UNIVERSIDADE REGIONAL DE BLUMENAU
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

# IMPLEMENTAÇÃO DISTRIBUÍDA DO ALGORITMO ADOPT

FERNANDO DOS SANTOS

BLUMENAU
2005

2005/I-19

**FERNANDO DOS SANTOS**

# IMPLEMENTAÇÃO DISTRIBUÍDA DO ALGORITMO ADOPT

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação – Bacharelado.

Prof. Dr. Jomi Fred Hübner – Orientador

**BLUMENAU**
**2005**

**2005/I-19**

# IMPLEMENTAÇÃO DISTRIBUÍDA DO ALGORITMO ADOPT

Por

**FERNANDO DOS SANTOS**

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

---

Presidente: Prof. Dr. Jomi Fred Hübner – Orientador, FURB

---

Membro: Prof. Dr. Paulo Cesar Rodacki Gomes, FURB

---

Membro: Prof. Dr. Mauro Marcelo Mattos, FURB

**BLUMENAU**
**2005**

Aos simples e humildes

*Eu estou sempre fazendo aquilo que não sou capaz,*

*numa tentativa de assim aprender como fazê-lo.*

(Pablo Ruiz Picasso).

# AGRADECIMENTOS

Aos pais, pela “especificação”, “implementação” e “suporte”.

Ao orientador, professor Jomi, pela confiança depositada e liberdade concedida no desenvolvimento deste trabalho.

Aos companheiros do grupo de pesquisa em inteligëncia artificial da FURB, professor Paulo Rodacki e amigo Daniel Tralamazza, pela oportunidade concedida de participação no grupo, principalmente nas reuniões “*off-topic*”.

Ao professor Clodoaldo Machado, do departamento de química desta universidade, pela revisão realizada nos tópicos relacionados a química.

Ao professor Roque, pelo apoio nos dias finais da redação deste volume em relação a flexibilização das tarefas delegadas a monitoria do departamento.

À namorada Fernanda, por sua compreensão, incentivo e sentimentos.

Aos amigos, pela amizade.

Ao George Lucas, por Star Wars e a Annakin Skywalker pois, apesar de ter passado para o lado negro da força transformando-se em Darth Vader, era realmente o escolhido e trouxe equilíbrio à força e assim trouxe paz ao universo. Ao chopp Eisenbahn, por proporcionar momentos saborosos e descontraídos, além de batizar o *framework*. A banda U2 cujas músicas muitas vezes embalaram o desenvolvimento deste trabalho. À energia elétrica e a Donald Knuth pelo LaTeX. Por fim, ao “verbo impessoal”, que acompanha este trabalho, inclusive estes agradecimentos.

# RESUMO

Este trabalho apresenta a especificação e implementação do algoritmo para resolução de problemas de otimização de restrições distribuído, denominado Adopt, tornando o algoritmo Adopt apto para operar em ambientes efetivamente distribuídos. Para validação, este trabalho apresenta um estudo de caso para problema de coloração de grafo. Como aplicação prática, este trabalho apresenta um estudo de caso onde é verificada a viabilidade do uso do algoritmo Adopt para resolução do problema de modelagem molecular.

**Palavras Chave**: DCOP. Adopt. Modelagem molecular.

# ABSTRACT

This work presents both the specification and implementation of one algorithm to solve distributed constraint optimization problems, called Adopt. One of the main contributions of this implementation is that the algorithm Adopt is now ready to work in effectively dstributed environments. This proposal is validated by two case studies: graph coloring problem and molecular modelling.

**Key-Words**: DCOP. Adopt. Molecular modelling.

# LISTA DE ILUSTRAÇÕES

# LISTA DE QUADROS

# LISTA DE TABELAS

# SUMÁRIO

# 1 INTRODUÇÃO

A área de Inteligência Artificial (IA) tem assistido nos últimos anos um significativo avanço científico no que diz respeito a Sistemas Multi-Agentes (SMA). Com o advento das redes de computadores, muitas informações, antes centralizadas, encontram-se distribuídas entre vários computadores interligados. Com isso, muitos problemas existentes na IA e até então resolvidos com computação centralizada tiveram que ser revistos, pois agora estavam localizados em um ambiente distribuído que nem sempre permite a centralização das informações. Na maioria dos novos paradigmas e algoritmos desenvolvidos para solucionar este problema, o conceito de SMA é aplicado, proporcionando a expansão do universo dos SMA também para a área de Computação Distribuída. Agentes de um SMA que antes eram executados em um único computador e disputavam o mesmo processador agora podem ser distribuídos em diversos computadores interligados, proporcionando um paralelismo de execução ainda maior.

Uma das classes de problemas que foi revista para operar em ambientes distribuídos foi a dos problemas de satisfação de restrição. Simplificadamente, um *Constraint Satisfaction Problem* (CSP) consiste de um conjunto de variáveis e um conjunto de restrições que influenciam a determinação dos valores para estas variáveis, cuja solução é encontrada quando cada variável assume um valor que satisfaça as restrições. Graficamente, costuma-se representar um CSP através de um grafo de restrições, conforme é apresentado na fig. 1.1, que ilustra um CSP com três variáveis($\{x_1, x_2, x_3\}$) e duas restrições (vértices e arestas do grafo, respectivamente). Quando um CSP está em um ambiente distribuído, é chamado de *Distributed Constraint Satisfaction Problem* (DCSP), e Yokoo (2001) apresenta três algoritmos assíncronos distintos para solucionar um DCSP, denominados *Asynchronous Backtracking* (AB), *Asynchronous Weak Commitment* (AWC) e *Branch and Bound* (BB). O GRUPO DE INTELIGÊNCIA ARTIFICIAL DA FURB (2005) disponibiliza um *framework*, chamado dynDCSP, com a implementação destes três

algoritmos, realizadas de maneira a possibilidar sua execução em ambientes distribuídos compostos por vários *hosts*.

