., RA, M., 1995, "A Mixed Fragmentation Methodology for Initial Distributed Database Design", *Journal of Computer and Software Engineering*, vol. 3(4)

NAVATHE, S., RA, M., 1989, "Vertical Partitioning for Database Design: A Graphical Algorithm". In: *Proceedings of the 1989 ACM SIGMOD*, Portland, Oregon, EUA, pp. 440-450

ÖZSU, M., VALDURIEZ, P., 1999, "Principles of Distributed Database Systems", 2nd edition (1st edition 1991), New Jersey, Prentice-Hall

PROVOST, F., HENNESSY, D., 1996, "Scaling-Up: Distributed Machine Learning with Cooperation", In: *Proceedings of AAAI,* AAAI Press, Portland, Oregon, pp. 74-79

RICHARDS, B., MOONEY, R., 1995, "Refinement of First-Order Horn-Clause Domain Theories", *Machine Learning*, vol. 19(2), pp. 95-131

RUBERG, G., 2001, "Um Modelo de Custos para o Processamento de Consultas em Bases de Objetos Distribuídos". Tese de M.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil

TOWELL, G., SHAVLIK, J., 1994, "Knowledge-Based Artificial Neural Networks", Artificial Intelligence, Vol.70 (1-2), pp. 119-165

WROBEL, S., 1996, "First Order Theory Refinement". In: L. De Raedt (ed.), *Advances in Inductive Logic Programming*, IOS Press