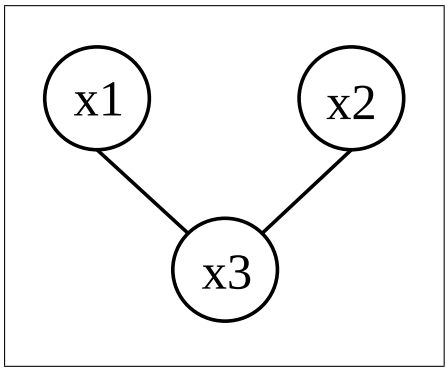



**Figura 1.1** – Um CSP simples

Porém, um CSP (e conseqüentemente um DCSP) não é suficiente para especificar determinados tipos de problemas onde é necessário encontrar a *melhor* solução, e não apenas *qualquer* solução (como ocorre por exemplo no problema de agendamento de tarefas). Nestes casos, é necessário utilizar uma extensão de CSP, denominada de problema de otimização de restrições.

Um *Constraint Optimization Problem* (COP) considera um custo para cada solução possível, e o objetivo é encontrar uma solução de custo ótimo. Para se solucionar um COP, Tsang (1993, p. 301) apresenta um algoritmo centralizado, denominado BB, cujo princípio de funcionamento é baseado em um custo global que serve de parâmetro para poda em uma busca em profundidade.

Um COP em ambiente distribuído é denominado de *Distributed Constraint Optimization Problem* (DCOP). Modi (2003b, p. 20) comenta que um algoritmo existente para solucionar um DCOP é o *Synchronous Branch and Bound* (SBB). Porém, como o próprio nome do algoritmo revela, é síncrono, o que faz com que o algoritmo não utilize o potencial do paralelismo de uma execução distribuída. Em função disto, Modi (2003b) apresenta o primeiro algoritmo assíncrono para solucionar DCOP, denominado *Asynchronous Distributed Optimization* (Adopt). O algoritmo Adopt utiliza o conceito de limiares (inferior e superior) de solução parcial para refinar a busca e otimizar o *backtrack*. Em Modi (2003a), o autor disponibiliza uma implementação na linguagem Java do algoritmo

Adopt. Porém, esta implementação utiliza *threads* para representar os agentes, e não dispõe de um mecanismo de comunicação que permita a comunicação entre os agentes quando os mesmos estão localizados em diferentes *hosts*.

Para disponibilizar uma implementação do algoritmo Adopt que esteja apta a operar em ambientes efetivamente distribuídos, surge a proposta de especificar e implementar este algoritmo a partir do *framework* dynDCSP.

Para apresentar uma aplicação prática do algoritmo Adopt é utilizado o problema de modelagem molecular. A expectativa é diminuir o tempo necessário para determinação de modelos moleculares (disposição tri-dimensional dos átomos de uma molécula) em função do paralelismo que uma execução distribuída pode proporcionar, além de verificar a viabilidade da abordagem através de DCOP para o problema de modelagem molecular.

## 1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho consiste em implementar o agoritmo proposto por Modi (2003b) para solução de DCOP, denominado Adopt, com a condição de que o mesmo possa ser executado em um ambiente distríbuído (vários *hosts*) para então especificar o problema de modelagem molecular como um DCOP e verificar a viabilidade da utilização dos conceitos de DCOP como forma de realizar a modelagem molecular.

## 1.2 ESTRUTURA DO TRABALHO

O conteúdo deste trabalho está dividido em seis capítulos. O capítulo 2 apresenta o conceito de problemas de satisfação de restrições, sendo detalhados problemas de satisfação de restrições centralizados (seção 2.1), distribuídos (seção 2.2), problemas de otimização de restrições centralizados (seção 2.3) e distribuídos (seção 2.4) sendo que nesta última seção é apresentado o algoritmo Adopt. Além disto, o capítulo 3 apresenta o *framework* dynDCSP.

O capítulo 4 apresenta a especificação e o desenvolvimento do trabalho, sendo descritos os principais requisitos e a implementação. O capítulo 5 apresenta um estudo de caso com problema de coloração de grafos para validar a implementação do algoritmo

Adopt. Já o capítulo 6 apresenta um estudo de caso verificando a viabilidade da especificação do problema de modelagem molecular como COP/DCOP e sua resolução através do algoritmo Adopt.

Por fim, o capítulo 7 apresenta as conclusões obtidas com a realização deste trabalho e sugestões para extensões do mesmo.

# 2 PROBLEMAS DE SATISFAÇÃO DE RESTRIÇÕES

Este capítulo apresenta conceitos e algoritmos referentes à problemas de satisfação de restrições centralizados e distribuídos e também sobre problemas de otimização de restrições centralizados e distribuídos, sendo que na seção de problema de otimização de restrições distribuído é apresentado o algoritmo Adopt. Por fim, este capítulo expõe o *framework* dynDCSP.

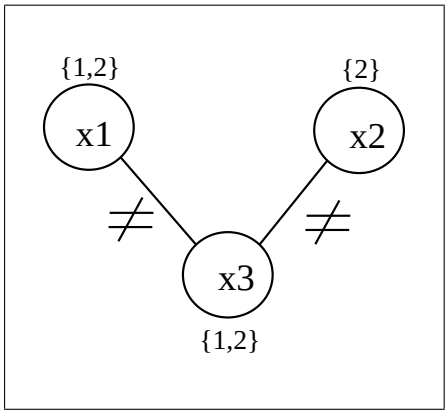
## 2.1 PROBLEMA DE SATISFAÇÃO DE RESTRIÇÕES CENTRALIZADO

"Um problema de satisfação de restrição, *Constraint Satisfaction Problem* (CSP), é um *framework* geral que pode formalizar vários problemas na Inteligência Artificial" (YOKOO, 2001, p. 1, tradução nossa). De maneira geral, um CSP é uma classe de problemas que envolvem três componentes: *variáveis*, *domínios* e *restrições*. Uma variável corresponde a uma parte do problema que pode ter seu valor alterado. Um domínio consiste de um conjunto de valores pré-definidos que uma variável pode assumir. Uma restrição corresponde a uma condição que deve ser satisfeita ao mapear valores dos domínios para as variáveis.

Segundo Russel e Norvig (2003), um CSP pode ser definido formalmente através da tupla $< X, D_i, C >$, onde: $X = \{x_1, x_2, \ldots, x_n\}$ é o conjunto de variáveis; $D_i = \{d_1, d_2, \ldots, d_n\}$ é o conjunto de domínios (sendo que $d_i$ é domínio de $x_i$) e $C = \{C_1, C_2, \ldots, C_n\}$ é o conjunto de restrições. A associação de uma variável a um valor ($\{x_i = v_i\}$) é denominada de *label* e um conjunto de *labels* ($\{x_i = v_i, x_j = v_j, \ldots\}$) é denominado de solução parcial. Uma solução parcial será a solução de um CSP se possuir um *label* para cada variável e se estes *labels* tornam consistente (satisfazem) todas as restrições.

Uma maneira bastante elucidativa de representar um CSP é através de um grafo de restrições. Nesta forma de representação, os vértices do grafo correspondem à variáveis

e os arcos correspondem à restrições (RUSSEL; NORVIG, 2003). Um exemplo de CSP, representado na forma de grafo de restrições, é apresentado na fig. 2.1. Consiste em um problema de coloração, onde o objetivo é definir uma cor (representada numéricamente) para cada variável ($\{x_1, x_2, x_3\}$) a partir das cores disponíveis nos domínios das variáveis ($\{\{1,2\},\{2\},\{1,2\}\}$), respeitando as restrições ($\{\{x_1 \neq x_3\},\{x_2 \neq x_3\}\}$). Yokoo (2001, p. 8-15) apresenta dois tipos de algoritmos para solucionar CSPs: *backtracking* e *iterative improvement*.




Fonte: adaptado de Yokoo (2001, p. 5)

**Figura 2.1** − Exemplo de CSP

### 2.1.1 *Backtracking*

Um dos tipos de algoritmos mais simples e sistemáticos para solucionar um CSP é o algoritmo de *backtracking*, cujo funcionamento, em sua forma mais simples de implementação, é idêntico ao de um algoritmo de busca em profundidade. Yokoo (2001, p. 9-14) apresenta três métodos para melhoria do algoritmo de *backtracking*: heurística de conflito mínimo, *forward-checking* e *dependecy-directed*.

A heurística de conflito mínimo consiste em escolher o valor da variável de forma a minimizar a quantidade de restrições violadas. Ou seja, ao adicionar um *label* a uma solução parcial, deve-se optar por adicionar aquele que viole nenhuma ou a menor quantidade de restrições.

Utilizando *forward-checking* é possível tornar o *backtracking* menos seqüencial. O método de *forward-checking* consiste em criar um *label* (associar um valor) para a variável

que possui a menor quantidade de valores do domínio consistentes com a solução parcial.

Já a técnica de *dependecy-directed* consiste em identificar qual é o *label* que está causando a eventual inconsistência da solução atual, para dessa forma, retornar o *backtracking* até o ponto em que este *label* foi adicionado a solução parcial e continuar a partir daquele ponto.

### 2.1.2 *Iterative improvement*

Segundo Yokoo (2001, p. 15), o funcionamento deste tipo de algoritmo é análogo aos algoritmos de busca *subida da montanha*. Inicialmente, um *label* é criado para cada variável, e em seguida, os valores dos *labels* são alterados até encontrar uma solução que satisfaça todas as restrições. Para otimizar a busca, pode-se utilizar heurística de conflito mínimo, objetivando "guiar" o processo de busca.

## 2.2 PROBLEMA DE SATISFAÇÃO DE RESTRIÇÕES DISTRIBUÍDO

Um problema de satisfação de restrições distribuído, *Distributed Constraint Satisfaction Problem* (DCSP), pode ser visto como um mapeamento *variáveis-agentes*. As variáveis, os domínios e as restrições são distribuídos entre agentes autônomos, onde cada agente possui uma ou mais variáveis e tem por objetivo determinar um valor para as mesmas. Existem restrições entre diferentes agente que devem ser considerados na resolução do DCSP.

Yokoo (2001, p. 48) define formalmente um DCSP como um conjunto de $m$ agentes, onde cada variável $x_i$ do CSP pertence a um agente $i$ (esta relação é representada por $pertence(x_j, i)$). As restrições são distribuídas entre os agentes, e o fato de que um agente $l$ conhece uma restrição $c_k$ é representada por $conhece(c_k, l)$. A solução de um DCSP é encontrada quando: $\forall i, \forall j$ onde $pertence(x_j, i)$, o valor de $x_i$ é igual a $v_j$ e $\forall l, \forall c_k$ onde $conhece(c_k, l)$, $c_k$ é satisfeita considerando a associação $x_j = v_j$.

Yokoo (2001, p. 55-92) apresenta três algoritmos para solucionar DCSPs. São eles: AB, AWC e *Distributed Breakout* (DB).

### 2.2.1 *Asynchronous Backtracking*

Neste algoritmo, cada agente $a_i$ está associado a uma variável $x_i$ e a um conjunto de restrições $C_i$ sobre a variável $x_i$. Assim que $a_i$ determina um valor $v_i$ para $x_i$, este envia uma mensagem com o valor de $x_i$ para todos os demais agentes $a_n$ cujas restrições $C_n$ contenham $x_i$ para que seja verificada a consistência da associação $x_i = v_i$ e $x_n = v_n$. Se a associação $x_i = v_i$ não for consistente, $a_n$ efetua *backtrack* enviando para $a_i$ uma mensagem indicando a não consistência. $a_i$ então determina um novo valor para $x_i$ e o processo é reiniciado (YOKOO, 2001; BRUNS, 2004).

### 2.2.2 *Asynchronous Weak Commitment*

O princípio de funcionamento deste algoritmo é bastante semelhante ao *Asynchronous Backtracking*. A diferença está no fato de que para cada agente está associada uma prioridade. As mensagens para verificação de consistência não são enviadas apenas para os agentes que possuem a variável $x_i$ no conjunto de restrições, mas a todos os agentes de menor prioridade. Além disto, este valor de prioridade é dinâmico, ou seja, altera-se durante a execução do algoritmo. Isto acontece para evitar a saturação de alguns poucos agentes na verificação da consistência da solução (YOKOO, 2001; TRALAMAZZA, 2004).