APÊNDICE A

ALGORITMO VFNB EM PROLOG

```
%Acha as seguintes particoes(P):
%P = [[],[8,2,9,3],[1,5,7],[10,6,4]] ;
%Estas sao exatamente as particoes achadas no artigo de (NAVATHE, 1989)

:-ensure_loaded(library(lists)).
:-ensure_loaded(library(sets)).
:-ensure_loaded(library(basics)).

:- dynamic partition/1.
:- dynamic cut/3.
:- dynamic candidateNewEdge/3.
:- dynamic removedProvisorilyEdge/3.
/*****************************************************

   O objetivo deste arquivo eh encontrar as particoes
verticais construindo um grafo e encontrando os ciclos
existentes no mesmo.
Data: 21/02/02
Flavia Cruz
Artigos:
  1.Vertical Partioning for Database Design: A Graphical Algorith
    Shamkant B. Navathe and Minyoung Ra1989

*****************************************************/
```

```
/*********************************************************
            INICIALIZACOES
*********************************************************/

/*********************************************************
          Attribute Affinity - AA
aa(no_1, no_2, w) w is an integer variable for the weight of the edge do no 1 para o 2
*********************************************************/

/*********************************************************
          Attribute Affinity - AA
Exemplo do artigo Nava89
*********************************************************/
/*
aa(1,2,25).
aa(1,3,25).
aa(1,4,0).
aa(1,5,75).
aa(1,6,0).
aa(1,7,50).
aa(1,8,25).
aa(1,9,25).
aa(1,10,0).
aa(2,3,75).
aa(2,4,0).
aa(2,5,25).
aa(2,6,0).
aa(2,7,60).
aa(2,8,110).
aa(2,9,75).
aa(2,10,0).
aa(3,4,15).
aa(3,5,25).
aa(3,6,15).
aa(3,7,25).
aa(3,8,75).
aa(3,9,115).
aa(3,10,15).
aa(4,5,0).
aa(4,6,40).
aa(4,7,0).
aa(4,8,0).
aa(4,9,15).
aa(4,10,40).
aa(5,6,0).
aa(5,7,50).
```

```
aa(5,8,25).
aa(5,9,25).
aa(5,10,0).
aa(6,7,0).
aa(6,8,0).
aa(6,9,15).
aa(6,10,40).
aa(7,8,60).
aa(7,9,25).
aa(7,10,0).
aa(8,9,75).
aa(8,10,0).
aa(9,10,15).
*/
/*********************************************************
        Attribute Affinity - AA
Exemplo do artigo Nava94
An Objetive Funcion for Vertically Partioning Relations in
Distributed Databases and Its Analysis

É o mesmo exemplo do Nava89!!!
*********************************************************/
/*********************************************************
        Attribute Affinity - AA
Exemplo do livro do Ozsu
Deu como resposta uma particao com todos os nos=[1,2,3,4].
Isto porque nao considera ciclos de 2 nós.
*********************************************************/

/*aa(1,2,0).
aa(1,3,45).
aa(1,4,0).

aa(2,3,5).
aa(2,4,75).

aa(3,4,3).
*/
```

```
/*******************************************************
        Attribute Affinity - AA
Eu inventei...so para testar...
*******************************************************/
aa(1,2,30).
aa(1,3,30).
aa(1,4,25).
aa(1,5,0).
aa(1,6,0).
aa(1,7,0).
aa(2,3,55).
aa(2,4,25).
aa(2,5,0).
aa(2,6,25).
aa(2,7,25).
aa(3,4,65).
aa(3,5,40).
aa(3,6,65).
aa(3,7,65).
aa(4,5,75).
aa(4,6,100).
aa(4,7,100).
aa(5,6,75).
aa(5,7,75).
aa(6,7,100).
```

```
/*******************************************************
      Outras variaveis para inicializar
*******************************************************/

/*******************************************************
          Auxiliary funcions
*******************************************************/

/*******************************************************
position(-+Pos,+-Elem,+List)

Pode ser utilizada para encontrar a posicao(-Pos) de um
elemento(+Elem) de uma certa lista(+List) ou para encontrar
um elemento(-Elem) em uma posicao(+Pos)de uma lista(+List).
*******************************************************/

position(1, Elem, [Elem|_]).
position(Cnt, Elem, [_|Tail]) :-
      position(P, Elem, Tail),
        Cnt is P + 1.


/*******************************************************
existsPrimitiveCycle(+K, +B, +NodeConnected, +CycleNode,
+Arvore, +Inicio, +Fim, -PrimitiveCycle,-NodeCompletingEdge,
-WeightCompletingEdge)

Indica se existe um primitive cycle.
Saídas: PrimitiveCycle conterá uma lista com os nós do ciclo formado
e serão identificados o NodeCompletingEdge e WeightCompletingEdge
(descritos em nodesCycle).

Utiliza o predicado nodesCycle.
*******************************************************/

existsPrimitiveCycle(K, B, NodeConnected, CycleNode, Arvore, Inicio, Fim,
PrimitiveCycle,
 NodeCompletingEdge,WeightCompletingEdge):-
                    % CycleNode==0,
                     member(K,B),
                    % position(P,K,B),
                    % P\==1,
                     nodesCycle(K, NodeConnected, Arvore, Inicio, Fim, PrimitiveCycle,
                       NodeCompletingEdge,WeightCompletingEdge)
                          /*,
```

```
                    %MUDEI:::
                    partition(P),
                    member(M,PrimitiveCycle),
                    \+member(M,P)
                    */
                    .

notExistsPrimitiveCycle(K, B, NodeConnected, CycleNode, Arvore, Inicio, Fim,
PrimitiveCycle,
  NodeCompletingEdge,WeightCompletingEdge):-
      \+ existsPrimitiveCycle(K, B, NodeConnected, CycleNode, Arvore, Inicio, Fim,
          PrimitiveCycle, NodeCompletingEdge,WeightCompletingEdge).

existsCycle(K, B, NodeConnected, CycleNode, Arvore, Inicio, Fim, Cycle,
 NodeCompletingEdge,WeightCompletingEdge):-
                  % CycleNode==0,
                   member(K,B),
                  % position(P,K,B),
                  % P\==1,
                    nodesCycle(K, NodeConnected, Arvore, Inicio, Fim, Cycle,
                     NodeCompletingEdge,WeightCompletingEdge).

notExistCycle(K, B, NodeConnected, CycleNode, Arvore, Inicio, Fim, Cycle,
 NodeCompletingEdge,WeightCompletingEdge):-
 \+existsCycle(K, B, NodeConnected, CycleNode, Arvore, Inicio, Fim, Cycle,
 NodeCompletingEdge,WeightCompletingEdge).

/*************************************************************
nodesCycle(+K,+NodeConnected,+Arvore,+Inicio,+Fim,-Ciclo,
-NodeCompletingEdge, -WeightCompletingEdge)

Encontra o nós que formam o ciclo e devolve a lista com estes nós
em "Ciclo".
Quando o nó novo K forma um ciclo e K foi conectado ao Fim da Arvore,
o Ciclo será a sublista contida na lista Arvore que começa em K
e termina em Fim. Quando K foi conectado ao Inicio da Arvore,
o Ciclo será a sublista contida na lista Arvore que começa em Inicio
e termina em K.

NodeCompletingEdge: Nó que juntamente com o CycleNode forma o
            cycle completing edge
WeightCompletingEdge: Peso do lado que formou o ciclo
             (peso do cycle completing edge)

Utiliza sublista(Xs,AsXsBs) e salvaSublista(Inicio,Fim,Lista,Sub)
*************************************************************/
```

```
nodesCycle(K,NodeConnected, Arvore, Inicio, Fim, Ciclo, Fim, Weight):-
          write('Nó escolhido foi o'),
          write(K),
          NodeConnected==Fim,
          write('Posicao = Fim'),
          write('****Chamada Fim do ciclo: '),
          write(salvaSublista(K,Fim,Arvore,Ciclo)),nl,
          salvaSublista(K,Fim,Arvore,Ciclo),
          write('****Retorno da chamada fim do ciclo: '),
          write(salvaSublista(K,Fim,Arvore,Ciclo)),nl,
          achaPeso(K,Fim,Weight),
           !.

nodesCycle(K,NodeConnected, Arvore, Inicio, Fim, Ciclo, Inicio,Weight):-
          write('Nó escolhido foi o'),
          write(K),
       NodeConnected==Inicio,
          write('Posicao = Inicio'),
          write('****Chamada Inicio do ciclo: '),
       write(salvaSublista(Inicio,K,Arvore,Ciclo)),nl,
          salvaSublista(Inicio,K,Arvore,Ciclo),
          write('****Retorno da chamada inicio do ciclo: '),
          write(salvaSublista(Inicio,K,Arvore,Ciclo)),
          achaPeso(K,Inicio,Weight),
          !.

/*************************************************************
sublista(+Xs,+AsXsBs)
Este predicado será verdadeiro se Xs for sublista de AsXsBs.
*************************************************************/
sublista(Xs,AsXsBs):-append(As,XsBs,AsXsBs),append(Xs,Bs,XsBs).

/*************************************************************
salvaSublista(+Inicio,+Fim,+Lista,-Sub)
Sub é sublista de Lista que começa com o elemento Inicio
e termina com o elemento Fim.
*************************************************************/
salvaSublista(Inicio,Fim,Lista,[Inicio|T]):-
       sublista([Inicio|T],Lista),
          last(Fim,[Inicio|T]).
```

```
/*************************************************************
existsFormerEdge(+K_novo,+B,+NodeConnected,+Arvore,+Inicio,+Fim,
        -NodeFormerEdge, -WeightFormerEdge)
Indica se existe um Former Edge.
O Former Edge será a aresta formada pelos nós K_novo e NodeFormerEdge
e terá peso WeightFormerEdge.

Utiliza o predicado findFormerEdge.
*************************************************************/
existsFormerEdge(K_novo,B,NodeConnected,Arvore,Inicio,Fim,NodeFormerEdge,
WeightFormerEdge):-
        %       member(K_novo,B),
        %       position(P,K_novo,B),
        %       P\==1,
        %       Q is P-1,
        %       position(Q,Elem,B),
                %quero achar Elem, talvez tenha q usar posicao
        %       (Elem) > 0,

findFormerEdge(K_novo,NodeConnected,Arvore,Inicio,Fim,NodeFormerEdge,WeightFor
merEdge),
                write('FORMER
EDGE:('),write(K_novo),write(','),write(NodeFormerEdge),write(')='),write(WeightFormer
Edge),nl,
                write('Verificando se não há cut no Former Edge...'),
                \+cut(K_novo,NodeFormerEdge,WeightFormerEdge),
                write('não há!!!'),nl.


notExistsFormerEdge(K_novo,B,NodeConnected,Arvore,Inicio,Fim,NodeFormerEdge,
WeightFormerEdge):-

\+existsFormerEdge(K_novo,B,NodeConnected,Arvore,Inicio,Fim,NodeFormerEdge,
WeightFormerEdge).


/*************************************************************
findFormerEdge(+K_novo,+NodeConnected,+Arvore,+Inicio,+Fim,
        -NodeFormerEdge, -WeightFormerEdge)
O Former Edge será a aresta formada pelos nós K_novo e NodeFormerEdge
e terá peso WeightFormerEdge.

Se K_novo foi inserido no final da Arvore e formou um ciclo entao
a aresta do former edge eh formada pelo nó anterior ao K_novo na
Arvore e K_novo, se foi no inicio será o nó posterior.
*************************************************************/
```

```
findFormerEdge(K_novo,NodeConnected,Arvore,Inicio,Fim,NodeFormerEdge,WeightFor
merEdge):-
          member(K_novo,Arvore),
               position(P,K_novo,Arvore),
               write('Posicao de K-novo é '), write(P), write('na arvore '), write(Arvore),nl,
           (
           (NodeConnected==Fim,
               write('Posicao de Fim='), write(Fim),write(' = NodeConnected ='),
write(NodeConnected),nl,
             Q is P-1,
               position(Q,NodeFormerEdge,Arvore),
               write('Posicao de NodeFormerEdge='), write(NodeFormerEdge),write(' é '),
write(Q), write('na arvore '), write(Arvore),nl,
             !);
           (NodeConnected==Inicio,
             write('Posicao de Inicio='), write(Inicio),write(' = NodeConnected ='),
write(NodeConnected),nl,
               Q is P+1,
               position(Q,NodeFormerEdge,Arvore),
               write('Posicao de NodeFormerEdge='), write(NodeFormerEdge),write(' é '),
write(Q), write('na arvore '), write(Arvore),nl,
             !)),
            ( aa(K_novo,NodeFormerEdge,WeightFormerEdge);
            aa(NodeFormerEdge,K_novo,WeightFormerEdge)).



/*************************************************************
possibilityOfCycle(+K_novo,+Ciclo,+NodeFormerEdge,+WeightFormerEdge)

Retornarar como verdadeiro ou falso todo o predicado.

Verifica se satisfaz a possibility of cycle: no former edge exist
or p(former edge) <= p(all the cycle edges). Nao verificaremos aqui
quando o former edge não existe, somente a 2a condição.
*************************************************************/

possibilityOfCycle(K_novo,[X,Y],NodeFormerEdge,WeightFormerEdge):-
       ( aa(X,Y,Peso);
         aa(Y,X,Peso)),
          (  ((WeightFormerEdge)<(Peso)) ; ((WeightFormerEdge)==(Peso)) ).

possibilityOfCycle(K_novo,[X,Y|Ciclo],NodeFormerEdge,WeightFormerEdge):-
       ( aa(X,Y,Peso);
         aa(Y,X,Peso)),
          (  ((WeightFormerEdge)<(Peso)) ; ((WeightFormerEdge)==(Peso)) ),
```

```
        possibilityOfCycle(K_novo,[Y|Ciclo],NodeFormerEdge,WeightFormerEdge).

notPossibilityOfCycle(K_novo,[X,Y|Ciclo],NodeFormerEdge,WeightFormerEdge):-
    \+ possibilityOfCycle(K_novo,[X,Y|Ciclo],NodeFormerEdge,WeightFormerEdge).


/*************************************************************
possibilityOfExtension(+Cycle,+K_novo,+NodeConnected,+Weight,+OtherNode)

Retornarar como verdadeiro ou falso todo o predicado.

Verifica se satisfaz a Possibilidade de extensao do ciclo:
P(edge being considered or cycle completing edge) >=
P(any one of the cycle edges).

NodeConnected: nó do ciclo que K se conectou
NodeExtend: nó pelo o qual esta se tentando estender
OtherNode: o outro nó do ciclo (será igual ao CycleNode ou o nó que
junto com o CycleNode formam o lado que completou o ciclo
se foi inserido no CycleNode o OtherNode sera o NodeCompleting
     NodeConnected
  O  _ 0 _ 0 NodeExtend
   \|
    0 OtherNode
*************************************************************/

possibilityOfExtension([X,Y],K_novo,NodeConnected,Weight,OtherNode):-
write(possibilityOfExtension([X,Y],K_novo,NodeConnected,Weight,OtherNode)),
        write('Aresta do ciclo: '),
        ( (   aa(X,Y,Peso),  write(aa(X,Y,Peso)),nl);
          (   aa(Y,X,Peso),  write(aa(Y,X,Peso)),nl)  ),
           write('Completing edge:('),
        ( (aa(K_novo,OtherNode,Weight_2),
write(K_novo),write(','),write(OtherNode),write(')='),write(Weight_2), nl);
          (aa(OtherNode,K_novo,Weight_2),
write(OtherNode),write(','),write(K_novo),write(')='),write(Weight_2), nl)),
           write('Edge being considered:(
'),write(K_novo),write(','),write(NodeConnected),write(')='),write(Weight),nl,
           (  ((Weight)>(Peso)) ; ((Weight)==(Peso)); ((Weight_2)>(Peso)) ;
((Weight_2)==(Peso)) ).

possibilityOfExtension([X,Y|Ciclo],K_novo,NodeConnected,Weight,OtherNode):-
write(possibilityOfExtension([X,Y],K_novo,NodeConnected,Weight,OtherNode)),
        write('Aresta do ciclo: '),
        ( (aa(X,Y,Peso),write(aa(X,Y,Peso)),nl);
          (aa(Y,X,Peso),write(aa(Y,X,Peso)),nl)),
           write('Completing edge: '),
```

```
        ( (aa(K_novo,OtherNode,Weight_2),
write(K_novo),write(','),write(OtherNode),write(')='),write(Weight_2),nl);
          (aa(OtherNode,K_novo,Weight_2),
write(OtherNode),write(','),write(K_novo),write(')='),write(Weight_2), nl)),
           write('Edge being considered:(
'),write(K_novo),write(','),write(NodeConnected),write(')='),write(Weight),nl,
           (  ((Weight)>(Peso)) ; ((Weight)==(Peso)); ((Weight_2)>(Peso)) ;
((Weight_2)==(Peso)) ),
           possibilityOfExtension([Y|Ciclo],K_novo,NodeConnected,Weight,OtherNode).

notPossibilityOfExtension([X,Y|Ciclo],K_novo,NodeConnected,Weight,OtherNode):-
    \+ possibilityOfExtension([X,Y|Ciclo],K_novo,NodeConnected,Weight,OtherNode).

/*******************************************************
Algoritmo
*******************************************************/

/*****************************************************************************
***
1.Ajuste dos ponteiros:
-ajustaPonteiros(+K, F1, F2, -F1_novo, -F2_novo, +NodeConnected)
Utilizado para ajustar os ponteiros relacionados com a lista "b" de nós já
visitados. f2 sempre aponta para o último nó escolhido.
-AjustaLimites(+K,+Inicio,+Fim,-InicioNovo,-FimNovo,+NodeConnected)
Utilizado para ajustar os ponteiros relacionados com a lista "arvore" de nós
já visitados em ordem. "Inicio" sempre aponta para o início da árvore e "Fim"
para o fim da árvore.

Ex.: Suponha que o nó escolhido K for um nó que se conecta ao nó J e este é
apontado por "Inicio" e "f1". Além disso, o nó L é apontado por "Fim" e "f2".
Entao teremos os seguintes ajustes: "Inicio" apontará para K, "Fim" apontará
para L, "f1" para L e "f2" para K.
*****************************************************************************
***/

/*******************************************************
ajustaPonteiros(+K, +F1, +F2, -F1_novo, -F2_novo,NodeConnected)

Ajusta ponteiro sabendo que o no K foi selecionado e neste
momento o ponteiro f1 aponta para o no "F1" e f2 aponta para
o nó "F2". O novo f2 sempre aponta para o último nó inserido
e NodeConnected é o nó ao qual K_novo foi conectado.
*******************************************************/
ajustaPonteiros(K, F1, F2, F1, K, NodeConnected):-
          F2==0,
            NodeConnected==F1,
          !.
```

```
ajustaPonteiros(K, F1, F2, F2, K, NodeConnected):-
            F2\==0,
            NodeConnected==F1,
                  !.
ajustaPonteiros(K, F1, F2, F1, K, NodeConnected).

/*****************************************************
ajustaLimites(+K,+Inicio,+Fim,-InicioNovo,-FimNovo, +NodeConnected)

Ajusta os limites da árvore: se incluir um nó conectado ao
nó apontado por Inicio então Inicio apontará para este novo nó.
Se incluir no fim da árvore então Fim apontará para o
novo nó.
*****************************************************/
ajustaLimites(K, Inicio, Fim, Inicio, K, NodeConnected):-
            Fim==0,
            NodeConnected==Inicio,
               !.
ajustaLimites(K, Inicio, Fim, K, Fim, NodeConnected):-
            Fim\==0,
               NodeConnected==Inicio,
               !.
ajustaLimites(K, Inicio, Fim, Inicio, K, NodeConnected).


/*************************************************************************
***
2.Atualização das listas: Arvore e B.juste dos ponteiros:
-atualizaB(+B, +K, -B_novo)
-atualizaArvore(+Arvore,+K_novo,+Inicio_Novo,+Fim_Novo,-Arvore_Nova)
*************************************************************************
***/

/*****************************************************
atualizaB(+B, +K, -B_novo)

Atualiza B para o novo B(B_novo) colocando o nó escolhido
K no final da lista B de nós ja visitados.

Utiliza o predicado insere.
*****************************************************/

insere(X,Ys,Zs):-select(X,Zs,Ys).

atualizaB(B,K,K_novo,B_novo):-
    position(PosK, K, B),
    insere(K_novo, B, B_aux),
```

```
    PosKNovo is PosK +1,
    position(PosKNovo, K_novo, B_aux),
    select(0,B_aux, B_novo), !.

/*******************************************************
atualizaArvore(+Arvore,+K_novo,+Inicio_Novo,+Fim_Novo,-Arvore_Nova)

Atualiza a arvore(Arvore) para a nova arvore(Arvore_Nova)
colocando K_novo no inicio da arvore se
K_novo=Inicio_novo e no fim se K_novo=Fim_Novo.
*******************************************************/

atualizaArvore(Arvore,K_novo,Inicio_Novo,Fim_Novo,[K_novo|Arvore]):-
  K_novo==Inicio_Novo.
atualizaArvore(Arvore,K_novo,Inicio_Novo,Fim_Novo,Arvore_Nova):-
  K_novo==Fim_Novo,
  append(Arvore,[K_novo],Arvore_Nova).

/*************************************************************************
*
3.Seleção da próxima aresta a ser inserida na spanning tree.
A próxima aresta deve ser linearly connected com a spanning tree já construída
e deve ter o maior valor entre todas as possibilidades.
selectNewEdge(-K,-Largest,-NodeConnected,+Inicio,+Fim)
*************************************************************************
***/

%Cria uma lista com todas as arestas existentes no predicado aa(atribute affinity)
createListNewEdge(LRev) :-
  findall([Weight,Node1,Node2],achaPeso(Node1,Node2,Weight),L_dup),
  list_to_set(L_dup,L),
  sort(L,LSort),
  reverse(LSort,LRev).
%Para todos os elementos da lista criada em createListNewEdge é dado um assert
%em candidateNewEdge
createCandidateNewEdge([[W,X,Y]|T]):-
  assert(candidateNewEdge(X,Y,W)),
  createCandidateNewEdge(T).

createCandidateNewEdge([]).

%Ordena a lista de candiadtos a nova aresta em ordem decrescente.
sortListCandidateNewEdge(LRev):-
  findall([Weight,Node1,Node2],candidateNewEdge(Node1,Node2,Weight),L_dup),
  list_to_set(L_dup,L),
  sort(L,LSort),
  reverse(LSort,LRev).
```

```
%Verifica se o nó selecionado é linearmente conectado à Spanning Tree.
%Para tal, este deve ser conectado ao Inicio ou Fim da Arvore.
isLinearllyConnected(NodeSelected,Weight,Inicio,Inicio,Fim):-
  candidateNewEdge(Inicio,NodeSelected,Weight);
  candidateNewEdge(NodeSelected,Inicio,Weight).

isLinearllyConnected(NodeSelected,Weight,Fim,Inicio,Fim):-
  candidateNewEdge(Fim,NodeSelected,Weight);
  candidateNewEdge(NodeSelected,Fim,Weight).

%Seleciona a aresta de maior peso que é conectada lineramente à Spanning Tree.
selectLargestCandidateNewEdge(NodeSelected,Weight,NodeConnected,Inicio,Fim,[[Weig
ht,NodeSelected,NodeConnected]|_]):-
   isLinearllyConnected(NodeSelected,Weight,NodeConnected,Inicio,Fim),!.

selectLargestCandidateNewEdge(NodeSelected,Weight,NodeConnected,Inicio,Fim,[H|T]):
-
   selectLargestCandidateNewEdge(NodeSelected,Weight,NodeConnected,Inicio,Fim,T).

%Seleciona a nova aresta a ser inserida(que tem o maior peso entre todas que podem ser
%linearmente conectadas à spanning tree.
selectNewEdge(NodeSelected,Weight,NodeConnected,Inicio,Fim):-
   sortListCandidateNewEdge(LSort),

selectLargestCandidateNewEdge(NodeSelected,Weight,NodeConnected,Inicio,Fim,LSort).

%Remove uma candidata a nova aresta.
%É utilizada normalmente após a inserção de uma aresta na Arvore.
removeCandidateNewEdge(X,Y,W):-
   retract(candidateNewEdge(X,Y,W)),
   retract(candidateNewEdge(Y,X,W)).

removeAllCandidateNewEdge :-
   retractall(candidateNewEdge(_,_,_)).

%Remove temporariamente uma candidata a nova aresta guardando esta em
%removedProvisorilyEdge (aresta provisoriamente removida)
%removeProvisorilyCandidateNewEdge(+X,+Y,+W):-

removeProvisorilyCandidateNewEdge(X,Y,W):-
   assert(removedProvisorilyEdge(X,Y,W)),
   assert(removedProvisorilyEdge(Y,X,W)),
   retract(candidateNewEdge(X,Y,W)),
   retract(candidateNewEdge(Y,X,W)).
```

```
%Cria uma lista com todas as arestas provisoriamente removidas
createListRemovedProvisorilyEdge(L) :-
  findall([Weight,Node1,Node2],removedProvisorilyEdge(Node1,Node2,Weight),L_dup),
  list_to_set(L_dup,L).

%Carrega todas as arestas removidas provisoriamente. Para isto:
%1.Cria uma lista com todas as arestas removidas temporariamente;
%2.Para cada elemento da lista acima, recoloca a aresta em candidateNewEdge;
%3.Limpa todas as arestas removidas temporariamente

loadAllRemovedProvisorilyEdge(L):-
  createListRemovedProvisorilyEdge(L),
  createCandidateNewEdge(L),
  retractall(removedProvisorilyEdge(_,_,_)).

%achaPeso(+J,+K_novo,-Peso)
%Acha Peso do lado (J,K_Novo) na matriz de afinidades (aa)
achaPeso(J,K_novo,Peso):-
  (aa(J,K_novo,Peso)
  ;aa(K_novo,J,Peso)).

%Estende o ciclo(Cycle) inserindo o no (Node) em Ciclo e passando este em NewCycle
extendCycle(Cycle,Node,NewCycle):-
  append(Cycle,[Node],NewCycle).

%isCandidateExtension(Node,Cycle)
%Verifica se Node faz parte do ciclo (Cycle).
%Quando for verificado se K é um candidato a extensão, o nó da ávore
%onde este se conectou deverá ser passado em Node.
isCandidateExtension(Node,Cycle):-
  member(Node,Cycle).

notIsCandidateExtension(Node,Cycle):- \+ isCandidateExtension(Node,Cycle).

%Verifica se eh candidato a extensao do ciclo
checkCandidateExtension(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio, Fim,
CandidatePartition,

NewCycle,NewNodeCompletingEdgeExtension,NewWeightCompletingEdgeExtension):-
  ((
   isCandidateExtension(NodeConnected,CandidatePartition),
   write(isCandidateExtension(NodeConnected,CandidatePartition)),
   existsCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio, Fim, NewCycle,
        NewNodeCompletingEdgeExtension,NewWeightCompletingEdgeExtension),
   write('Ciclo:'),write(NewCycle),
   member(CycleNode,NewCycle)
  );
```

```
  (isCandidateExtension(NodeConnected,CandidatePartition),
   write(isCandidateExtension(NodeConnected,CandidatePartition)),
   notExistCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio, Fim, NewCycle,
        NewNodeCompletingEdgeExtension,NewWeightCompletingEdgeExtension),
   write('Não forma ciclo pelo nó que está tentando estender:'),write(NewCycle)
   )).



%Salva um cut entre dois nós
saveCut(Node1,Node2,Weight):-
  assert(cut(Node1,Node2,Weight)),
  assert(cut(Node2,Node1,Weight)),
  write('SALVOU OS CUTs:'),write(cut(Node1,Node2,Weight)),
  write(' e '), write(assert(cut(Node2,Node1,Weight))),nl.

%Salva particao
savePartition(CandidatePartition):-
   assert(partition(CandidatePartition)),
   write('SALVOU A PARTICAO:'),write(CandidatePartition).


%Confere se existe ainda nó que deva ser conectado a um "end" da spanning tree.
%Para isto deve existir uma aresta que contém o nó End e que não existe um cut nesta
aresta
%e o nó que junto com o End forma aresta não deve pertencer a uma particao.
%existsNodeConnectedEnd(+End)
%O predicado será true ou false


existsNodeConnectedEnd(End):-
  candidateNewEdge(End,_,_);
  candidateNewEdge(_,End,_).
/* achaPeso(End,NodeCandidate,Weight),
 partition(P),
 \+ member(NodeCandidate,P),
 \+ cut(End,NodeCandidate,Weight),
 write(achaPeso(End,NodeCandidate,Weight)),
 write(partition(P)),
 write('Nao faz parte de particao:'), write(member(NodeCandidate,P)),
 write('Nao ha cut:'),write(cut(End,NodeCandidate,Weight)).
*/


notExistsNodeConnectedEnd(End):-
 \+ existsNodeConnectedEnd(End),!.
```

```
%CASO -1: Verifica se ainda há nós a serem conectados a um end que faz parte de
%um candidate partition.
%Se nao houver, marca o candidate partition como particao e escolhe o proximo no
%(que com certeza sera conectado ao outro end da arvore).
%Descarta o nó escolhido e escolhe outro se este fizer parte de alguma particao.
%          End1 and NodeCompletingEdge
%       O _0 --------0 Node (exist???)
%        \| WheightCompletingEdge
%          0 CycleNode
%          |
%          0
%          |
%          0 End2
buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
PrimitiveCycle,CandidatePartition,Partition,N,K):-
 nl,
 write('******* CASO -1 ********'),nl,   write('N='), write(N),
 CandidatePartition\==[],
%MUDEI:::
 member(NodeCompletingEdge,CandidatePartition),
 notExistsNodeConnectedEnd(NodeCompletingEdge),
 write(notExistsNodeConnectedEnd(NodeCompletingEdge)),
 savePartition(CandidatePartition),
 T is N,


 write('Candidate partition:'), write(CandidatePartition),nl,
 write('foi salvo como particao:'), write(savePartition(CandidatePartition)),nl,
 write('*******Numero de nos que faltam ser inseridos na arvore: '), write(T), nl,

 buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
  0,0,0,0,0,[],[],Partition,T,K),!.


%CASO 0: Descarta o nó escolhido e escolhe outro se este fizer parte de alguma particao.
%ou se fizer parte de uma particao candidata.
buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
PrimitiveCycle,CandidatePartition,Partition,N,K):-
 nl,
 write('******* CASO 0 ********'),nl,   write('N='), write(N),
 selectNewEdge(K_novo,Maior,NodeConnected,Inicio,Fim),
 %Nó escolhido faz parte de alguma particao:
```

```
 partition(P),
 (member(K_novo,P);member(K_novo,CandidatePartition)),
 % saveCut(K_novo,NodeConnected,Maior),
 removeProvisorilyCandidateNewEdge(K_novo,NodeConnected,Maior),
 T is N,
 write('  No
escolhido='),write(K_novo),nl,write(selectNewEdge(K_novo,Maior,NodeConnected,Inicio,
Fim)),
 write('Candidate partition:'), write(CandidatePartition),nl,
 write(removeProvisorilyCandidateNewEdge(K_novo,NodeConnected,Maior)),
 write('*******Numero de nos que faltam ser inseridos na arvore: '), write(T), nl,

 buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,

CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
   PrimitiveCycle,CandidatePartition,Partition,T,K),!.

/**************************************************************************
Caso 1: O nó escolhido nao forma ciclo, nao existe candidate partition,
retira da lista de candidatos a nova aresta inserida e escolhe o próximo nó.
**************************************************************************/

buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
PrimitiveCycle,CandidatePartition,Partition,N,K):-
  nl,
  write('******* CASO 1 ********'),nl,
  write('N='), write(N),
  selectNewEdge(K_novo,Maior,NodeConnected,Inicio,Fim),

  ajustaPonteiros(K_novo,F1,F2,F1_novo,F2_novo,NodeConnected),

  ajustaLimites(K_novo,Inicio,Fim,Inicio_novo,Fim_novo,NodeConnected),

  atualizaArvore(Arvore,K_novo,Inicio_novo,Fim_novo,Arvore_nova),
  notExistsPrimitiveCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio, Fim,
PrimitiveCycle_novo, NodeCompletingEdge_novo,WeightCompletingEdge_novo),
  CandidatePartition==[],
  atualizaB(B,K,K_novo,B_novo),

  removeCandidateNewEdge(K_novo,NodeConnected,Maior),
  T is N-1,

% Escrever os resultados:
```

```
  write('  No escolhido='),
write(K_novo),nl,write(selectNewEdge(K_novo,Maior,NodeConnected,Inicio,Fim)),
  write('Ponteiros de F1='),write(F1),write(' e F2='), write(F2),
  write(' para os novos F1='), write(F1_novo), write(' e F2='), write(F2_novo), nl,
  write('Limites da arvore de Inicio='),write(Inicio),write(' e Fim='), write(Fim),
  write(' para os novos Inicio='), write(Inicio_novo), write(' e Fim='), write(Fim_novo), nl,
  write('Atualiza a arvore antiga = '),write(Arvore),write('para a nova arvore = '),
write(Arvore_nova),nl,
  write('NAO EXISTE PRIMITIVE CYCLE!!!'),nl,
  write(notExistsPrimitiveCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio,
Fim, PrimitiveCycle_novo, NodeCompletingEdge_novo,WeightCompletingEdge_novo)),

  write('NAO EXISTE CANDIDATE PARTITION!!!' ),
  write('Atualiza B='),write(B),write(' para o novo B='), write(B_novo),nl,
  write(removeCandidateNewEdge(K_novo,NodeConnected,Maior)),
  write('*******Numero de nos que faltam ser inseridos na arvore: '), write(T), nl,


  buildAndPartition(B_novo,Arvore_nova,F1_novo,F2_novo,Inicio_novo,Fim_novo,

CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
  PrimitiveCycle,CandidatePartition,Partition,T,K_novo),!.

%**********************************************************************
**
%CASO 2: O nó escolhido forma ciclo, nao existe ciclo anterior,
%satisfaz o affinity cycle e marca este como candidate partition
%Neste caso satisfaz a parte 2 de possibility of affinity cycle:
%p(former edge)<= (all the cycles edges)
%**********************************************************************
**
buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
PrimitiveCycle,CandidatePartition,Particao,N,K):-
nl,
write('******* CASO 2 ********'),nl,
%write(buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
%CycleNode,PrimitiveCycle,CandidatePartition,Partition,N,K)),nl,

  write('N='), write(N),
  selectNewEdge(K_novo,Maior,NodeConnected,Inicio,Fim),
  write('  No escolhido='), write(K_novo),nl,
  ajustaPonteiros(K_novo,F1,F2,F1_novo,F2_novo,NodeConnected),

  ajustaLimites(K_novo,Inicio,Fim,Inicio_novo,Fim_novo,NodeConnected),
```

```
  atualizaArvore(Arvore,K_novo,Inicio_novo,Fim_novo,Arvore_nova),
  CycleNode==0,
  existsPrimitiveCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio, Fim,
PrimitiveCycle_novo,
    NodeCompletingEdge_novo,WeightCompletingEdge_novo),

existsFormerEdge(K_novo,B,NodeConnected,Arvore,Inicio,Fim,NodeFormerEdge_novo,
WeightFormerEdge_novo),

possibilityOfCycle(K_novo,PrimitiveCycle_novo,NodeFormerEdge_novo,WeightFormerE
dge_novo),
  removeCandidateNewEdge(K_novo,NodeConnected,Maior),
  T is N,

% Escrever os resultados:
  write('  No escolhido='),
write(K_novo),nl,write(selectNewEdge(K_novo,Maior,NodeConnected,Inicio,Fim)),nl,
  write('Ajuste de Ponteiros de F1='),write(F1),write(' e F2='), write(F2),nl,
  write(' para os novos F1='), write(F1_novo), write(' e F2='), write(F2_novo), nl,
  write('Ajuste de Limites da arvore de Inicio='),write(Inicio),write(' e Fim='),
write(Fim),nl,
  write(' para os novos Inicio='), write(Inicio_novo), write(' e Fim='), write(Fim_novo), nl,
  write('Atualiza a arvore antiga = '),write(Arvore),write('para a nova arvore = '),
write(Arvore_nova),nl,
  write('Não existe ciclo anterior!'), nl,
  write('*******EXISTE PRIMITIVE CYCLE FORMADO PELOS SEGUINTES NÓS:'),
write(PrimitiveCycle_novo),

write('CompletingEdge=('),write(NodeCompletingEdge_novo),write(','),write(K_novo),writ
e(')'),
  write('='), write(WeightCompletingEdge_novo),nl,write('NodeConnected:'),
write(NodeConnected),write('com '),write(Maior),nl,
  write('NodeFormerEdge'),write(NodeFormerEdge_novo),nl,
  write('WeightFormerEdge'),write(WeightFormerEdge_novo),nl,
  write('Possibility of cycle == Yes !!!!!!!!'),nl,
  write(removeCandidateNewEdge(K_novo,NodeConnected,Maior)),
  write('Numero de nos que faltam ser inseridos na arvore: '), write(T),nl,


  buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,K_novo,NodeCompletingEdge_novo,

WeightCompletingEdge_novo,NodeFormerEdge_novo,WeightFormerEdge_novo,[],Primiti
veCycle_novo,
    Partition,T,K_novo),!.
```

```
%**********************************************************************
**
%CASO 3: O nó escolhido forma ciclo, nao existe ciclo anterior,
%satisfaz o affinity cycle e marca este como candidate partition
%Neste caso satisfaz a parte 1 de possibility of affinity cycle:
%not exists former edge
%**********************************************************************
**
buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
PrimitiveCycle,CandidatePartition,Partition,N,K):-
nl,
write('******* CASO 3 ********'),nl,
%write(buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
%CycleNode,PrimitiveCycle,CandidatePartition,Partition,N,K)),nl,

  write('N='), write(N), nl,
  selectNewEdge(K_novo,Maior,NodeConnected,Inicio,Fim),
  ajustaPonteiros(K_novo,F1,F2,F1_novo,F2_novo,NodeConnected),
  ajustaLimites(K_novo,Inicio,Fim,Inicio_novo,Fim_novo,NodeConnected),
  atualizaArvore(Arvore,K_novo,Inicio_novo,Fim_novo,Arvore_nova),
  CycleNode==0,
  existsPrimitiveCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio, Fim,
PrimitiveCycle_novo,
    NodeCompletingEdge_novo,WeightCompletingEdge_novo),


notExistsFormerEdge(K_novo,B,NodeConnected,Arvore,Inicio,Fim,NodeFormerEdge_no
vo, WeightFormerEdge_novo),
  removeCandidateNewEdge(K_novo,NodeConnected,Maior),
  T is N,

% Escrever os resultados:
  write('No escolhido='), write(K_novo),nl,
 % write('Ajuste de Ponteiros de F1='),write(F1),write(' e F2='), write(F2),nl,
 % write(' para o novo F1='), write(F1_novo), write(' e o novo F2='), write(F2_novo), nl,
 % write('Ajuste de Limites da arvore de Inicio='),write(Inicio),write(' e Fim='),
write(Fim),nl,
 % write(' para o novo Inicio='), write(Inicio_novo), write(' e o novo Fim='),
write(Fim_novo), nl,
 % write('Atualiza a arvore antiga = '),write(Arvore),write('para a nova arvore = '),
write(Arvore_nova),nl,
  write('Não existe ciclo anterior!'), nl,
  write('EXISTE PRIMITIVE CYCLE FORMADO PELOS SEGUINTES NÓS:'),
write(PrimitiveCycle_novo),
  write('ATENCAO!!!!!!!!!!!!!!!!!!!!!!!'),nl,
```

```
  write('CompletingEdge=('),write(NodeCompletingEdge_novo),write(','),write(K_novo),
  write('='), write(WeightCompletingEdge_novo),nl,
  write(removeCandidateNewEdge(K_novo,NodeConnected,Maior)),
  write('CandidatePartition='), write(PrimitiveCycle_novo),
  write('Numero de nos que faltam ser inseridos na arvore: '), write(T),nl,

  buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,K_novo, NodeCompletingEdge_novo,
    WeightCompletingEdge_novo, 0,0, [], PrimitiveCycle_novo, Partition,T,K_novo),!.


/**************************************************************************
CASO 4: O nó escolhido forma o ciclo, não existe ciclo anterior, não
satisfaz o Affinity Cycle(p(former edge)> (any cycle edge)),
retira da lista de candidatos a nova aresta(ListCandidateNewEdge)
e escolhe proximo nó.
**************************************************************************/
buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
PrimitiveCycle,CandidatePartition,Partition,N,K):-
nl,
write('******* CASO 4 ********'),nl,

  write('N='), write(N), nl,
  selectNewEdge(K_novo,Maior,NodeConnected,Inicio,Fim),
  ajustaPonteiros(K_novo,F1,F2,F1_novo,F2_novo,NodeConnected),
  ajustaLimites(K_novo,Inicio,Fim,Inicio_novo,Fim_novo,NodeConnected),
  atualizaArvore(Arvore,K_novo,Inicio_novo,Fim_novo,Arvore_nova),
  CycleNode==0,
  existsPrimitiveCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio, Fim,
PrimitiveCycle_novo,
    NodeCompletingEdge_novo,WeightCompletingEdge_novo),

existsFormerEdge(K_novo,B,NodeConnected,Arvore,Inicio,Fim,NodeFormerEdge_novo,
WeightFormerEdge_novo),

notPossibilityOfCycle(K_novo,PrimitiveCycle_novo,NodeFormerEdge_novo,WeightForm
erEdge_novo),

  saveCut(K_novo,NodeConnected,Maior),

  removeCandidateNewEdge(K_novo,NodeConnected,Maior),
  T is N,
% Escrever os resultados:
  write('No escolhido='), write(K_novo),nl,
  write('Ajuste de Ponteiros de F1='),write(F1),write(' e F2='), write(F2),nl,
  write(' para o novo F1='), write(F1_novo), write(' e o novo F2='), write(F2_novo), nl,
```

```
  write('Ajuste de Limites da arvore de Inicio='),write(Inicio),write(' e Fim='),
write(Fim),nl,
  write(' para o novo Inicio='), write(Inicio_novo), write(' e o novo Fim='),
write(Fim_novo), nl,
  write('Atualiza a arvore antiga = '),write(Arvore),write('para a nova arvore = '),
write(Arvore_nova),nl,
  write('Não existe Ciclo anterior'),nl,
  write('EXISTE PRIMITIVE CYCLE FORMADO PELOS SEGUINTES NÓS:'),
write(PrimitiveCycle_novo),
  write('ATENCAO!!!!!!!!!!!!!!!!!!!!!!!!'),nl,
  write('CompletingEdge=('),write(NodeCompletingEdge_novo),write(','),write(K_novo),
  write(','), write(WeightCompletingEdge_novo),nl,
  write('NodeFormerEdge'),write(NodeFormerEdge_novo),nl,
  write('WeightFormerEdge'),write(WeightFormerEdge_novo),nl,
  write('Possibility of cycle == No !!!!!!!!'),nl,
  write(saveCut(K_novo,NodeConnected,Maior)),
  write(removeCandidateNewEdge(K_novo,NodeConnected,Maior)),
  write('Numero de nos que faltam ser inseridos na arvore: '), write(T),nl,

  buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,

CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
  PrimitiveCycle,CandidatePartition,Partition,N,K),!.

/*************************************************************************
CASO 5: O nó escolhido forma ciclo, existe ciclo anterior,
retira provisoriamente a aresta da lista de candidatos a nova
aresta  e escolhe o próximo nó.
*************************************************************************/
buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
PrimitiveCycle,CandidatePartition,Partition,N,K):-
  nl,
  write('******* CASO 5 ********'),nl,
  write('N='), write(N), nl,
  selectNewEdge(K_novo,Maior,NodeConnected,Inicio,Fim),
  ajustaPonteiros(K_novo,F1,F2,F1_novo,F2_novo,NodeConnected),
  ajustaLimites(K_novo,Inicio,Fim,Inicio_novo,Fim_novo,NodeConnected),
  atualizaArvore(Arvore,K_novo,Inicio_novo,Fim_novo,Arvore_nova),
  CycleNode\==0,
  existsCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio, Fim,
PrimitiveCycle_novo,
    NodeCompletingEdge_novo,WeightCompletingEdge_novo),
   removeProvisorilyCandidateNewEdge(K_novo,NodeConnected,Maior),
   T is N,
```

```
% Escrever os resultados:
  write('No escolhido='), write(K_novo),nl,
  write('Ajuste de Ponteiros de F1='),write(F1),write(' e F2='), write(F2),nl,
  write(' para o novo F1='), write(F1_novo), write(' e o novo F2='), write(F2_novo), nl,
  write('Ajuste de Limites da arvore de Inicio='),write(Inicio),write(' e Fim='),
write(Fim),nl,
  write(' para o novo Inicio='), write(Inicio_novo), write(' e o novo Fim='),
write(Fim_novo), nl,
  write('Atualiza a arvore antiga = '),write(Arvore),write('para a nova arvore = '),
write(Arvore_nova),nl,
  write('Existe Ciclo anterior'),nl,
  write('EXISTE PRIMITIVE CYCLE FORMADO PELOS SEGUINTES NÓS:'),
write(PrimitiveCycle_novo),
  write('ATENCAO!!!!!!!!!!!!!!!!!!!!!!!!'),nl,
  write(removeProvisorilyCandidateNewEdge(K_novo,NodeConnected,Maior)),
  write('Numero de nos que faltam ser inseridos na arvore: '), write(T),nl,

  buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,

CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
  PrimitiveCycle,CandidatePartition,Partition,N,K),!.

/*************************************************************************
CASO 6: O nó escolhido não forma ciclo, existe Candidate Partition,
existe a possibilidade de extensão, estende o ciclo,
retira da lista de candidatos a nova aresta e escolhe o próximo nó.
*************************************************************************/
buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
PrimitiveCycle,CandidatePartition,Partition,N,K):-
  nl,
  write('******* CASO 6 ********'),nl,
  write('N='), write(N), nl,
  selectNewEdge(K_novo,Maior,NodeConnected,Inicio,Fim),
  ajustaPonteiros(K_novo,F1,F2,F1_novo,F2_novo,NodeConnected),
  ajustaLimites(K_novo,Inicio,Fim,Inicio_novo,Fim_novo,NodeConnected),
  atualizaArvore(Arvore,K_novo,Inicio_novo,Fim_novo,Arvore_nova),
  CandidatePartition\==0,
  existsPrimitiveCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio, Fim,
PrimitiveCycle_novo,
    NodeCompletingEdge_novo,WeightCompletingEdge_novo),

  %notExistsPrimitiveCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio, Fim,
PrimitiveCycle_novo,
  %                NodeCompletingEdge_novo,WeightCompletingEdge_novo),
```

```
% Escrever os resultados ate o momento:
  write('No escolhido='), write(K_novo),nl,
  write('Ajuste de Ponteiros de F1='),write(F1),write(' e F2='), write(F2),nl,
  write(' para o novo F1='), write(F1_novo), write(' e o novo F2='), write(F2_novo), nl,
  write('Ajuste de Limites da arvore de Inicio='),write(Inicio),write(' e Fim='),
write(Fim),nl,
  write(' para o novo Inicio='), write(Inicio_novo), write(' e o novo Fim='),
write(Fim_novo), nl,
  write('Atualiza a arvore antiga = '),write(Arvore),write('para a nova arvore = '),
write(Arvore_nova),nl,
  write('Existe Candidate Partition'),nl,
  write(notExistsPrimitiveCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio,
Fim, PrimitiveCycle_novo,
                    NodeCompletingEdge_novo,WeightCompletingEdge_novo)),
  write('NAO EXISTE PRIMITIVE CYCLE :'), write(PrimitiveCycle_novo),

%checkCandidateExtension(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio, Fim,
CandidatePartition,
%
NewCycle,NewNodeCompletingEdgeExtension,NewWeightCompletingEdgeExtension),

 %Verifica se eh candidato a extensao do ciclo
  ((
   isCandidateExtension(NodeConnected,CandidatePartition),
   write(isCandidateExtension(NodeConnected,CandidatePartition)),
   existsCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio, Fim, NewCycle,
        NewNodeCompletingEdgeExtension,NewWeightCompletingEdgeExtension),
   write('Ciclo:'),write(NewCycle),
   member(CycleNode,NewCycle)
  );
  (isCandidateExtension(NodeConnected,CandidatePartition),
   write(isCandidateExtension(NodeConnected,CandidatePartition)),
   notExistCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio, Fim, NewCycle,
        NewNodeCompletingEdgeExtension,NewWeightCompletingEdgeExtension),
   write('Não forma ciclo pelo nó que está tentando estender:'),write(NewCycle)
   )),

%checkPossibilityOfExtension()

%Verifica se satisfaz a Possibilidade de extensao do ciclo:
%P(edge being considered or cycle completing edge) >= p(any one of the cycle edges).
%possibilityOfExtension(+Cycle,+K_novo,+NodeConnected,+Weight,+OtherNode)
%NodeConnected: nó do ciclo que K se conectou
%NodeExtend: nó pelo o qual esta se tentando estender
%OtherNode: o outro nó do ciclo (será igual ao CycleNode ou o nó que
%junto com o CycleNode formam o lado que completou o ciclo
```

```
%se foi inserido no CycleNode o OtherNode sera o NodeCompleting
%      NodeConnected
% O  _ 0 _ 0 NodeExtend
%   \|
%     0 OtherNode
/*
K_novo-->NodeExtend
NodeCompleting ---> NodeConnected


*/
  ((CycleNode==NodeConnected,

possibilityOfExtension(CandidatePartition,K_novo,NodeConnected,Maior,NodeCompletin
gEdge),
    write('ë possível estender pelo nó'),write(K_novo),write('conectado
ao'),write(NodeConnected) );
   (NodeCompletingEdge==NodeConnected,
    possibilityOfExtension(CandidatePartition,K_novo,NodeConnected,Maior,CycleNode) ,
    write('ë possível estender pelo nó'),write(K_novo),write('conectado
ao'),write(NodeConnected) )
    ),

 extendCycle(CandidatePartition,K_novo,NewCandidatePartition),



 atualizaB(B,K,K_novo,B_novo),


 removeCandidateNewEdge(K_novo,NodeConnected,Maior),
 T is N-1,

% Escrever os resultados:


 write(extendCycle(CandidatePartition,K_novo,NewCandidatePartition)),nl,
 write('Atualiza B='),write(B),write(' para o novo B='), write(B_novo),nl,
 write(removeCandidateNewEdge(K_novo,NodeConnected,Maior)),
 write('*******Numero de nos que faltam ser inseridos na arvore: '), write(T), nl,