### 2.2.3 *Distributed Breakout*

Neste algoritmo, um valor de peso é determinado para cada restrição violada. O objetivo do algoritmo é encontrar valores para as variáveis de forma a minimizar a quantidade de restrições violadas, buscando zero para esta quantidade. Assim que o algoritmo detecta a execução de várias iterações sem diminuição da quantidade de restrições violadas (estado de mínimo local), o valor de prioridade das variáveis que estão causando o estado de mínimo local é incrementado. Sendo assim, todas as restrições dos demais agentes passam a ser analisadas em função dos valores destas variáveis de maior prioridade (YOKOO, 2001).

## 2.3 PROBLEMA DE OTIMIZAÇÃO DE RESTRIÇÕES

Tsang (1993, p. 299) chama a atenção para o fato de que, quando um CSP apresenta mais de uma solução, estas soluções são consideradas igualmente satisfatórias. Porém, existem determinados tipos de problemas em que uma solução é melhor que outras. Nestes casos, o objetivo não é apenas encontrar uma solução, mas sim encontrar a *melhor* solução. Os tipos de problemas que enquadram-se nesta categoria (cujo objetivo é encontrar a melhor solução que satisfaça as restrições) são denominados de problemas de otimização de restrições (TSANG, 1993, p. 299).

Segundo Tsang (1993, p. 300), um problema de otimização de restrições, *Constraint Optimization Problem* (COP), pode ser formalmente definido através da tupla $< X, D_i, C, f >$, onde $X$, $D_i$, e $C$ correspondem aos mesmos elementos da tupla que define um CSP (seção 2.1). A diferença está no elemento $f$, que representa uma função de otimização que mapeia cada solução para um valor numérico. A solução de um COP corresponde a um conjunto de *labels* contendo um *label* para cada variável, que satisfaça todas as restrições e cujo valor de $f$ seja ótimo (mínimo ou máximo, dependendo do caso).

Um exemplo de COP é apresentado na fig. 2.2 (página 24). Consiste em associar um valor para cada uma das variáveis, respeitando as restrições e de forma a *minimizar* a função de custo $f$, dada por: $f = x_1 + x_2 + x_3$. Tsang (1993, p. 301) apresenta um algoritmo para solucionar um COP, denominado *Branch and Bound* (BB), cujo princípio de funcionamento é descrito a seguir.

### 2.3.1 *Branch and Bound*

O algoritmo BB consiste de uma adaptação da busca em profundidade através da aplicação de heurística sobre a função de otimização para estimar os "melhores" valores para variáveis. Desta maneira, a criação de novos nós folhas é restringida de forma a guiar o ramo da árvore de busca para a melhor solução. No caso de um COP, a heurística corresponde a uma função $h$ que mapeia uma solução parcial para um valor numérico, sobrestimando ou subestimando (para maximizar ou minimizar) o valor de $f$ (TSANG,

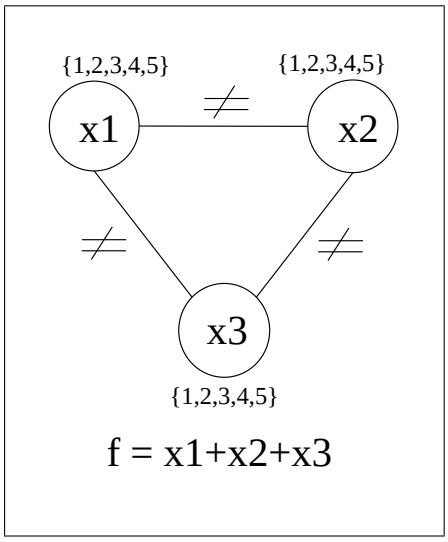



**Figura 2.2** – Exemplo de COP

1993, p. 301)

O algoritmo BB requer uma variável global, denominada *bound*, que armazena o melhor valor de $h$ obtido no decorrer da execução. Inicialmente o valor de *bound* é definido como $+\infty$ para um problema de minimização ou $-\infty$ para um problema de maximização. O algoritmo então inicia uma busca em profundidade por soluções. Porém, assim que um novo *label* está para ser adicionado à solução parcial, o valor de $h$ é computado. Se este valor for menor (em um problema de maximização) do que *bound*, a sub-árvore que seria gerada por esta solução parcial é podada. Assim que uma solução para o COP é encontrada, o valor de $f$ é computado e passará a ser o novo *bound* se e somente se este valor for maior (em um problema de maximização) que o *bound* atual. Se o valor de $f$ assumir esta condição, a solução encontrada passa a ser uma das ou a melhor(es) solução(ões). O algoritmo termina quando todas as partes do espaço de busca foram examinadas ou podadas (TSANG, 1993, p. 301-302).

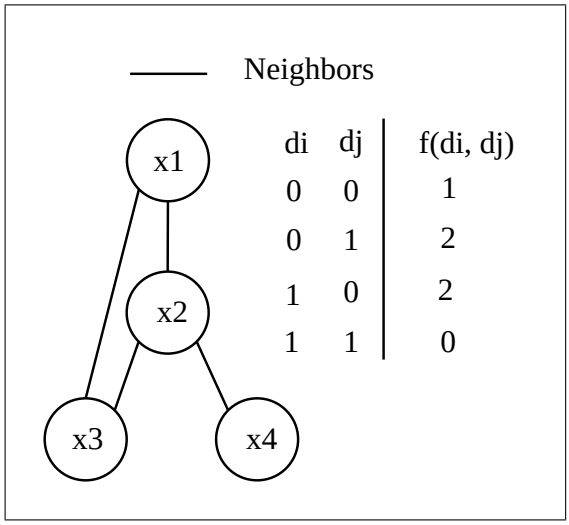
## 2.4 PROBLEMA DE OTIMIZAÇÃO DE RESTRIÇÕES DISTRIBUÍDO

Segundo Modi (2003b, p. 2), um problema de otimização de restrições distribuído, *Distributed Constraint Optimization Problem* (DCOP), consiste de uma extensão de um DCSP onde também há um mapeamento *variável-agente*. Da mesma forma como ocorre com COP, há uma função de otimização que deve ser considerada para obtenção da melhor

solução.

Formalmente, Modi (2003b, p. 13) define um DCOP como um conjunto $V$ de $n$ variáveis $V = \{x_1, x_2, \ldots, x_n\}$ cada uma associada a um agente, sendo os valores das variáveis tomados de domínios discretos e finitos $D_1, D_2, \ldots, D_n$ respectivamente. O objetivo de um DCOP é encontrar valores para as variáveis de forma a maximizar ou minimizar uma função de otimização $f$, que é composta de um conjunto de funções de custo (análogas as *constraints* dos DCSPs) que em vez de retornarem *satisfeita* ou *insatisfeita* (o que ocorre nas *constraints*), retornam um valor numérico. Para cada par de variáveis $x_i$, $x_j$ deve existir uma função de custo $f_{ij}$ que mapeia os *labels* para um valor numérico.

A fig. 2.3 apresenta um exemplo de DCOP. Neste exemplo, existem quatro variáveis ($\{x_1, x_2, x_3, x_4\}$), e cada variável possui o domínio $D = \{0, 1\}$. Existem também quatro restrições (para clareza, interpretar como *relações*) entre estas variáveis, sendo: $\{(x_1, x_2), (x_1, x_3), (x_2, x_3), (x_2, x_4)\}$. Em DCOP, duas variáveis relacionadas são denominadas *neighbors*. Existe uma função de custo $f(d_i, d_j)$ associada a cada relação $(x_i, x_j)$, cuja avaliação é dada em função dos valores $d_i$ e $d_j$ que as variáveis da relação assumem.




Fonte: Modi (2003b, p. 12)

**Figura 2.3** – Exemplo de DCOP

Neste exemplo, o objetivo é encontrar uma associação *variável-valor* $A$ em que o custo total $F$ seja mínimo, conforme apresenta a equação (2.1) (MODI, 2003b, p. 14).

$$F(A) = \sum_{x_i, x_j \in V} f_{ij}(d_i, d_j), \text{ onde } x_i \leftarrow d_i,\ x_j \leftarrow d_j \text{ em } A \tag{2.1}$$

Modi (2003b) comenta que um algoritmo disponível para solucionar um DCOP é o SBB. Trata-se de uma extensão do BB para ambientes distribuídos. Porém, este algoritmo requer um agente que centralize os valores das funções de custo para dessa forma, comparar com o *bound* global, tornando o SBB um algoritmo síncrono que não aproveita o paralelismo que uma computação distribuída pode oferecer. Em vista disto, Modi (2003b) apresenta um algoritmo assíncrono para solucionar DCOPs, denominado Adopt, e que será apresentado na seção a seguir.

### 2.4.1 *Asynchronous Distributed Optimization*

"Adopt é o primeiro algoritmo para DCOP que encontra soluções ótimas usando apenas comunicação assíncrona localizada" (MODI, 2003b, p. 20, tradução nossa). Neste algoritmo, um agente não necessita informação global para tomar decisões, bastanto apenas informação local. A comunicação é dita localizada pois é feita apenas entre *neighbors* (duas variáveis são ditas *neighbors* se apresentam relação entre si). A seguir, são apresentadas as características do Adopt. Porém, antes é necessário definir dois conceitos importantes utilizados no algoritmo: *lower bound* e *upper bound*.

Os conceitos *lower bound* e *upper bound* correspondem à valores numéricos (calculados em função de um valor $d_i$ do domínio da variável) que determinam um intervalo, de limite inferior dado por *lower bound* e limite superior dado por *upper bound*. Este intervalo corresponde ao espaço em que o custo da solução ótima estará localizado se o valor da variável for $d_i$. A fig. 2.4 representa graficamente esta definição através de retângulos (pontilhados e tracejados) que correspondem a "área" em que a solução ótima estará localizada dado um valor de domínio (sendo *lower bound* representado por LB e *upper bound* por UB). Por exemplo, ao se considerar o valor de domínio 3 (eixo $d_i$), serão obtidos LB(3) = 1 e UB(3) = 3 (eixo *bound*), ou seja, o custo de uma solução para o problema que considere $d_i = 3$ não será menor que 1 e nem maior que 3, ficando (o custo)

restrito ao retângulo tracejado entre $bound = 1$ e $bound = 3$.