 buildAndPartition(B_novo,Arvore_nova,F1_novo,F2_novo,Inicio_novo,Fim_novo,

CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
```

```
   PrimitiveCycle,NewCandidatePartition,Partition,T,K_novo),!.

%**********************************************************************
%CASO 7.1: O nó escolhido não forma ciclo, existe Candidate Partition,
%não existe Possibilidade de Extensão pois o nó escolhido não está conectado
%a um nó que faz parte do Candidate Partition (NotIsCandidateExtension),
%remove temporariamente a aresta como candidata e escolhe o próximo nó.
%**********************************************************************
buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
PrimitiveCycle,CandidatePartition,Partition,N,K):-
  nl,
  write('******* CASO 7.1 ********'),nl,
  write('N='), write(N), nl,
  selectNewEdge(K_novo,Maior,NodeConnected,Inicio,Fim),
  ajustaPonteiros(K_novo,F1,F2,F1_novo,F2_novo,NodeConnected),
  ajustaLimites(K_novo,Inicio,Fim,Inicio_novo,Fim_novo,NodeConnected),
  atualizaArvore(Arvore,K_novo,Inicio_novo,Fim_novo,Arvore_nova),
  CandidatePartition\==0,
  notExistsPrimitiveCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio, Fim,
PrimitiveCycle_novo,
                NodeCompletingEdge_novo,WeightCompletingEdge_novo),

%Escrever os resultados ate o momento:
  write('No escolhido='), write(K_novo),nl,
  write('Ajuste de Ponteiros de F1='),write(F1),write(' e F2='), write(F2),nl,
  write(' para o novo F1='), write(F1_novo), write(' e o novo F2='), write(F2_novo), nl,
  write('Ajuste de Limites da arvore de Inicio='),write(Inicio),write(' e Fim='),
write(Fim),nl,
  write(' para o novo Inicio='), write(Inicio_novo), write(' e o novo Fim='),
write(Fim_novo), nl,
  write('Atualiza a arvore antiga = '),write(Arvore),write('para a nova arvore = '),
write(Arvore_nova),nl,
  write('Existe Candidate Partition'),nl,
  write(notExistsPrimitiveCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio,
Fim, PrimitiveCycle_novo,
                NodeCompletingEdge_novo,WeightCompletingEdge_novo)),
  write('NAO EXISTE PRIMITIVE CYCLE :'), write(PrimitiveCycle_novo),

%Nao é candidato a extensao do ciclo

      notIsCandidateExtension(NodeConnected,CandidatePartition),
       write('Não é Candidato a extensao:'),
       write(notIsCandidateExtension(NodeConnected,CandidatePartition)),
```

```
 removeProvisorilyCandidateNewEdge(K_novo,NodeConnected,Maior),

 T is N,

%Escrever resultados:
  write(notIsCandidateExtension(NodeConnected,CandidatePartition)),
  write(removeProvisorilyCandidateNewEdge(K_novo,NodeConnected,Maior)),
 write('*******Numero de nos que faltam ser inseridos na arvore: '), write(T), nl,

buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,CycleNode,NodeCompletingEdge,
WeightCompletingEdge,NodeFormerEdge,WeightFormerEdge, PrimitiveCycle,
CandidatePartition ,Partition,T,K),!.

/**************************************************************************
CASO 7.2: O nó escolhido não forma ciclo, existe Candidate Partition,
não existe Possibilidade de Extensão, não existe Former Edge,
salva a partição, retira da lista de candidatos a nova aresta, carrega
novamente as arestas que foram removidas temporariamente e escolhe o
próximo nó.
**************************************************************************/
buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
PrimitiveCycle,CandidatePartition,Partition,N,K):-
  nl,
  write('******* CASO 7.2 ********'),nl,
  write('N='), write(N), nl,
  selectNewEdge(K_novo,Maior,NodeConnected,Inicio,Fim),
  ajustaPonteiros(K_novo,F1,F2,F1_novo,F2_novo,NodeConnected),
  ajustaLimites(K_novo,Inicio,Fim,Inicio_novo,Fim_novo,NodeConnected),
  atualizaArvore(Arvore,K_novo,Inicio_novo,Fim_novo,Arvore_nova),
  CandidatePartition\==0,
  notExistsPrimitiveCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio, Fim,
PrimitiveCycle_novo,
                  NodeCompletingEdge_novo,WeightCompletingEdge_novo),


%Escrever os resultados ate o momento:
  write('No escolhido='), write(K_novo),nl,
  write('Ajuste de Ponteiros de F1='),write(F1),write(' e F2='), write(F2),nl,
  write(' para o novo F1='), write(F1_novo), write(' e o novo F2='), write(F2_novo), nl,
  write('Ajuste de Limites da arvore de Inicio='),write(Inicio),write(' e Fim='),
write(Fim),nl,
  write(' para o novo Inicio='), write(Inicio_novo), write(' e o novo Fim='),
write(Fim_novo), nl,
  write('Atualiza a arvore antiga = '),write(Arvore),write('para a nova arvore = '),
write(Arvore_nova),nl,
```

```
  write('Existe Candidate Partition'),nl,
  write(notExistsPrimitiveCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio,
Fim, PrimitiveCycle_novo,
                    NodeCompletingEdge_novo,WeightCompletingEdge_novo)),
  write('NAO EXISTE PRIMITIVE CYCLE :'), write(PrimitiveCycle_novo),

%Nao é candidato a extensao do ciclo

  (
    ( notIsCandidateExtension(NodeConnected,CandidatePartition),
      write('Não é Candidato a extensao:'),
      write(notIsCandidateExtension(NodeConnected,CandidatePartition)),nl);
    ( isCandidateExtension(NodeConnected,CandidatePartition),
      write('É Candidato a extensao:'),
      write(isCandidateExtension(NodeConnected,CandidatePartition)),nl,
      existsCycle(K, B, NodeConnected, CycleNode, Arvore, Inicio, Fim, NewCycle,
        NodeCompletingEdgeExtension,WeightCompletingEdgeExtension),
      write('Ciclo:'),write(NewCycle),nl,
      \+ member(CycleNode,NewCycle),
      write('CycleNode não pertence ao Ciclo'),nl);
     ( isCandidateExtension(NodeConnected,CandidatePartition),
      write('É Candidato a extensao:'),
      write(isCandidateExtension(NodeConnected,CandidatePartition)),nl,
      existsCycle(K, B, NodeConnected, CycleNode, Arvore, Inicio, Fim, NewCycle,
        NodeCompletingEdgeExtension,WeightCompletingEdgeExtension),
      write('Ciclo:'),write(NewCycle),
      member(CycleNode,NewCycle),
      write('CycleNode pertence ao Ciclo'),nl)
  ),

%Não satisfaz a possibilidade de extensão
    (
     ( CycleNode==NodeConnected,

notPossibilityOfExtension(CandidatePartition,K_novo,NodeConnected,Maior,NodeCompl
etingEdge),
      write('não é possível estender pelo nó'),write(K_novo),write('conectado
ao'),write(NodeConnected) );
     ( NodeCompletingEdge==NodeConnected,

notPossibilityOfExtension(CandidatePartition,K_novo,NodeConnected,Maior,CycleNode)
,
      write('não é possível estender pelo nó'),write(K_novo),write('conectado
ao'),write(NodeConnected) )
    ) ,

%Nao existe Former Edge
```

```
NodeFormerEdge==0,

%Salva particao
 savePartition(CandidatePartition),

 atualizaB(B,K,K_novo,B_novo),

 removeCandidateNewEdge(K_novo,NodeConnected,Maior),
 loadAllRemovedProvisorilyEdge(ListProvisorilyEdges),

%Salva cut
 saveCut(K_novo,NodeConnected,Maior),

 T is N-1,

%Escrever resultados:
 write('Nao ha Former Edge!!!'),nl,
 write(
notExistsFormerEdge(K_novo,B,NodeConnected,Arvore,Inicio,Fim,NodeFormerEdge_no
vo, WeightFormerEdge_novo)), nl,
 write(savePartition(CandidatePartition)),nl,
 write('Atualiza B='),write(B),write(' para o novo B='), write(B_novo),nl,
 write(saveCut(K_novo,NodeConnected,Maior)),

%MUDEI:::
 write(saveCut(K_novo,NodeConnected,Maior)),nl,
 write(removeCandidateNewEdge(K_novo,NodeConnected,Maior)),
 write(loadAllRemovedProvisorilyEdge(ListProvisorilyEdges)),
 write('*******Numero de nos que faltam ser inseridos na arvore: '), write(T), nl,

 buildAndPartition(B_novo,Arvore_nova,F1_novo,F2_novo,Inicio_novo,Fim_novo,
  0,0,0,0,0,[],[],Partition,T,K_novo),!.

%**********************************************************************
%CASO 8: O nó escolhido não forma ciclo, existe Candidate Partition,
%não existe Possibilidade de Extensão, existe Former Edge, troca o CycleNode
%e tenta estender pelo Former Edge. Não há possibilidade de extensão pelo
%Former Edge, salva a partição, retira da lista de candidatos a nova aresta,
%carrega novamente as arestas que foram removidas temporariamente e escolhe
%o próximo nó.
%**********************************************************************
buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
PrimitiveCycle,CandidatePartition,Partition,N,K):-
  nl,
  write('******* CASO 8 ********'),nl,
```

```
  write('N='), write(N), nl,
  selectNewEdge(K_novo,Maior,NodeConnected,Inicio,Fim),
  ajustaPonteiros(K_novo,F1,F2,F1_novo,F2_novo,NodeConnected),
  ajustaLimites(K_novo,Inicio,Fim,Inicio_novo,Fim_novo,NodeConnected),
  atualizaArvore(Arvore,K_novo,Inicio_novo,Fim_novo,Arvore_nova),
  CandidatePartition\==0,
  notExistsPrimitiveCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio, Fim,
PrimitiveCycle_novo,
                    NodeCompletingEdge_novo,WeightCompletingEdge_novo),

 % notExistsPrimitiveCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio, Fim,
PrimitiveCycle_novo),

%Escrever os resultados ate o momento:
  write('No escolhido='), write(K_novo),nl,
  write('Ajuste de Ponteiros de F1='),write(F1),write(' e F2='), write(F2),nl,
  write(' para o novo F1='), write(F1_novo), write(' e o novo F2='), write(F2_novo), nl,
  write('Ajuste de Limites da arvore de Inicio='),write(Inicio),write(' e Fim='),
write(Fim),nl,
  write(' para o novo Inicio='), write(Inicio_novo), write(' e o novo Fim='),
write(Fim_novo), nl,
  write('Atualiza a arvore antiga = '),write(Arvore),write('para a nova arvore = '),
write(Arvore_nova),nl,
  write('Existe Candidate Partition'),nl,
  write(notExistsPrimitiveCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio,
Fim, PrimitiveCycle_novo,
                    NodeCompletingEdge_novo,WeightCompletingEdge_novo)),
  write('NAO EXISTE PRIMITIVE CYCLE :'), write(PrimitiveCycle_novo),

%Nao é candidato a extensao do ciclo

  (
    ( notIsCandidateExtension(NodeConnected,CandidatePartition),
      write('Não é Candidato a extensao:'),
      write(notIsCandidateExtension(NodeConnected,CandidatePartition)),nl);
    ( isCandidateExtension(NodeConnected,CandidatePartition),
      write('É Candidato a extensao:'),
      write(isCandidateExtension(NodeConnected,CandidatePartition)),nl,
      existsCycle(K, B, NodeConnected, CycleNode, Arvore, Inicio, Fim, NewCycle,
        NodeCompletingEdgeExtension,WeightCompletingEdgeExtension),
      write('Ciclo:'),write(NewCycle),nl,
      \+ member(CycleNode,NewCycle),
      write('CycleNode não pertence ao Ciclo'),nl);
     ( isCandidateExtension(NodeConnected,CandidatePartition),
      write('É Candidato a extensao:'),
      write(isCandidateExtension(NodeConnected,CandidatePartition)),nl,
      existsCycle(K, B, NodeConnected, CycleNode, Arvore, Inicio, Fim, NewCycle,
```

```
        NodeCompletingEdgeExtension,WeightCompletingEdgeExtension),
       write('Ciclo:'),write(NewCycle),
       member(CycleNode,NewCycle),
       write('CycleNode pertence ao Ciclo'),nl)
  ),
%Não satisfaz a possibilidade de extensão
   (
    ( CycleNode==NodeConnected,

notPossibilityOfExtension(CandidatePartition,K_novo,NodeConnected,Maior,NodeCompl
etingEdge),
       write('não é possível estender pelo nó'),write(K_novo),write('conectado
ao'),write(NodeConnected) );
     ( NodeCompletingEdge==NodeConnected,

notPossibilityOfExtension(CandidatePartition,K_novo,NodeConnected,Maior,CycleNode)
,
       write('não é possível estender pelo nó'),write(K_novo),write('conectado
ao'),write(NodeConnected) )
   %;
%    (write('Erro!!!'),write('Candidate Partition:'),write(CandidatePartition),!)
   ),

NodeFormerEdge\==0,

%Não é possível estender pelo Former Edge
%NotPossibilityOfExtension ("Completing" será o "NodeFormerEdge_novo" com o
"CycleNode"(antigo)
%ou NodeFormerEdge_novo com CycleNode_novo) verificar....ou K_novo será
NodeFormerEdge
notPossibilityOfExtension(CandidatePartition,NodeFormerEdge,CycleNode,WeightFormer
Edge,NodeCompletingEdge),

%Salva particao
 savePartition(CandidatePartition),

 atualizaB(B,K,K_novo,B_novo),

 removeCandidateNewEdge(K_novo,NodeConnected,Maior),
 loadAllRemovedProvisorilyEdge(ListProvisorilyEdges),

%Salva cut
 saveCut(K_novo,NodeConnected,Maior),
 saveCut(NodeFormerEdge,CycleNode,WeightFormerEdge),

 T is N-1,
```

```
 %Escrever os resultados
 write('Não é possível estender pelo nó do former edge '),write(NodeFormerEdge),nl,
 write('conectado ao cycle node'),write(CycleNode),nl,
 write(savePartition(CandidatePartition)),nl,
 write('Atualiza B='),write(B),write(' para o novo B='), write(B_novo),nl,
 write(saveCut(K_novo,NodeConnected,Maior)),nl,
 write(saveCut(NodeFormerEdge,CycleNode,WeightFormerEdge)),nl,
 write(removeCandidateNewEdge(K_novo,NodeConnected,Maior)),
 write(loadAllRemovedProvisorilyEdge(ListProvisorilyEdges)),

 write('*******Numero de nos que faltam ser inseridos na arvore: '), write(T), nl,

 buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
   0,0,0,0,0,[],[],Partition,T,K_novo),!.

% buildAndPartition(B_novo,Arvore_nova,F1_novo,F2_novo,Inicio_novo,Fim_novo,
 %  0,0,0,0,0,[],[],Partition,T,K_novo),!.

%**************************************************************************
%CASO 9: O nó escolhido não forma ciclo, existe Candidate Partition,
%não existe Possibilidade de Extensão, existe Former Edge, troca o CycleNode
%e tenta estender pelo Former Edge. Há possibilidade de extensão pelo Former
%Edge, estende o ciclo, retira da lista de candidatos a nova aresta e
%escolhe o próximo nó.
%**************************************************************************
buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
PrimitiveCycle,CandidatePartition,Partition,N,K):-
   nl,
   write('******* CASO 9 ********'),nl,
   write('N='), write(N), nl,
   selectNewEdge(K_novo,Maior,NodeConnected,Inicio,Fim),
   ajustaPonteiros(K_novo,F1,F2,F1_novo,F2_novo,NodeConnected),
   ajustaLimites(K_novo,Inicio,Fim,Inicio_novo,Fim_novo,NodeConnected),
   atualizaArvore(Arvore,K_novo,Inicio_novo,Fim_novo,Arvore_nova),
   CandidatePartition\==0,
   notExistsPrimitiveCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio, Fim,
PrimitiveCycle_novo,
                  NodeCompletingEdge_novo,WeightCompletingEdge_novo),

%Escrever os resultados ate o momento:
   write('No escolhido='), write(K_novo),nl,
   write('Ajuste de Ponteiros de F1='),write(F1),write(' e F2='), write(F2),nl,
   write(' para o novo F1='), write(F1_novo), write(' e o novo F2='), write(F2_novo), nl,
   write('Ajuste de Limites da arvore de Inicio='),write(Inicio),write(' e Fim='),
write(Fim),nl,
```

```
  write(' para o novo Inicio='), write(Inicio_novo), write(' e o novo Fim='),
write(Fim_novo), nl,
  write('Atualiza a arvore antiga = '),write(Arvore),write('para a nova arvore = '),
write(Arvore_nova),nl,
  write('Existe Candidate Partition'),nl,
  write(notExistsPrimitiveCycle(K_novo, B, NodeConnected, CycleNode, Arvore, Inicio,
Fim, PrimitiveCycle_novo,
                    NodeCompletingEdge_novo,WeightCompletingEdge_novo)),
  write('NAO EXISTE PRIMITIVE CYCLE :'), write(PrimitiveCycle_novo),

%Nao é candidato a extensao do ciclo

  (
    ( notIsCandidateExtension(NodeConnected,CandidatePartition),
      write('Não é Candidato a extensao:'),
      write(notIsCandidateExtension(NodeConnected,CandidatePartition)),nl);
    ( isCandidateExtension(NodeConnected,CandidatePartition),
      write('É Candidato a extensao:'),
      write(isCandidateExtension(NodeConnected,CandidatePartition)),nl,
      existsCycle(K, B, NodeConnected, CycleNode, Arvore, Inicio, Fim, NewCycle,
       NodeCompletingEdgeExtension,WeightCompletingEdgeExtension),
      write('Ciclo:'),write(NewCycle),nl,
      \+ member(CycleNode,NewCycle),
      write('CycleNode não pertence ao Ciclo'),nl);
     ( isCandidateExtension(NodeConnected,CandidatePartition),
      write('É Candidato a extensao:'),
      write(isCandidateExtension(NodeConnected,CandidatePartition)),nl,
      existsCycle(K, B, NodeConnected, CycleNode, Arvore, Inicio, Fim, NewCycle,
       NodeCompletingEdgeExtension,WeightCompletingEdgeExtension),
      write('Ciclo:'),write(NewCycle),
      member(CycleNode,NewCycle),
      write('CycleNode pertence ao Ciclo'),nl)
  ),

%Não satisfaz a possibilidade de extensão
    (
     ( CycleNode==NodeConnected,

notPossibilityOfExtension(CandidatePartition,K_novo,NodeConnected,Maior,NodeCompl
etingEdge),
       write('não é possível estender pelo nó'),write(K_novo),write('conectado
ao'),write(NodeConnected) );
     ( NodeCompletingEdge==NodeConnected,

notPossibilityOfExtension(CandidatePartition,K_novo,NodeConnected,Maior,CycleNode)
,
```

```
      write('não é possível estender pelo nó'),write(K_novo),write('conectado
ao'),write(NodeConnected) )
    %;
%    (write('Erro!!!'),write('Candidate Partition:'),write(CandidatePartition),!)
    ),

%Indica que existe Former Edge
NodeFormerEdge\==0,

%É possível estender pelo Former Edge
possibilityOfExtension(CandidatePartition,NodeFormerEdge,CycleNode,WeightFormerEd
ge,NodeCompletingEdge),

%Estende o ciclo
extendCycle(CandidatePartition,NodeFormerEdge,NewCandidatePartition),

 atualizaB(B,K,K_novo,B_novo),

 removeCandidateNewEdge(K_novo,NodeConnected,Maior),

 T is N-1,
 achaPeso(NodeCompletingEdge, NodeFormerEdge,NewWeightCompletingEdge),

%Escrever os resultados
 write('É possível estender pelo nó do former edge '),write(NodeFormerEdge),nl,
 write('conectado ao cycle node'),write(CycleNode),nl,
 write(extendCycle(CandidatePartition,K_novo,NewCandidatePartition)),nl,
 write('Atualiza B='),write(B),write(' para o novo B='), write(B_novo),nl,
 removeCandidateNewEdge(K_novo,NodeConnected,Maior),
 write('*******Numero de nos que faltam ser inseridos na arvore: '), write(T), nl,

 buildAndPartition(B_novo,Arvore_nova,F1_novo,F2_novo,Inicio_novo,Fim_novo,
   NodeCompletingEdge, NodeFormerEdge,NewWeightCompletingEdge,0,0,
   PrimitiveCycle,NewCandidatePartition,Partition,T,K_novo),!.

%CASO 10: Nenhuma das opcoes anteriores, descarta o nó escolhido e escolhe outro.
buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
PrimitiveCycle,CandidatePartition,Partition,N,K):-
  nl,
  write('******* CASO 10 ********'),nl,
  write('N='), write(N),
  selectNewEdge(K_novo,Maior,NodeConnected,Inicio,Fim),
  ajustaPonteiros(K_novo,F1,F2,F1_novo,F2_novo,NodeConnected),
  ajustaLimites(K_novo,Inicio,Fim,Inicio_novo,Fim_novo,NodeConnected),
  atualizaArvore(Arvore,K_novo,Inicio_novo,Fim_novo,Arvore_nova),
```

```
%Salva cut
  (NodeConnected==NodeCompletingEdge,saveCut(K_novo,NodeConnected,Maior)),

 removeProvisorilyCandidateNewEdge(K_novo,NodeConnected,Maior),
  T is N,

 %Escrever os resultados
  write('  No escolhido='),
write(K_novo),nl,write(selectNewEdge(K_novo,Maior,NodeConnected,Inicio,Fim)),
  write('Ponteiros de F1='),write(F1),write(' e F2='), write(F2),
  write(' para os novos F1='), write(F1_novo), write(' e F2='), write(F2_novo), nl,
  write('Limites da arvore de Inicio='),write(Inicio),write(' e Fim='), write(Fim),
  write(' para os novos Inicio='), write(Inicio_novo), write(' e Fim='), write(Fim_novo), nl,
  write('Atualiza a arvore antiga = '),write(Arvore),write('para a nova arvore = '),
write(Arvore_nova),nl,

(NodeConnected==NodeCompletingEdge,write(saveCut(K_novo,NodeConnected,Maior)))
,
  write(removeProvisorilyCandidateNewEdge(K_novo,NodeConnected,Maior)),
  write('*******Numero de nos que faltam ser inseridos na arvore: '), write(T), nl,

 buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,

CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
   PrimitiveCycle,CandidatePartition,Partition,T,K),!.

/*%Clausula para parar quando não houver mais nós para serem inseridos(N=0).
buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
PrimitiveCycle,CandidatePartition,Partition,0,K):-
  write('Fim'),
  write(buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,

CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
   PrimitiveCycle,CandidatePartition,Partition,0,K)).
*/

%Clausula para parar quando não houver mais arestas para serem inseridas e
%existir um Candidate Partition. O CandidatePartition deve ser passado a Partition
%ja que como não há mais arestas a serem selecionadas não posso mais tentar estender
%o CandidatePartition.

buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
```

```
CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
PrimitiveCycle,CandidatePartition,Partition,N,K):-
  \+selectNewEdge(K_novo,Maior,NodeConnected,Inicio,Fim),
  CandidatePartition\==[],
  savePartition(CandidatePartition),
  buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,CycleNode,NodeCompletingEdge,
    WeightCompletingEdge,NodeFormerEdge,WeightFormerEdge,
    PrimitiveCycle,[],Partition,N,K).

%Clausula para parar quando não houver mais arestas para serem inseridas.

buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,
CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
PrimitiveCycle,CandidatePartition,Partition,N,K):-
  \+selectNewEdge(K_novo,Maior,NodeConnected,Inicio,Fim),
  write('*&*&*&*&*&*&*&*&*&**&*&*  FIM
*&*&*&*&**&*&*&*&*&*&*'),nl,
  write(buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,

CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightForm
erEdge,
    PrimitiveCycle,CandidatePartition,Partition,N,K)).

%roda(+N,-Part)
%N indica o número de nós que faltam ser inseridos
%Part indica as particoes que foram encontradas
roda(FirstNode,N,ListPartition):- retractall(partition(_)),
          retractall(cut(_,_,_)),
          retractall(candidateNewEdge(_,_,_)),
        retractall(removedProvisorilyEdge(_,_,_)),
          %É preciso para  existsNodeConnectedEnd
          %assert(partition([])),
        %Inicializacao de cut senao da erro quando tenta verificar se tem cut e nao tem
nenhum
          assert(cut(0,0,0)),
          createListNewEdge(ListCandidateNewEdge),
          createCandidateNewEdge(ListCandidateNewEdge),
%         buildAndPartition([1,0,0,0,0,0,0,0,0,0,0],[1],1,0,1,0,0,0,0,0,0,0,[],[],[],N,1),
%         buildAndPartition([9,0,0,0,0,0,0,0,0,0,0],[9],9,0,9,0,0,0,0,0,0,0,[],[],[],N,9),

buildAndPartition([FirstNode,0,0,0,0,0,0,0,0,0,0],[FirstNode],FirstNode,0,FirstNode,0,0,0,0,
0,0,[],[],[],N,FirstNode),
          %retract(partition([])),
          findall(P,partition(P),ListPartition_dup),
          list_to_set(ListPartition_dup, ListPartition),
```

```
        retractall(partition(_)).
%       retractall(cut(_,_,_)),
%       retractall(candidateNewEdge(_,_,_)),
%       retractall(removedProvisorilyEdge(_,_,_)).
```

APÊNDICE B

TABELA DE PASSAGEM DE PARÂMETROS DE BUILDANDPARTITION

Especificação dos Parâmetros

buildAndPartition(B,Strongest,Maxwt,Arvore,F1,F2,Inicio,Fim,Pmim,CycleNode,PrimitiveCycle,CandidatePartition,Partition,N,K)

Parâmetros	Explicação
B	lista com os nós escolhidos em ordem de escolha
Arvore	lista com os nós na ordem da formação da spanning tree
F1	Aponta para a ponta da árvore que é diferente do último nó escolhido
F2	Aponta para a ponta da árvore que é igual ao último nó escolhido
Inicio	Aponta para o ínicio da spanning tree
Fim	Aponta para o final da spanning tree
CycleNode	Nó escolhido que formou um ciclo(nó do cycle completing edge que foi selecionado anteriormente)
NodeCompletingEdge	Nó que juntamente com o CycleNode forma o cycle completing edge(lado selecionado que completou o ciclo)
WeightCompletingEdge	Peso do lado que formou o ciclo(peso do cycle completing edge)
NodeFormerEdge	Nó que juntamente com o CycleNode forma o FormerEdge
WeightFormerEdge	Peso do FormerEdge
PrimitiveCycle	Identifica o Ciclo deAfinidades
CandidatePartition	Identifica a Partição Candidata
Partition	Identifica a Partição
N	Número de nós que faltam ser inseridos
K	Nó escolhido no loop anterior

%CASO -1: Verifica se ainda há nós a serem conectados a um end que faz parte de
%um candidate partition.
%Se nao houver, marca o candidate partition como particao e escolhe o proximo no
%(que com certeza sera conectado ao outro end da arvore).
%Descarta o nó escolhido e escolhe outro se este fizer parte de alguma particao.
% End1 and NodeCompletingEdge
% O _0 --------0 Node (exist???)
% \| WheightCompletingEdge
% 0 CycleNode
% |
% 0
% |
% 0 End2
Obs.: Não é preciso escolher novo nó nem remover aresta da lista de candidatos.

%CASO 0: Descarta o nó escolhido e escolhe outro se este fizer parte de alguma particao.
%ou se fizer parte de uma particao candidata.
Remover aresta da lista de candidatos TEMPORARIAMENTE.

Caso 1: O nó escolhido nao forma ciclo, nao existe candidate partition, retira da lista de candidatos a nova aresta inserida e escolhe o próximo nó.
Chamada: buildAndPartition(B,Strongest,Maxwt,Arvore,F1,F2,Inicio,Fim,
CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightFormerEdge,
PrimitiveCycle,CandidatePartition,Partition,N,K):-

Ação	Predicado(s) utilizado(s)	Nova variável encontrada
1. Escolhe nó a ser inserido	SelectNewEdge	NodeSelected,Weight, NodeConnected
2. Ajusta F1 e F2	AjustaPonteiros	F1_novo, F2_novo
3. Ajusta limites da árvore	AjustaLimites	Inicio_novo, Fim_novo
4. Atualiza a árvore	AtualizaArvore	Arvore_nova
5. Verifica se NÃO existe Primitive Cycle	NotExistsPrimitiveCycle	PrimitiveCycle_novo(neste caso continuará vazia esta lista)
6. Não existe partição candidata	CandidatePartition==[]	
7. Atualiza B	AtualizaB	B_novo
8. Retira a aresta de ListCandidateNewEdge	RemoveCandidateNewEdge	
9. Decrementa o número de nós que faltam ser inseridos	T is N-1	

Chamada recursiva:
buildAndPartition(B_novo,Strongest_novo,Maxwt_novo,Arvore_nova,F1_novo,F2_novo,Inicio_novo,Fim_novo,CycleNode,No

deCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightFormerEdge, PrimitiveCycle,CandidatePartition,Partition,T,K_novo),!.

Caso 2: O nó escolhido forma ciclo, nao existe ciclo anterior, satisfaz o affinity cycle e marca este como candidate partition. Neste caso satisfaz a parte 2 de possibility of affinity cycle: p(former edge)<= (all the cycles edges)
Chamada: buildAndPartition(B,Strongest,Maxwt,Arvore,F1,F2,Inicio,Fim, CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightFormerEdge, PrimitiveCycle,CandidatePartition,Partition,N,K)

Ação	Predicado(s) utilizado(s)	Nova variável encontrada
10. Escolhe nó a ser inserido	SelectNewEdge	NodeSelected,Weight, NodeConnected
1. Ajusta F1 e F2	AjustaPonteiros	F1_novo, F2_novo
2. Ajusta limites da árvore	AjustaLimites	Inicio_novo, Fim_novo
3. Atualiza a árvore	AtualizaArvore	Arvore_nova
4. Não existe Cycle Node	CycleNode == 0	
5. Verifica se existe Primitive Cycle	ExistsPrimitiveCycle	PrimitiveCycle_novo(lista de nós com o primitive cycle), NodeCompletingEdge_novo, WeightCompletingEdge_novo
6. Verifica se existe Former Edge	ExistsFormerEdge	NodeFormerEdge_novo (nó que junto com o cycle node forma o Former Edge), WeightFormerEdge_novo(peso do Former Edge)
7. Verifica possibilidade de ter um Affinity Cycle e consequentemente uma Candidate Partition	PossibilityOfCycle	True or false
11. Retira a aresta de ListCandidateNewEdge	RemoveCandidateNewEdge	
8. Não modifica o número de nós a serem inseridos	T is N	Não contabiliza como um nó incluido já que o nó selecionado formou um ciclo

Chamada recursiva: buildAndPartition(B,Strongest,Maxwt_novo,Arvore,F1,F2,Inicio,Fim, K_novo, NodeCompletingEdge_novo, WeightCompletingEdge_novo, NodeFormerEdge_novo, WeightFormerEdge_novo,[],PrimitiveCycle_novo,Partition,T,K_novo).
Obs.: Não são modificados quando forma um primitive cycle: B, Strongest, Arvore, F1,F2,Inicio,Fim,N,
São modificados: maxwt_novo(é retirado k_novo de Maxwt para não ser escolhido novamente), cycleNode para K_novo, PrimitiveCycle para [], CandidatePartition paraPrimitiveCycle_novo e tambem NodeCompletingEdge_novo, WeightCompletingEdge_novo, NodeFormerEdge_novo, WeightFormerEdge_novo.

Caso 3: O nó escolhido forma ciclo, nao existe ciclo anterior, satisfaz o affinity cycle e marca este como candidate partition. Neste caso satisfaz a parte 1 de possibility of affinity cycle: not exists former edge
Chamada: buildAndPartition(B,Strongest,Maxwt,Arvore,F1,F2,Inicio,Fim, CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightFormerEdge, PrimitiveCycle,CandidatePartition,Partition,N,K):-

Ação	Predicado(s) utilizado(s)	Nova variável encontrada
12. Escolhe nó a ser inserido	SelectNewEdge	NodeSelected,Weight, NodeConnected
1. Ajusta F1 e F2	AjustaPonteiros	F1_novo, F2_novo
2. Ajusta limites da árvore	AjustaLimites	Inicio_novo, Fim_novo
3. Atualiza a árvore	AtualizaArvore	Arvore_nova
4. Não existe Cycle Node	CycleNode = = 0	
5. Verifica se existe Primitive Cycle	ExistsPrimitiveCycle	PrimitiveCycle_novo(lista de nós com o primitive cycle), NodeCompletingEdge_novo, WeightCompletingEdge_novo
6. Não existe former edge	NotExistsFormerEdge	NodeFormerEdge_novo (nó que junto com o cycle node forma o Former Edge), WeightFormerEdge_novo(peso do Former Edge)
13. Retira a aresta de ListCandidateNewEdge	RemoveCandidateNewEdge	
7. Não modifica o número de nós a serem inseridos	T is N	Não contabiliza como um nó incluido já que o nó selecionado formou um ciclo

Chamada recursiva:
buildAndPartition(B,Strongest,Maxwt_novo,Arvore,F1,F2,Inicio,Fim,K_novo, NodeCompletingEdge_novo, WeightCompletingEdge_novo, 0,0, [], PrimitiveCycle_novo, Partition,T,K_novo).