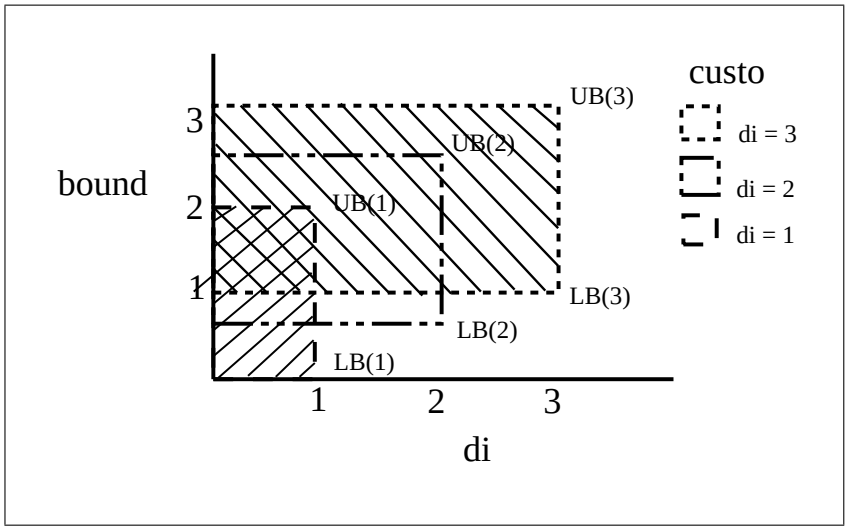


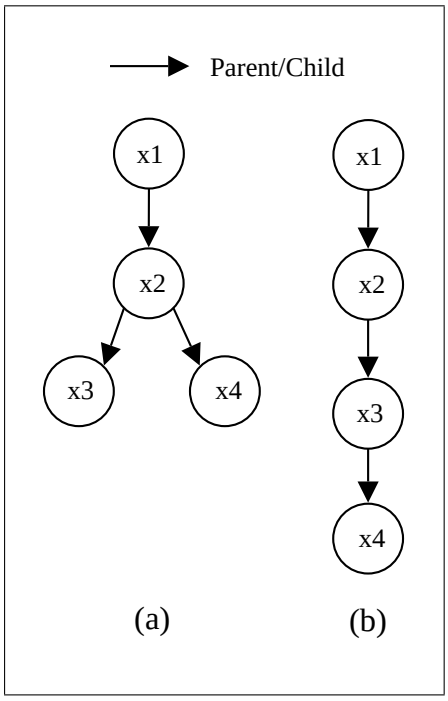

**Figura 2.4** – Representação gráfica de *lower bound* e *upper bound*

Um requisito importante necessário para o algoritmo Adopt é que os agentes sejam organizados em uma árvore de busca em profundidade, *Depth-First Search Tree* (DFST). Esta organização tem como objetivo priorizar os agentes dispondo-os de maneira acíclica, evitando com isso "ciclos" na comunicação. Nesta organização em DFST, um agente deve possuir um *parent* (com exceção do raiz) e pode ter múltiplos *childrens*. A fig. 2.5(a) (página 28) apresenta a organização em DFST dos agentes do exemplo da fig. 2.3 em que o agente $x_1$ é *parent* de $x_2$, logo, $x_2$ é *children* de $x_1$. Na fig. 2.5(b) (página 28) os agentes estão dispostos em uma DFST linear (que também pode ser utilizada pelo algoritmo Adopt).

#### 2.4.1.1 Características do algoritmo Adopt

O algoritmo Adopt apresenta três características particulares: a forma de busca por solução (baseada em *lower bounds*); a maneira como uma solução anteriormente examinada é reconstruída; e a deteção de término integrada (MODI, 2003b, p. 20-24). A seguir, são detalhadas estas três características.

Modi (2003b, p. 21) afirma que um *lower bound* pode ser computado apenas baseado nas informações locais, sem o conhecimento dos custos globais. Seguindo este princípio, no algoritmo Adopt cada agente determina o valor da sua variável baseado




Fonte: Modi (2003b, p. 24)

**Figura 2.5** – DCOP ordenado em DFST

apenas nas informações locais (proveniente dos *neighbors*), selecionando o valor que apresente o menor *lower bound*. Porém, esta estratégia permite que um agente abandone uma solução parcial antes que seja provado que esta solução não é ótima, o que pode fazer com que o agente tenha que revisitar esta solução em um dado momento.

Para permitir que um agente revisite eficientemente uma solução previamente examinada, o algoritmo Adopt utiliza o conceito de *threshold*. O *threshold* pode ser visto como uma "versão distribuída" do *bound* presente no algoritmo BB (distribuído pois é computado em cada agente). O valor de *threshold* funciona como um grau de aceitação de uma solução parcial, pois o valor da variável do agente só será alterado quando o *lower bound* do *label* atual for maior que o *threshold* do agente (de onde pode-se deduzir que qualquer solução cujo custo seja inferior ao valor de *threshold* é válida). A estratégia utilizada para revisitar uma solução parcial é armazenar o valor de *threshold* de um agente $x_q$ em seu *parent* $x_p$. Desta forma, quando $x_p$ decide revisitar uma solução, o valor de *threshold* é enviado para $x_q$, fazendo com que $x_q$ tome como válida qualquer solução parcial cujo *lower bound* seja menor que o *threshold* recebido. É importante lembrar que o valor do *threshold* deve considerar também o custo computado pelos *childrens* da DFST.

Desta forma, quando $x_q$ recebe o valor de *threshold* de seu *parent*, $x_q$ deve subdividir este valor entre si e seus *childrens* (MODI, 2003b, p. 21).

Por fim, o algoritmo Adopt apresenta um mecanismo integrado de detecção de término, baseado no intervalo formado entre *lower bound* e *upper bound*, pois quando o tamanho deste intervalo é zero, a solução ótima foi encontrada. O mecanismo integrado de detecção de término suprime a utilização de algum outro algoritmo específico para esta tarefa (MODI, 2003b, p. 23).

Textualmente, cada agente no algoritmo Adopt executa concorrentemente os passos apresentados no quadro 2.1 (página 29).

1. Inicializa o *lower bound* de cada valor em $D_i$ para zero. Associa um valor randômico para sua variável $x_i$;
2. Envia o valor atual de $x_i$ para cada *neighbor* localizado abaixo de si (de acordo com a DFST);
3. Quando recebe o valor de um *neighbor* $x_j$, para cada valor $d$ de $D_i$, avalia a restrição entre $x_i$ e $x_j$. Adiciona o custo da restrição ao *lower bound* de cada um de seus possíveis valores. Se o *lower bound* do valor atual é maior que o *lower bound* de algum valor $d$, alterar o valor atual para $d$;
4. Envia o *lower bound* de seu valor com menor *lower bound* para seu *parent* na DFST. Anexa o valor da variável do *parent* sob o qual este *lower bound* foi computado como sendo o *context*;
5. Quando recebe um *lower bound* de um *child* com um valor $d$, adiciona o *lower bound* reportado no *lower bound* de $d$. Se o *lower bound* do valor atual for maior que o *lower bound* de algum outro valor do domínio, alterar o valor atual para aquele com menor *lower bound*;
6. Quando um *neighbor* localizado acima (de acordo com a DFST) altera o valor de sua variável, reinicializa o *lower bound* de cada valor de $D_i$ para zero;
7. Continua enviando e recebendo mensagens e alterando os valores como descrito acima até que a seguinte condição de término seja verdadeira: o *lower bound* LB de um valor $d$ é também seu *upper bound* e o *lower bound* de qualquer outro valor de $D_i$ é maior que LB. Quando esta condição ocorrer, $d$ é o valor globalmente ótimo para $x_i$ até que (ou a menos que) um *neighbor* de cima (de acordo com a DFST) altere seu valor.