Obs.: Não são modificados quando forma um primitive cycle: B, Strongest, Arvore, F1,F2,Inicio,Fim,N,
São modificados: maxwt_novo(é retirado k_novo de Maxwt para não ser escolhido novamente), cycleNode para K_novo, NodeCompletingEdge_novo, WeightCompletingEdge_novo. Como não há FormerEdge então NodeFormerEdge_novo será 0 e WeightFormerEdge_novo também será 0.E também PrimitiveCycle para [], CandidatePartition paraPrimitiveCycle_novo.

Caso 4: O nó escolhido forma ciclo, nao existe ciclo anterior, não satisfaz o affinity cycle(p(former edge)> (any cycle edge)), retira da lista de candidatos a nova aresta(ListCandidateNewEdge) e escolhe proximo nó.

Chamada:

buildAndPartition(B,Strongest,Maxwt,Arvore,F1,F2,Inicio,Fim,CycleNode,NodeCompletingEdge, WeightCompletingEdge,NodeFormerEdge,WeightFormerEdge,PrimitiveCycle,CandidatePartition,Partition,N,K):-

Ação	Predicado(s) utilizado(s)	Nova variável encontrada
14. Escolhe nó a ser inserido	selectNewEdge	NodeSelected,Weight, NodeConnected
1. Ajusta F1 e F2	ajustaPonteiros	F1_novo, F2_novo
2. Ajusta limites da árvore	ajustaLimites	Inicio_novo, Fim_novo
3. Atualiza a árvore	atualizaArvore	Arvore_nova
4. Não existe Cycle Node	CycleNode = = 0	
5. Verifica se existe Primitive Cycle	existsPrimitiveCycle	PrimitiveCycle_novo(lista de nós com o primitive cycle), NodeCompletingEdge_novo, WeightCompletingEdge_novo
6. Verifica se existe Former Edge	existsFormerEdge	NodeFormerEdge (nó que junto com o cycle node forma o Former Edge), WeightFormerEdge(peso do Former Edge)
7. Verifica que não há a possibilidade de ter um Affinity Cycle (p(formerEdge)	notPossibilityOfCycle	True or false
15. Retira a aresta de ListCandidateNewEdge	RemoveCandidateNewEdge	
8. Não modifica o número de nós a serem inseridos	T is N	Não contabiliza como um nó incluido já que o nó selecionado formou um ciclo

Chamada recursiva:

buildAndPartition(B,Strongest,Maxwt_novo,Arvore,F1,F2,Inicio,Fim,CycleNode, NodeCompletingEdge, WeightCompletingEdge,NodeFormerEdge,WeightFormerEdge,PrimitiveCycle, CandidatePartition,Partition,N,K).

Caso 5: O nó escolhido forma ciclo, existe ciclo anterior, retira provisoriamente a aresta da lista de candidatos a nova aresta e escolhe o próximo nó.

Chamada:

buildAndPartition(B,Strongest,Maxwt,Arvore,F1,F2,Inicio,Fim,CycleNode,NodeCompletingEdge, WeightCompletingEdge,NodeFormerEdge,WeightFormerEdge,PrimitiveCycle,CandidatePartition,Partition,N,K):-

Ação	Predicado(s) utilizado(s)	Nova variável encontrada
16. Escolhe nó a ser inserido	selectNewEdge	NodeSelected,Weight, NodeConnected
1. Ajusta F1 e F2	ajustaPonteiros	F1_novo, F2_novo
2. Ajusta limites da árvore	ajustaLimites	Inicio_novo, Fim_novo
3. Atualiza a árvore	atualizaArvore	Arvore_nova
4. Existe Cycle Node	CycleNode \== 0	
5. Verifica se existe Primitive Cycle	existsPrimitiveCycle	PrimitiveCycle_novo(lista de nós com o primitive cycle), NodeCompletingEdge_novo, WeightCompletingEdge_novo
6. Remove temporariamente a candidata a nova aresta (carrego novamente quando é salva a partição)	removeProvisorilyCandidateNewEdge	
7. Não modifica o número de nós a serem inseridos	T is N	Não contabiliza como um nó incluido já que o nó selecionado formou um ciclo

Chamada recursiva:

buildAndPartition(B,Strongest,Maxwt_novo,Arvore,F1,F2,Inicio,Fim,CycleNode, NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightFormerEdge,PrimitiveCycle, CandidatePartition,Partition,N,K).

Caso 6: O nó escolhido não forma ciclo, existe Candidate Partition, existe Possibilidade de Extensão, estende o ciclo,retira da lista de candidatos a nova aresta e escolhe o próximo nó.
Chamada:
buildAndPartition(B,Strongest,Maxwt,Arvore,F1,F2,Inicio,Fim,CycleNode,NodeCompletingEdge,
WeightCompletingEdge,NodeFormerEdge,WeightFormerEdge,PrimitiveCycle,CandidatePartition,Partition,N,K):-

Ação	Predicado(s) utilizado(s)	Nova variável encontrada
17. Escolhe nó a ser inserido	selectNewEdge	NodeSelected,Weight, NodeConnected
1. Ajusta F1 e F2	ajustaPonteiros	F1_novo, F2_novo
2. Ajusta limites da árvore	ajustaLimites	Inicio_novo, Fim_novo
3. Atualiza a árvore	atualizaArvore	Arvore_nova
4. Existe Candidate Partition	CandidatePartition \== 0	
5. Verifica se não existe Primitive Cycle	NotExistsPrimitiveCycle	PrimitiveCycle_novo(neste caso continuará vazia esta lista)
6. Verifica se o nó escolhido é candidato a extensão do Candidate Partition(se o nó escolhido foi conectado a um nó que pertence ao Candidate Partition e este nó forma um ciclo que contém o Cycle Node existente)	IsCandidateExtension	True or false
7. Verifica o "Possibility of Extension", ou seja, p(edge being considered or cycle completing edge) >= p(any one of cycle edges)	PossibilityOfExtension existCycle \+member(K_novo,Cycle),	True or false Cycle
8. Estende o ciclo	ExtendCycle	NewCandidatePartition
18. Retira a aresta de ListCandidateNewEdge	RemoveCandidateNewEdge	
9. Decrementa o número de nós que faltam ser inseridos	T is N-1	

Chamada recursiva:
buildAndPartition(B_novo,Strongest_novo,Maxwt_novo,Arvore_nova,F1_novo,F2_novo,Inicio_novo,Fim_novo,CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightFormerEdge,
PrimitiveCycle, NewCandidatePartition ,Partition,T,K_novo),!.

Caso 7.1: O nó escolhido não forma ciclo, existe Candidate Partition, não existe Possibilidade de Extensão pois o nó escolhido não está conectado a um nó que faz parte do Candidate Partition (NotIsCandidateExtension), remove temporariamente a aresta como candidata e escolhe o próximo nó.

Chamada:

buildAndPartition(B,Strongest,Maxwt,Arvore,F1,F2,Inicio,Fim,CycleNode,NodeCompletingEdge,
WeightCompletingEdge,NodeFormerEdge,WeightFormerEdge,PrimitiveCycle,CandidatePartition,Partition,N,K):-

Ação	Predicado(s) utilizado(s)	Nova variável encontrada
19. Escolhe nó a ser inserido	SelectNewEdge	NodeSelected,Weight, NodeConnected
1. Ajusta F1 e F2	AjustaPonteiros	F1_novo, F2_novo
2. Ajusta limites da árvore	AjustaLimites	Inicio_novo, Fim_novo
3. Atualiza a árvore	AtualizaArvore	Arvore_nova
4. Existe Candidate Partition	CandidatePartition \== 0	
5. Verifica se não existe Primitive Cycle	NotExistsPrimitiveCycle	PrimitiveCycle_novo(neste caso continuará vazia esta lista)
6. Verifica se não há "Possibility of Extension"	NotIsCandidateExtension	True or false
8. Remove temporariamente a candidata a nova aresta (carrego novamente quando é salva a partição)	removeProvisorilyCandidateNewEdge	
7. Não altera o número de nós que faltam ser inseridos	T is N	

Chamada recursiva:

buildAndPartition(B,Arvore,F1,F2,Inicio,Fim,CycleNode,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,WeightFormerEdge, PrimitiveCycle, CandidatePartition,Partition,T,K),!.

Obs.: CycleNode será 0 (o que indica que não haverá CycleNode) pois o ciclo existente passará a ser uma partição. A CandidatePartition será [ ] (lista vazia) pois a partição candidata foi estendida e virou uma partição (savePartition dá um assert(partition(CandidatePartition))).

Caso 7.2: O nó escolhido não forma ciclo, existe Candidate Partition, não existe Possibilidade de Extensão, não existe Former Edge, salva a partição, retira da lista de candidatos a nova aresta, carrega novamente as arestas que foram removidas temporariamente e escolhe o próximo nó.

Chamada:

buildAndPartition(B,Strongest,Maxwt,Arvore,F1,F2,Inicio,Fim,CycleNode,NodeCompletingEdge,
WeightCompletingEdge,NodeFormerEdge,WeightFormerEdge,PrimitiveCycle,CandidatePartition,Partition,N,K):-

Ação	Predicado(s) utilizado(s)	Nova variável encontrada
20. Escolhe nó a ser inserido	SelectNewEdge	NodeSelected,Weight, NodeConnected
8. Ajusta F1 e F2	AjustaPonteiros	F1_novo, F2_novo
9. Ajusta limites da árvore	AjustaLimites	Inicio_novo, Fim_novo
10. Atualiza a árvore	AtualizaArvore	Arvore_nova
11. Existe Candidate Partition	CandidatePartition \== 0	
12. Verifica se não existe Primitive Cycle	NotExistsPrimitiveCycle	PrimitiveCycle_novo(neste caso continuará vazia esta lista)
13. Verifica se não há "Possibility of Extension", ou seja, p(edge being considered and cycle completing edge) < p(all of cycle edges)	isCandidateExtension and notPossibilityOfExtension	True or false
8. Não existe former edge	NotExistsFormerEdge	NodeFormerEdge_novo (nó que junto com o cycle node forma o Former Edge), WeightFormerEdge_novo(peso do Former Edge)
14. Salva partição	SavePartition	Dá um assert(partition(CandidatePartition))
15. Atualiza B	AtualizaB	B_novo
21. Retira a aresta de ListCandidateNewEdge	RemoveCandidateNewEdge	
16. Carrega novamente as arestas que foram removidas temporariamente	loadAllRemovedProvisorilyEdge	
17. Salva cut entre K_novo e NoOndeConectou	SaveCut	saveCut(K_novo,NoConectado,Maior)
18. Decrementa o número de	T is N-1	

nós que faltam ser inseridos		

Chamada recursiva:
buildAndPartition(B_novo,Strongest_novo,Maxwt_novo,Arvore_nova,F1_novo,F2_novo,Inicio_novo,Fim_novo,0,0,0,0,0,[],[],Partition,T,K_novo),!.

Obs.: CycleNode será 0 (o que indica que não haverá CycleNode) pois o ciclo existente passará a ser uma partição. A CandidatePartition será [ ] (lista vazia) pois a partição candidata foi estendida e virou uma partição (savePartition dá um assert(partition(CandidatePartition))).

Caso 8: O nó escolhido não forma ciclo, existe Candidate Partition, não existe Possibilidade de Extensão, existe Former Edge, troca o CycleNode e tenta estender pelo Former Edge. Não há possibilidade de extensão pelo Former Edge, salva a partição, retira da lista de candidatos a nova aresta, carrega novamente as arestas que foram removidas temporariamente e escolhe o próximo nó.
Chamada:
buildAndPartition(B,Strongest,Maxwt,Arvore,F1,F2,Inicio,Fim,CycleNode,NodeCompletingEdge,
WeightCompletingEdge,NodeFormerEdge,WeightFormerEdge,PrimitiveCycle,CandidatePartition,Partition,N,K):-

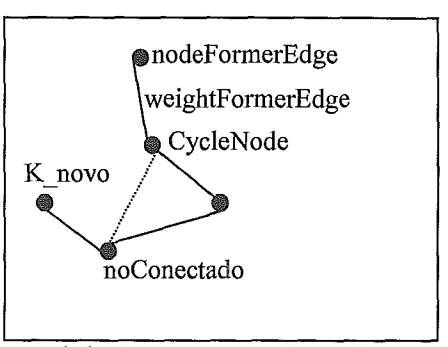
Ação	Predicado(s) utilizado(s)	Nova variável encontrada
22. Escolhe nó a ser inserido	selectNewEdge	NodeSelected,Weight, NodeConnected
1. Ajusta F1 e F2	AjustaPonteiros	F1_novo, F2_novo
2. Ajusta limites da árvore	AjustaLimites	Inicio_novo, Fim_novo
3. Atualiza a árvore	AtualizaArvore	Arvore_nova
4. Existe Candidate Partition	CandidatePartition \== 0	
5. Verifica se não existe Primitive Cycle	NotExistsPrimitiveCycle	PrimitiveCycle_novo(neste caso continuará vazia esta lista)
6. Verifica se não há "Possibility of Extension", ou seja, p(edge being considered and cycle completing edge) < p(all of cycle edges)	notIsCandidateExtension OR (isCandidateExtension and notPossibilityOfExtension)	True or false
7. Verifica se existe Former Edge	ExistsFormerEdge	NodeFormerEdge_novo (nó que junto com o cycle node forma o Former Edge), WeightFormerEdge_novo(peso do Former Edge)
9. Não é possível estender pelo Former Edge	NotPossibilityOfExtension ("Completing" será o	

	"NodeFormerEdge_novo" com o "CycleNode"(antigo) ou NodeFormerEdge_novo com CycleNode_novo)	
8. Salva partição	SavePartition	NewPartition
9. Atualiza B	AtualizaB	B_novo
23. Retira a aresta de ListCandidateNewEdge	RemoveCandidateNewEdge	
19. Carrega novamente as arestas que foram removidas temporariamente	LoadAllRemovedProvisorilyEdge	
20. Salva cut entre K_novo e NoOndeConectou e entre o NodeFormerEdge e CycleNode	SaveCut	saveCut(K_novo,NoConectado,Maior), saveCut(NodeFormerEdge,CycleNode,WeightFormerEdge),
10. Decrementa o número de nós que faltam ser inseridos	T is N-1	

Chamada recursiva:
buildAndPartition(B_novo,Strongest_novo,Maxwt_novo,Arvore_nova,F1_novo,F2_novo,Inicio_novo,Fim_novo,0,NodeCompletingEdge,WeightCompletingEdge,NodeFormerEdge,
WeightFormerEdge,
PrimitiveCycle,[],NewPartition,T,K_novo),!.

Obs. 1: CycleNode será 0 (o que indica que não haverá CycleNode) pois o ciclo existente passará a ser uma partição. A CandidatePartition será [ ] (lista vazia) pois a partição candidata foi estendida e virou uma partição (NewPartition).

Obs. 2: Reutilização do predicado PossibilityOfExtension para verificar a possibilidade de extensão pelo Former Edge. Considere que estamos na seguinte situação:



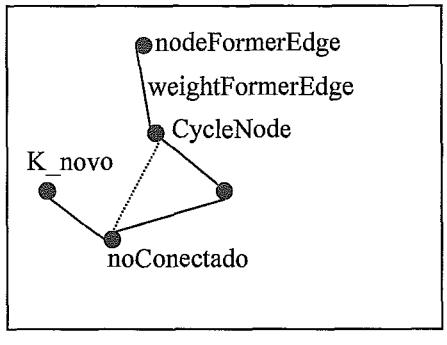
PossibilityOfExtension de um ciclo (tentar estender pelo K_novo):

%Verifica se satisfaz a Possibilidade de extensao do ciclo:
%P(edge being considered or cycle completing edge) >= p(any one of the cycle edges).

```
%possibilityOfExtension(+Cycle,+K_novo,+NodeConnected,+Weight,+OtherNode)
%NodeConnected: nó do ciclo que K se conectou
%NodeExtend: nó pelo o qual esta se tentando estender
%OtherNode: o outro nó do ciclo (será igual ao CycleNode ou o nó que
%junto com o CycleNode formam o lado que completou o ciclo
%se foi inserido no CycleNode o OtherNode sera o NodeCompleting
%      NodeConnected
% O  _ 0 _ 0 NodeExtend
%   \|
%     0 OtherNode
/*
K_novo-->NodeExtend
NodeCompleting ---> NodeConnected


*/
  ((CycleNode==NoConectado,

possibilityOfExtension(CandidatePartition,K_novo,NoConectado,Maior,NodeCompletingE
dge),
    write('ë possível estender pelo nó'),write(K_novo),write('conectado
ao'),write(NoConectado) );
   (NodeCompletingEdge==NoConectado,
    possibilityOfExtension(CandidatePartition,K_novo,NoConectado,Maior,CycleNode) ,
    write('ë possível estender pelo nó'),write(K_novo),write('conectado
ao'),write(NoConectado) );
    (write('Erro!!!'),write('Candidate Partition:'),write(CandidatePartition),!)
    ),
```



PossibilityOfExtension de um ciclo (tentar estender former edge):

```
%Verifica se satisfaz a Possibilidade de extensao do ciclo:
%P(edge being considered or cycle completing edge) >= p(any one of the cycle edges).
%possibilityOfExtension(+Cycle,+K_novo,+NodeConnected,+Weight,+OtherNode)
%NodeConnected: nó do ciclo que K se conectou
%NodeExtend: nó pelo o qual esta se tentando estender
%OtherNode: o outro nó do ciclo (será igual ao CycleNode ou o nó que
%junto com o CycleNode formam o lado que completou o ciclo
```

```
%se foi inserido no CycleNode o OtherNode sera o NodeCompleting
%      NodeConnected
% O _ 0 _ 0 NodeExtend
%  \|
%     0 OtherNode
/*
K_novo-->NodeExtend
NodeCompleting ---> NodeConnected


*/

notPossibilityOfExtension(CandidatePartition,NodeFormerEdge,CycleNode,WeightFormer
Edge,NodeCompletingEdge),
write('Não é possível estender pelo nó do former edge '),write(NodeFormerEdge),nl,
write('conectado ao cycle node'),write(CycleNode),nl,
```

Caso 9: O nó escolhido não forma ciclo, existe Candidate Partition, não existe Possibilidade de Extensão, existe Former Edge, troca o CycleNode e tenta estender pelo Former Edge. Há possibilidade de extensão pelo Former Edge, estende o ciclo, retira da lista de candidatos a nova aresta e escolhe o próximo nó.

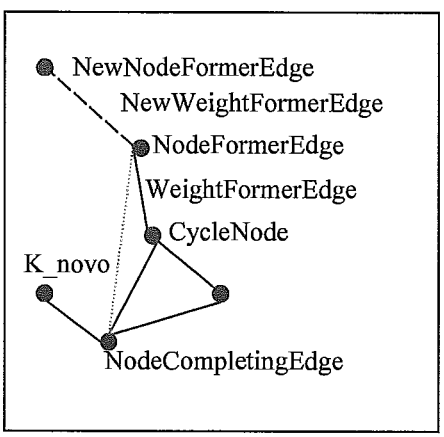
Chamada:

buildAndPartition(B,Strongest,Maxwt,Arvore,F1,F2,Inicio,Fim,CycleNode,NodeCompleti ngEdge,
WeightCompletingEdge,NodeFormerEdge,WeightFormerEdge,PrimitiveCycle,CandidateP artition,Partition,N,K):-

Ação	Predicado(s) utilizado(s)	Nova variável encontrada
24. Escolhe nó a ser inserido	selectNewEdge	NodeSelected,Weight, NodeConnected
11. Ajusta F1 e F2	ajustaPonteiros	F1_novo, F2_novo
12. Ajusta limites da árvore	ajustaLimites	Inicio_novo, Fim_novo
13. Atualiza a árvore	atualizaArvore	Arvore_nova
14. Existe Candidate Partition	CandidatePartition \== 0	
15. Verifica se não existe Primitive Cycle	NotExistsPrimitiveCycle	PrimitiveCycle_novo(neste caso continuará vazia esta lista)
16. Verifica se não há "Possibility of Extension", ou seja, p(edge being considered and cycle completing edge) < p(all of cycle edges)	notIsCandidateExtension OR (isCandidateExtension and notPossibilityOfExtension)	True or false
17. Verifica se existe Former Edge	existsFormerEdge	NodeFormerEdge_novo (nó que junto com o cycle node

		forma o Former Edge), WeightFormerEdge_novo(peso do Former Edge)
10. É possível estender pelo Former Edge	PossibilityOfExtension (by NodeFormerEdge)	
10. Estende o ciclo	ExtendCycle	NewCandidatePartition
18. Acha NewWeightCompletingEdge	achaPeso	NewWeightCompletingEdge
19. Atualiza B	AtualizaB	B_novo
25. Retira a aresta de ListCandidateNewEdge	RemoveCandidateNewEdge	
20. Decrementa o número de nós que faltam ser inseridos	T is N-1	

Chamada recursiva:
buildAndPartition(B_novo,Strongest_novo,Maxwt_novo,Arvore_nova,F1_novo,F2_novo,Inicio_novo,Fim_novo, NodeCompletingEdge,
NodeFormerEdge,NewWeightCompletingEdge, 0,0,
PrimitiveCycle, NewCandidatePartition,Partition,T,K_novo),!.



Obs.: CycleNode novo será o NodeCompletingEdge já que há a troca de CycleNode (teoricamente é antes mas para nós só é preciso trocar se eu conseguir estender pelo Former Edge), o NodeCompletingEdge novo será o NodeFormerEdge, o WeightCompletingEdge novo será o o peso existente entre os dois mencionados anteriormente.

ATRIBUTOS DE ENTRADA:
/**********

Attribute Affinity - AA

aa(no_1, no_2, w) w is an integer variable for the weight of the edge do no 1 para o 2.
**********/

aa(1,2,25).	aa(1,3,25).	aa(1,4,0).	aa(1,5,75).	aa(1,6,0).
aa(1,7,50).	aa(1,8,25).	aa(1,9,25).	aa(1,10,0).	aa(2,3,75).
aa(2,4,0).	aa(2,5,25).	aa(2,6,0).	aa(2,7,60).	aa(2,8,110).

aa(2,9,75).	aa(2,10,0).	aa(3,4,15).	aa(3,5,25).	aa(3,6,15).
aa(3,7,25).	aa(3,8,75).	aa(3,9,115).	aa(3,10,15).	aa(4,5,0).
aa(4,6,40).	aa(4,7,0).	aa(4,8,0).	aa(4,9,15).	aa(4,10,40).
aa(5,6,0).	aa(5,7,50).	aa(5,8,25).	aa(5,9,25).	aa(5,10,0).
aa(6,7,0).	aa(6,8,0).	aa(6,9,15).	aa(6,10,40).	aa(7,8,60).
aa(7,9,25).	aa(7,10,0).	aa(8,9,75).	aa(8,10,0).	aa(9,10,15).

Mudanças importantes para retirar as listas maxwt e strongest encontradas em (NAVATHE, 1989) e que não funcionava direito:
%Insercao dos predicados na inicializacao:
% createListNewEdge(ListCandidateNewEdge),
% createCandidateNewEdge(ListCandidateNewEdge),
%Inclusao em builAndPartition do argumento ListCandidateNewEdge
%Verificar qdo devera ser computada a NewListCandidateNewEdge em cada caso....

MODIFICAÇÃO:
O programa chega a conectar todos os nós porém deve ser verificado o critério de parada. Não pode ser o N=0 pois mesmo com N=0 ainda é preciso rodar para achar as arestas que formam ciclos entre os nós q já estao conectados.

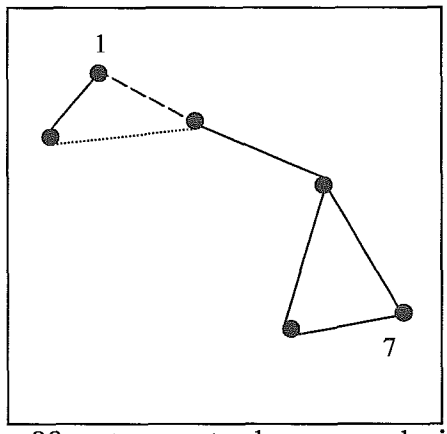
Correção do problema acima: Foi colocada uma clausula para parar quando não houver mais arestas para serem inseridas.

Inicialmente a lista ListCandidateNewEdge recebe todas as arestas que podem ser inseridas;

Se após ser escolhida uma aresta for encontrada a situação descrita num dos casos abaixo relacionados, esta aresta deve ser removida da lista ListCandidateNewEdge através do predicado RemoveCandidateNewEdge e esta aresta não poderá mais ser escolhida novamente durante a construção do grafo. Casos: 1,2,3,4,6,7.2,8,9
No caso de 7.2 e 8, onde é salva uma partição, todas as arestas removidas temporariamente deverão ser novamente incorporadas à lista de candidatas através do predicado loadAllRemovedProvisorilyEdge.

Se após ser escolhida uma aresta for encontrada a situação descrita num dos casos abaixo relacionados, esta aresta deve ser removida da lista ListCandidateNewEdge temporariamente através do predicado RemoveProvisorilyCandidateNewEdge. Esta aresta poderá ser escolhida novamente durante a construção do grafo, porém somente após a definição da partição que deverá ser feita para a CandidatePartition em questão.

Ou seja, se há um CandidatePartition, primeiro tentaremos estender este ciclo para depois comecarmos a trabalhar com outro ciclo. Ou a aresta escolhida pode ser um subciclo do ciclo já formado ou pode, por exemplo, acontecer a seguinte situação(ver Figura abaixo): existe um CandidatePartition formado pelos nós [5,7,9] e ser escolhida a aresta (6,2). Pelo



algoritmo descrito em Navathe 89 outra aresta deve ser selecionada, portanto devemos retira-la da lista de candidatos a nova aresta.Casos: 5,7.1

APÊNDICE C

ARQUIVOS PARA A REVISÃO DE TEORIA

Neste apêndice são apresentados os principais arquivos para a revisão de teoria. A teoria revisada neste arquivo, somente para exemplificar, foi a existência de aresta de extensão uando formado um ciclo, após a inserção de nova aresta no grafo de afinidades.

C.1 – Teoria Fundamental de Domínio

```
/*******************************************************
        FormerEdge Fundamental Domain Theory

    File: ForteFormerEdge.fdt
  Author: Flavia Cruz (Copyright 2002)
Description: Predicates which FORTE is not allowed to revise
    Notes:
*******************************************************/

:- module(fdt, [ node/1, edge/1, cut/2, tree/1,
            nodesAccessByTree/2, edgeAccess/3,
            notExistsFormerEdge/4,
            lastList/2,
            position/3,
            equal/2,
            not_equal/2,
            great/2,
            not_great/2,
            less/2,
```

```
        not_less/2,
      ]).

/* one clause for each of the attributes and relations used in the examples,
   plus one clause for each object type */

:- ensure_loaded(library(basics)).

node(N) :- example( node(N) ).
edge(E) :- example( edge(E) ).
cut(C,B)  :- example( cut(C,B) ).
tree(T) :- example( tree(T) ).
nodesAccessByTree(T,L) :- example( nodesAccessByTree(T,L) ).
edgeAccess(E,N,M) :- example( edgeAccess(E,N,M) ).


/* predicates which the theory references, but which FORTE is not allowed to revise */

%Garante que se existe uma aresta do nó 1 para o nó 2,
%também existe do nó 2 para o nó 1.
edgeAccess(Edge,Node1,Node2).
edgeAccess(Edge,Node2,Node1).

notExistsFormerEdge(NewNode,NodeConnected,IdTree,IdFormerEdge):-
        \+existsFormerEdge(NewNode,NodeConnected,IdTree,IdFormerEdge).


%Acha o ultimo da lista
lastList(X,[X]).
lastList(X,[H|Tail]):-
  lastList(X,Tail).
```

```
/*****************************************************
position(-+Pos,+-Elem,+List)

Pode ser utilizada para encontrar a posicao(-Pos) de um
elemento(+Elem) de uma certa lista(+List) ou para encontrar
um elemento(-Elem) em uma posicao(+Pos)de uma lista(+List).
*****************************************************/

position(1, Elem, [Elem|_]).
position(Cnt, Elem, [_|Tail]) :-
        position(P, Elem, Tail),
            Cnt is P + 1.

/* Predicados built-in utilizados na revisão*/
equal(X,Y) :-    X == Y.
not_equal(X,Y) :-    X \== Y.
great(X,Y) :-   X > Y.
not_great(X,Y) :- \+ great(X,Y).
less(X,Y) :-      X < Y.
not_less(X,Y) :-  \+ less(X,Y).

/*****************************************************
  Predicates which, although need not be redefined from
  Quintus Prolog standard definition for the domain,
  are called in the Theory file and must not be revised.
  Therefore, they must be declared here
*****************************************************/
% none...
```

C.2 – Teoria Inicial

```
/*******************************************************
      FormerEdge Initial Theory

    File: ForteFormerEdge.thy
   Author: Flavia Cruz (Copyright 2002)
Description: Predicates for FORTE to revise
    Notes:
*******************************************************/


:-ensure_loaded(library(basics)).


/*************************************************************
existsFormerEdge(+K_novo,+NodeConected,+IdTree,IdFormerEdge)
Indica se existe um Former Edge.
O Former Edge será a aresta formada pelos nós K_novo e NodeFormerEdge.

Utiliza o predicado findFormerEdge.
*************************************************************/
existsFormerEdge(NewNode,NodeConnected,IdTree,IdFormerEdge):-
        fdt:nodesAccessByTree(IdTree,Tree),
        findFormerEdge(NewNode,NodeConnected,Tree,IdFormerEdge),
%tirei para revisar fdt:cut(IdFormerEdge,0),
        !.
/*************************************************************
findFormerEdge(+K_novo,+NodeConnected,+Arvore,+Inicio,+Fim,
        -NodeFormerEdge, -WeightFormerEdge)
```

O Former Edge será a aresta formada pelos nós K_novo e NodeFormerEdge
e terá peso WeightFormerEdge.

Se K_novo foi inserido no final da Arvore e formou um ciclo entao
a aresta do former edge eh formada pelo nó anterior ao K_novo na
Arvore e K_novo, se foi no inicio será o nó posterior.
**********/

```
findFormerEdge(NewNode,NodeConnected,Tree,IdFormerEdge):-
                fdt:member(NewNode,Tree),
                 fdt:position(P,NewNode,Tree),
                 fdt:lastList(End,Tree),
                 NodeConnected==End,
                 Q is P-1,
                 fdt:position(Q,NodeFormerEdge,Tree),
                 fdt:edgeAccess(IdFormerEdge,NewNode,NodeFormerEdge),
                 ! .
findFormerEdge(NewNode,NodeConnected,[H|TailTree],IdFormerEdge):-
                 fdt:member(NewNode,[H|TailTree]),
                 fdt:position(P,NewNode,[H|TailTree]),
                 fdt:lastList(End,[H|TailTree]),
                 NodeConnected==H,
                 Q is P+1,
                 fdt:position(Q,NodeFormerEdge,[H|TailTree]),
                 fdt:edgeAccess(IdFormerEdge,NewNode,NodeFormerEdge),
                 !.
```