Fonte: Modi (2003b, p. 27-28)

**Quadro 2.1** – Passos executados por cada agente no algoritmo Adopt

Para comunicação entre os agentes, o algoritmo Adopt apresenta quatro tipos de mensagens: VALUE, THRESHOLD, COST e TERMINATE. Um agente $x_i$ envia mensagens VALUE para seus *neighbors* que estão localizados abaixo de si (de acordo com a DFST) com o objetivo de informar estes agentes do valor atual $d_i$ de $x_i$. Mensagens do tipo THRESHOLD são enviadas apenas para os *childrens* de $x_i$, e objetivam informar estes *childrens* os respectivos valores de *threshold*. Já mensagens do tipo COST são enviadas de *childrens* para *parent* e visam informar ao *parent* (supor $x_i$) o custo calculado quando $x_i$ assume o valor $d_i$, custo este, informado através do *lower bound* e do *upper bound* dos agentes *childrens* de $x_i$. Por fim, uma mensagem do tipo TERMINATE é enviada de *parent* para *childrens* e tem por objetivo informar a finalização da execução do algoritmo.

Antes de apresentar o pseudo-código do algoritmo Adopt, é preciso definir o conceito de *context* e também expor como é calculado o custo, o *lower bound* e o *upper bound*.

Um *context* é uma solução parcial $\{(x_i, d_i), (x_j, d_j), \dots\}$, onde uma variável pode apresentar apenas um *label*. Dois *contexts* são compatíveis se apresentarem o mesmo valor para cada uma de suas variáveis. *CurrentContext* é um *context* presente em um agente $x_i$ que contém *labels* das variáveis localizadas acima de $x_i$ na DFST.

O custo local de um agente $x_i$ quando este escolhe um valor $d_i \in D_i$, denotado por $\delta(d_i)$, corresponde a soma dos custos das relações (funções de custo) entre $x_i$ e os agentes localizados acima na DFST. Formalmente: $\delta(d_i) = \sum_{(x_j,d_j)\in CurrentContext} f_{ij}(d_i, d_j)$

O *lower bound* de um valor $d$ para um agente $x_i$, denotado por $LB(d)$ corresponde a soma da função de custo $\delta(d)$ com os *lower bounds* reportados pelos agentes *childrens* $x_l$ através de mensagens COST. Formalmente: $\forall d \in D_i, LB(d) = \delta(d) + \sum_{x_l \in Children} lb(d, x_l)$. Da mesma forma, o *upper bound* de um valor $d$ ($UB(d)$) corresponde a soma do custo local $\delta(d)$ com os *upper bounds* reportados pelos *childrens*: $\forall d \in D_i, UB(d) = \delta(d) + \sum_{x_l \in Children} ub(d, x_l)$

Além do cálculo de *lower bound* e *upper bound* para cada valor $d$ do domínio $D_i$

de uma variável, o algoritmo Adopt requer que seja computado um *lower bound* e um *upper bound* para cada *variável*. O *lower bound* de uma variável $x_i$, denotado por $LB$, é o menor *lower bound* dentre todos os valores $d$. Formalmente: $LB = min_{d \in D_i} LB(d)$. Já o *upper bound* de uma variável $x_i$, denotado por $UB$, é o menor *upper bound* dentre todos os valores $d$, ou seja: $UB = min_{d \in D_i} UB(d)$

O valor de *threshold* de um agente é armazenado em uma variável local (em cada agente) chamada *threshold*. O valor de *threshold*, inicialmente zero, é alterado de duas formas: através de mensagens THRESHOLD, enviadas por um agente *parent* ou então para manter a seguinte relação: $LB \leq threshold \leq UB$.

O pseudo-código do algoritmo Adopt, (MODI, 2003b, p. 35-36), é apresentado nos algoritmos 2.1 e 2.2 (páginas 32 e 33 respectivamente).

O procedimento `maintainThresholdInvariant` (linha 55) é responsável por manter a relação $LB \leq threshold \leq UB$. Já `maintainAllocationInvariant` (linha 60) tem o papel de subdividir o *threshold* atual entre os *childrens*. Por fim, `maintainChildThresholdInvariant` (linha 68) realoca o *threshold* atual entre os *childrens* (quando necessário) após estes terem reportado seus custos ($LB$ e $UB$) através de mensagens COST.

#### 2.4.1.2 Exemplo de execução do algoritmo Adopt

Nesta seção é apresentado um exemplo de execução do algoritmo Adopt para o problema de otimização apresentado na fig. 2.3. Vale lembrar que um agente é criado para cada variável do problema, sendo este agente responsável pela execução do algoritmo apresentado nos algoritmos 2.1 e 2.2. Nota-se que neste exemplo de execução, apenas são feitas referências as ações ou mensagens que alteram o estado de um ou mais agentes.

Inicialmente, cada agente executa as inicializações descritas no algoritmo (linhas linha 5 e linha 3), seleciona um valor para sua variável, neste caso, todos os agentes selecionaram o valor zero, e envia este valor através de mensagens VALUE para seus *childrens*.

```
1  initialize;
2  begin
3      threshold ← 0; CurrentContext ← {};
4      forall x_l ∈ Children, d ∈ D_i do
5          lb(d, x_l) ← 0; ub(d, x_l) ← ∞; t(d, x_l) ← 0; context(d, x_l) ← {};
6      d_i ← d that minimizes LB(d);
7      backTrack;
8  end
9  when received (THRESHOLD, t, context)
10     if context compatible with CurrentContext then
11         threshold ← t;
12         maintainThresholdInvariant;
13         backTrack;
14 when received (TERMINATE, context)
15     record TERMINATE received from parent;
16     CurrentContext ← context;
17     backTrack;
18 when received (VALUE, (x_j, d_j))
19     if TERMINATE not received from parent then
20         add (x_j, d_j) to CurrentContext;
21         forall x_l ∈ Children, d ∈ D_i do
22             if context(d, x_l) incompatible with CurrentContext then
23                 lb(d, x_l) ← 0; ub(d, x_l) ← ∞;
24                 t(d, x_l) ← 0; context(d, x_l) ← {};
25     maintainThresholdInvariant;
26     backTrack;
27 when received (COST, x_k, context, lb, ub)
28     d = value of x_i in context;
29     remove (x_i, d) from context;
30     if TERMINATE not received from parent then
31         forall (x_j, d_j) ∈ context and x_i is not my neighbor do
32             add (x_j, d_j) to CurrentContext;
33         forall x_l ∈ Children, d' ∈ D_i do
34             if context(d', x_l) incompatible with CurrentContext then
35                 lb(d', x_l) ← 0; up(d', x_l) ← ∞;
36                 t(d', x_l) ← 0; context(d', x_l) ← {};
37     if context compatible with CurrentContext then
38         lb(d, x_k) ← lb; ub(d, x_k) ← ub; context(d, x_k) ← context;
39         maintainChildThresholdInvariant;
40         maintainThresholdInvariant;
41     backTrack;
```