C.2 – Exemplos

```
/*****************************************************
FormerEdge dataset for FORTE
        FormerEdge.dat
examples output version:

Notes: neste arquivo tree estará representando o conjunto
de nós da árvore onde, ao ser esolhido um nó, poderá
formar um ciclo e former edge é a aresta de extensão.
*****************************************************/

/*************** parameters section ******************/

/* Concepts to be learned */
top_level_predicates( [existsFormerEdge(node,node,tree,edge)
        ] ).

/* Other predicates defined in the initial theory that may be called by the
  top_level and therefore may be revised */
intermediate_predicates( [
                ] ).

/* Relations used to define BK in the examples and predicates in the fdt file */
object_relations( [
        nodesAccessByTree(tree,list(node)),
        edgeAccess(edge, node, node),

        fdt: edgeAccess(edge, node, node),
        fdt: notExistsFormerEdge(node,node,tree,edge),
```

```
                fdt: lastList(node,list(node))
                %,
                %fdt: position(_,node,list(node))
                ]).

/* Propositional attributes which appear in the examples */
object_attributes( [
            edge([cut([1,0])]),
                node([]),
                tree([])
            ] ).

/* Predicates in the initial theory which should not be revised */
shielded( [findFormerEdge(node,node,tree,edge)] ).

/* depth_limit = maximum resolution depth the theory meta-interpreter will
            allow before it decides that a query has failed due to
            possible looping
  use_attr = FORTE is allowed to add antecedents representing object attributes
          to the theory
  use_relations =  FORTE is allowed to add antecedents representing object
          relations to the theory
  use_theory = FORTE is allowed to add antecedents representing predicates in
          the theory
  use_built_in = FORTE is allowed to add antecedents representing standard
          functions to the theory
  relation_tuning(highly_relational) = Relational pathfinding is always
          preferred to hill-climbing in the antecedent addition task
          relational   = The two algorithms compete normally
          non_relational    = Relational pathfinding is disabled
 */
```

```
language_bias([depth_limit(15), use_attr, use_relations, use_theory,
        use_built_in, relation_tuning(non_relational)]).

/***************** examples section ********************/
example( /* First Exemple */
    [ existsFormerEdge(3,1,t1,edge34),
      existsFormerEdge(3,5,t2,edge23
     ],
    [ existsFormerEdge(4,2,t2,edge45),
      existsFormerEdge(2,4,t1,edge12)
      %,

     /*Testes:
       existsFormerEdge(3,1,t1,edge12),
       existsFormerEdge(3,1,t1,edge13),
       existsFormerEdge(3,1,t1,edge14),
       existsFormerEdge(3,1,t1,edge23),
       existsFormerEdge(3,1,t1,edge24),

      existsFormerEdge(2,4,t1,edge13),
      existsFormerEdge(2,4,t1,edge14),
      existsFormerEdge(2,4,t1,edge23),
      existsFormerEdge(2,4,t1,edge24),
      existsFormerEdge(2,4,t1,edge34),

      existsFormerEdge(4,2,t2,edge23),
     existsFormerEdge(4,2,t2,edge24),
     existsFormerEdge(4,2,t2,edge25),
     existsFormerEdge(4,2,t2,edge34),
     existsFormerEdge(4,2,t2,edge35),
```

```
    existsFormerEdge(2,4,t1,edge13),
    existsFormerEdge(2,4,t1,edge14),
    existsFormerEdge(2,4,t1,edge23),
    existsFormerEdge(2,4,t1,edge24),
    existsFormerEdge(2,4,t1,edge34)
*/
    ],
 [ edge([ [edge12,1],
        [edge13,0],
        [edge14,0],
        [edge15,0],
        [edge23,0],
        [edge24,0],
        [edge25,0],
        [edge34,0],
        [edge35,0],
        [edge45,1]
            ]),
   /*Mesmo nao tendo atributos, tem que colocar...*/
     node([ [1],
            [2],
            [3],
            [4],
            [5]
     ]),
     tree([ [t1],
          [t2],
          [t3]
      ])
 ],
```

```
facts([
        edgeAccess(edge12,1,2),
        edgeAccess(edge13,1,3),
        edgeAccess(edge14,1,4),
        edgeAccess(edge15,1,5),
        edgeAccess(edge23,2,3),
        edgeAccess(edge24,2,4),
        edgeAccess(edge25,2,5),
        edgeAccess(edge34,3,4),
        edgeAccess(edge35,3,5),
        edgeAccess(edge45,4,5),

        nodesAccessByTree(t1,[1,2,3,4]),
        nodesAccessByTree(t2,[2,3,4,5]),
        nodesAccessByTree(t3,[1,4,3,2])
    ])
).
```

APÊNDICE D

ALGORITMO VFN

D.1 – Visão Geral

O algoritmo proposto por (NAVATHE e RA, 1989) utiliza a Matriz de Afinidades entre os Atributos, a qual é gerada através da Matriz de Uso dos Atributos, usando o mesmo método utilizado por (NAVATHE, 1984) anteriormente. Estas estruturas, já com as extensões propostas por (BAIÃO, 2001), são detalhadas nas seções 3.3.1 (Matriz de Uso dos Elementos) e 3.3.2 (Matriz de Afinidades entre os Elementos) desta dissertação.

A Matriz de Uso dos Atributos representa o uso dos atributos em transações importantes. Cada linha refere-se a uma transação. A entrada "1" na coluna indica que a transação "usa" o atributo correspondente. Se a transação recupera ou atualiza a relação pode também ser indicada por outra coluna usando R, para recuperação (retrieval) ou U para atualização (update).

A afinidade de um atributo é definida como:

$$aff_{ij} = \Sigma_{k \in T} \; acc_{kij}$$

onde acc_{kij} é o número de acessos da transação k que referencia ambos os atributos i e j. O somatório ocorre sobre todas as transações T. Esta definição de afinidades de atributos mede a força (ligação) imaginária entre dois atributos, baseado no fato que os atributos são usados juntos por transações. Baseado nesta definição de afinidades entre atributos, a matriz de afinidades entre atributos é definida como segue: é uma matriz n x n composta de n-atributos cujo elemento (i,j) é igual a aff_{ij}. A Tabela D.2 mostra a Matriz de Afinidades de Atributos, chamada de AA (*Atribuite Affinity Matrix*) que foi formada a partir da Tabela D.1. O elemento da diagonal AA(i,i) é igual a soma dos elementos na Matriz de Uso de Atributos da coluna que representa a_i. Isto é razoável já que mostra a "força" deste atributo em termos do seu uso por todas as transações.

Em abordagens anteriores, foi aplicado um algoritmo de agrupamento na matriz AA. Na abordagem de (NAVATHE e RA, 1989), porém, a matriz AA foi considerada como um

grafo completo chamado grafo de afinidades onde o valor de uma aresta representa a afinidade entre dois atributos. Em seguida, construindo uma árvore geradora linearmente conectada (*spanning tree linearly connected*), o algoritmo gera todos os fragmentos em uma iteração considerando o ciclo como um fragmento. Uma árvore linearmente conectada tem apenas duas pontas.

Tabela D.1: Matriz de Uso dos Elementos

Transações (T) x Atributos (A)											Tipo	Acc
T \ A	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}		
T_1	1	0	0	0	1	0	1	0	0	0	R	25
T_2	0	1	1	0	0	0	0	1	1	0	R	50
T_3	0	0	0	1	0	1	0	0	0	1	R	25
T_4	0	1	0	0	0	0	1	1	0	0	R	35
T_5	1	1	1	0	1	0	1	1	1	0	U	25
T_6	1	0	0	0	1	0	0	0	0	0	U	25
T_7	0	0	1	0	0	0	0	0	1	0	U	25
T_8	0	0	1	1	0	1	0	0	1	1	U	15

Tabela D.2: Matriz de Afinidades entre os Atributos (AA)

A	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}
A_1	75	25	25	0	75	0	50	25	25	0
A_2	25	110	75	0	25	0	60	110	75	0
A_3	25	75	115	15	25	15	25	75	115	15
A_4	0	0	15	40	0	40	0	0	15	40
A_5	75	25	25	0	75	0	50	25	25	0
A_6	0	0	15	40	0	40	0	0	15	40
A_7	50	60	25	0	50	0	85	60	25	0
A_8	25	110	75	0	25	0	60	110	75	0
A_9	25	75	115	15	25	15	25	75	115	15
A_{10}	0	0	15	40	0	40	0	0	15	40

As principais vantagens do método proposto acima por (NAVATHE e RA, 1989) do método exposto em (NAVATHE, 1984) são as seguintes:

- Não há necessidade de particionamento binário iterativo. A principal deficiência do particionamento binário iterativo é que a cada passo dois novos problemas são gerados aumentando a complexidade; além disso, o término do algoritmo é dependente do poder distintivo da função objetivo.
- O método não necessita de função objetivo. As funções objetivos empíricas (NAVATHE, 1984) são selecionadas após alguns testes e erros em experimentos para verificar se aquelas funções possuem um um bom poder de discriminação. Embora razoável, foi estabelecida uma escolha arbitrária. Esta arbitrariedade foi eliminada na metodologia proposta.

D.2 – Definições e notações

São usadas as seguintes notações e terminologias na descrição do algoritmo de (NAVATHE e RA, 1989):

- A,B,C,... simboliza os nós.
- a,b,c.... simboliza as arestas.
- p(e) indica o valor de afinidade da aresta e.
- ciclo primitivo indica qualquer ciclo no grafo de afinidades.
- ciclo de afinidades indica um ciclo primitivo que contém um nó de ciclo (*cycle node*). Em (NAVATHE e RA, 1989) é assumido que um ciclo significa um ciclo de afinidades, a menos que este seja determinado de outra maneira.
- aresta formadora do ciclo (*cycle completing edge*) indica a aresta que sendo selecionada forma um ciclo no grafo.
- nó de ciclo (*cycle node*) indica o nó que foi selecionado anteriormente e que faz parte da aresta formadora do ciclo.
- aresta de extensão (*former edge*) indica a aresta que foi selecionada entre o último corte e o nó ciclo.
- aresta do ciclo (*cycle edge*) indica quaisquer das arestas que participam do ciclo.

- extensão do ciclo (*extension of a cycle*) se refere ao ciclo que estende o ciclo existente através do nó ciclo.

D.3 – Conceitos Fundamentais

D.3.1 – Formação de ciclos

Baseados nas definições apresentadas na seção D.2, (NAVATHE e RA, 1989) explica a formação dos ciclos. Por exemplo, na Fig. D.1, suponha que as arestas a e b já foram selecionadas e c será a próxima aresta a ser selecionada. Neste momento, visto que c forma um ciclo, nós temos que verificar se este é um ciclo de afinidades. Isto pode ser feito verificando sua possibilidade de ciclo. A possibilidade de ciclo resulta da condição que não existe aresta extensão ou p(aresta extensão) é menor ou igual a p(todas as arestas do ciclo). O ciclo primitivo a, b, c é um ciclo de afinidades porque não há aresta extensão e portanto satisfaz a possibilidade de ciclo. Por esta razão, o ciclo primitivo a, b, c é marcado como uma partição candidata e o nó A será o nó de ciclo.

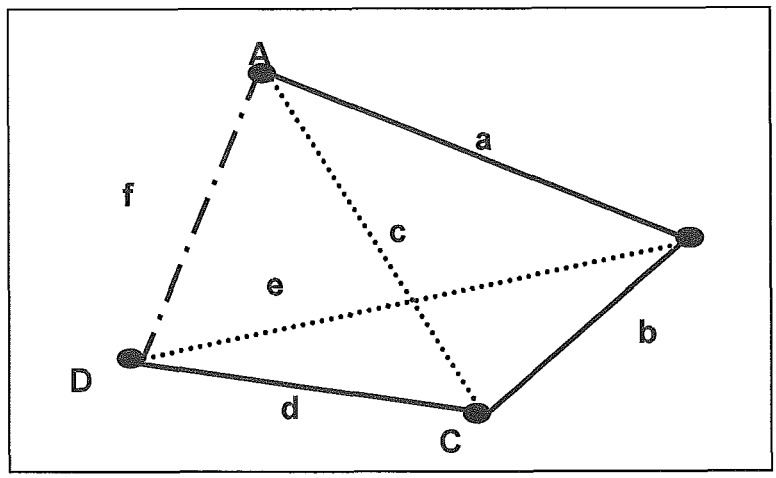


Fig. D.1 – Ciclo e extensão

D.3.2 – Extensão de ciclos

A extensão do ciclo é executada, segundo (NAVATHE e RA, 1989) da seguinte maneira: na Fig. D.1, após o nó de ciclo ser determinado, suponha que a aresta d seja selecionada. Neste momento, d é marcada como uma aresta potencial para a extensão. Isto pode ser feito verificando a possibilidade de extensão do ciclo pela aresta d. A

possibilidade de extensão do ciclo resulta da condição de que p(aresta a ser considerada ou aresta formadora do ciclo) ser maior ou igual p(qualquer aresta do ciclo). Desta maneira o antigo ciclo a, b, c é estendido pelo ciclo a, b, d, f se a aresta d que está sendo considerada, ou a aresta formadora do ciclo f, satisfaz a possibilidade de extensão que é: p(d) ou p(f) ≥ mínimo de (p(a), p(b), p(c)). O processo continua: suponha que e foi selecionada como a próxima aresta. Mas pela definição de extensão do ciclo (seção D.2) sabemos que e não pode ser considerada como uma extensão potencial porque o ciclo primitivo d, b, e não inclui o nó ciclo A. Por esta razão esta aresta é descartada e o processo continua.

D.3.3 – FORMAÇÃO DE PARTIÇÃO

Segundo (NAVATHE e RA, 1989) há duas maneiras de ocorrer um particionamento: atravé de uma nova aresta ou através de uma aresta de extensão.

1. Criando uma partição através de uma nova aresta:

Quando a próxima aresta selecionada (por exemplo a aresta d da Fig. D.1) não foi considerada anteriormente, esta é chamada de nova aresta. Se a nova aresta não satisfaz a possibilidade de extensão, então é verificada a possibilidade de extensão pela aresta formadora do ciclo (por exemplo f da Fig. D.1). Na Fig. D.1, as novas arestas d e f proporcionam, desta maneira, a possibilidade de extensão do ciclo anterior formado pelas arestas a, b, c.

Se d ou f satisfizerem a condição de possibilidade de extensão determinada acima(isto é, p(d) ou p(f) ≥ mínimo de (p(a), p(b), p(c))), então o novo ciclo estendido conteria os arestas a, b, d, f. Se a condição não fosse satisfeita, nós produziríamos um corte na aresta d (chamada de aresta cortada) isolando o ciclo a, b, c. Este ciclo agora é considerado uma partição.

2. Criando uma partição através de uma aresta extensão:

Após o corte em (1), se houver uma aresta extensão, então o nó de ciclo anterior é mudado para o nó onde a aresta cortada incide, e é verificada a possibilidade de extensão do ciclo pela aresta extensão. Por exemplo, na Fig. D.2, suponha que a, b, e c formem um ciclo onde o nó de ciclo seja A, exista um corte em d, e que a aresta extensão w exista. O nó de ciclo A é modificado para C pois a aresta cortada d foi originada em C . Agora é estimada a possibilidade de estender o ciclo a, b, c por outro ciclo que contém w. Consequentemente é considerada a possibilidade do ciclo a, b, e, w. Assuma que w ou e

não satisfaz a possibilidade de extensão, isto é, se "p(w) ou p(e) ≥ mínimo de (p(a), p(b), p(c))" não é verdadeiro. Então, segundo (NAVATHE e RA, 1989) há os seguintes resultados:

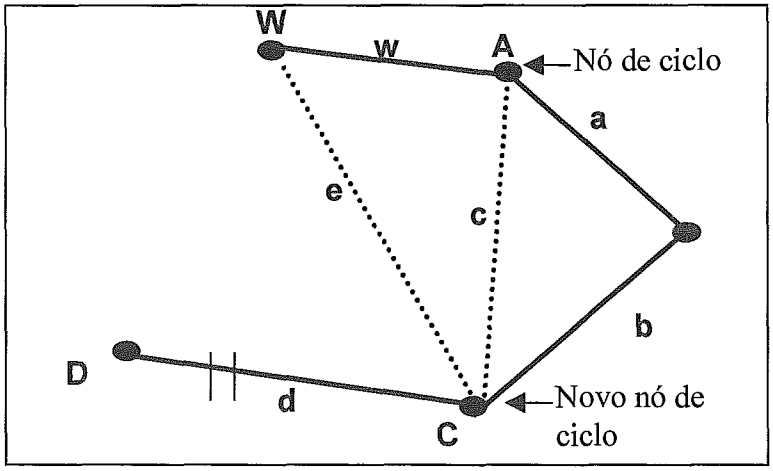


Fig. D.2 - Partição

- w será declarado como uma aresta cortada,
- C permanece como nó de ciclo, e
- a, b, c torna-se uma partição.

Alternativamente, se a possibilidade de extensão é satisfeita, o resultado seria:

- a, b, c é estendido pelo ciclo w, a, b, e
- C permanece como nó de ciclo, e
- nenhuma partição é ainda formada

D.4 – O algoritmo

Nesta seção é apresentado o algoritmo para a geração de fragmentos verticais através de um grafo de afinidades proposto por (NAVATHE e RA, 1989). Cada ciclo do grafo gera um fragmento vertical.

O algoritmo é descrito em 5 passos:

- Passo 1: Construção do Grafo de Afinidades entre os Atributos do objeto a ser considerado. Note que a matriz AA já é uma estrutura de dados adequada para representar este grafo. Nenhum outro armazenamento físico dos dados é necessário.

- Passo 2: Comece por qualquer nó.
- Passo 3: Selecione uma aresta que satisfaça as seguintes condições:

-A aresta deve ser linearmente conectada com a árvore geradora já construída;

-Deve ter o maior valor entre todas as possíveis escolhas de arestas que satisfazem a condição anterior, ou seja, deverá ser escolhida a aresta de maior valor que possa ser conectada a uma das pontas da árvore.

Esta iteração será finalizada quando todos os nós forem usados na constrtução da árvore.

- Passo 4: Se a aresta selecionada forma um ciclo primitivo então é vericada a existência do nó de ciclo (o que indica se um ciclo já foi formado anteriormente e que está na fase de tentar estender o ciclo):

-Se não existe nó de ciclo é verificada a "Possibilidade do Ciclo" (definida anteriormente) e se esta é satisfeita o ciclo é marcado como um Ciclo de Afinidades e este ciclo é considerado uma Partição Candidata. Depois selecione a próxima aresta (passo 3).

-Se existe nó de ciclo descarte esta aresta e selecione uma nova aresta (passo 3).

- Passo 5: Quando a aresta selecionada não forma ciclo e existe uma partição candidata, dedve ser verificada a existência de aresta de extensão.

-Se não existir aresta de extensão deve ser checada a possibilidade de extensão do ciclo por esta nova aresta. Caso não exista possibilidade de extensão esta aresta é cortada e considerada como partição. Vá para o passo 3.

-Se existir aresta de extensão, troque o nó de ciclo de lugar e verifique a possibilidade de extensão do ciclo pela arsta de extensão. Se não houver possibilidade de extensão, corte a aresta de extensão e considere o ciclo como uma partição. Vá para o passo 3.