**Algoritmo 2.1** – Algoritmo Adopt, parte 1

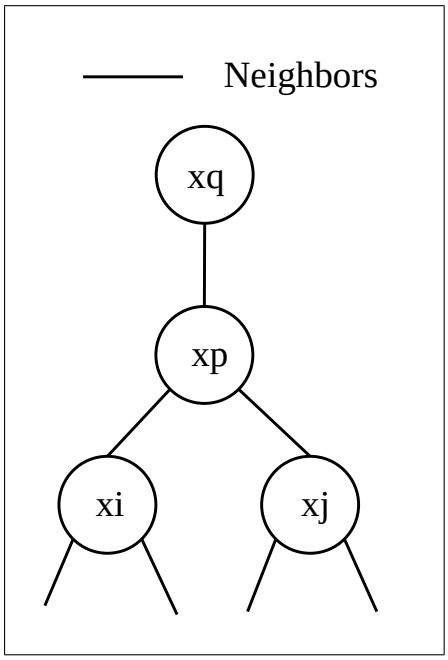
```
42 procedure backTrack
43     if threshold == UB then
44         d_i = d that minimizes UB(d);
45     else
46         if LB(d_i) > threshold then
47             d_i = d that minimizes LB(d);
48     SEND(VALUE, (x_i, d_i)) to each lower priority neighbor;
49     maintainAllocationInvariant;
50     if threshold == UB then
51         if TERMINATE received from parent or x_i is root then
52             SEND(TERMINATE, CurrentContext ∪ {(x_i, d_i)}) to each child;
53             Terminate execution;
54     SEND(COST,x_i, CurrentContext, LB, UB) to parent;
55 procedure maintainThresholdInvariant
56     if threshold < LB then
57         threshold ← LB;
58     if threshold > UB then
59         threshold ← UB;
60 procedure maintainAllocationInvariant
61     while threshold > δ(d_i) + Σ_{x_l∈Children} t(d_i, x_l) do
62         choose x_l ∈ Children where ub(d_i, x_l) > t(d_i, x_l) ;
63         increment t(d_i, x_l);
64     while threshold < δ(d_i) + Σ_{x_l∈Children} t(d_i, x_l) do
65         choose x_l ∈ Children where t(d_i, x_l) > lb(d_i, x_l) ;
66         decrement t(d_i, x_l);
67     SEND(THRESHOLD,t(d_i, x_l), CurrentContext) to each child x_l;
68 procedure maintainChildThresholdInvariant
69     forall x_l ∈ Children, d ∈ D_i do
70         while lb(d, x_l) > t(d, x_l) do
71             increment t(d, x_l);
72     foreach x_l ∈ Children, d ∈ D_i do
73         while t(d, x_l) > ub(d, x_l) do
74             decrement t(d, x_l);
```

**Algoritmo 2.2** – Algoritmo Adopt, parte 2

Quando $x_4$ recebe VALUE$(x_2, 0)$, adiciona este *label* ao seu *CurrentContext*, determina que seu LB é 1 e seu UB é 2 e envia esta informação para $x_2$ através de uma mensagem COST. Da mesma forma, $x_3$ recebe VALUE$(x_1, 0)$ e VALUE$(x_2, 0)$, adiciona estes *labels* ao seu *CurrentContext*, determina LB = 2 e UB = 4 e envia esta informação para $x_2$ através de mensagem COST. $x_2$, ao receber estas mensagens COST de $x_3$ e $x_4$ computa LB(0) = 4 e LB(1) = 2. Como LB do valor atual de $x_2$ (0) é maior que o LB de um outro valor do domínio de $x_2$ (1), então $x_2$ troca seu valor para 1 e envia mensagens VALUE$(x_2, 1)$ para os *childrens* $x_3$ e $x_4$ e uma mensgem COST$(x_2, \{(x_1, 0)\}, 2, \infty)$ para $x_1$. Ao receber VALUE$(x_2, 1)$, $x_4$ determina LB(0) = 2 e LB(1) = 0, e sendo assim, troca seu valor para 1 e envia COST$(x_4, \{(x_2, 1)\}, 0, 0)$ para $x_2$. Semelhantemente, $x_3$ processa VALUE$(x_2, 1)$ e determina LB(0) = 3 e LB(1) = 2, trocando seu valor para 1 e enviando COST$(x_3, \{(x_1, 0), (x_2, 1)\}, 2, 3)$. Então, $x_1$ recebe a mensagem COST previamente enviada por $x_2$, determina LB(0) = 2 e LB(1) = 0 e também altera seu valor para 1, enviando VALUE$(x_1, 1)$ para $x_2$. Supondo que $x_2$ tenha recebido esta mensagem VALUE de $x_1$ e atualizado seu *CurrentContext*, ao receber as mensagens COST de $x_3$ e $x_4$, irá computar seu LB = 0 e UB = 0, enviando COST$(x_2, \{(x_1, 1)\}, 0, 0)$ para $x_1$, que por sua vez, ao receber esta mensagem COST de $x_2$ também irá computar LB = 0 e UB = 0, satisfazendo a condição da linha 50 e enviando mensagem TERMINATE para o *children* $x_2$, sinalizando a finalização da execução. $x_2$ por sua vez, recebe TERMINATE, atualiza seu *CurrentContext* e envia TERMINATE para seus *childrens*.

Devido a simplicidade, este exemplo não demonstra o comportamento do *threshold* quando uma solução é revisitada. Para ilustrar este comportamento, considere-se os agentes e a DFST representados na fig. 2.6 e a seguinte situação: O agente $x_q$, de valor $d$, recebeu mensagem COST de seu *children* $x_p$ informando LB = 11. $x_1$ então armazenou $lb(d, x_p) = 11$ e atualizou $t(d, x_p)$ para 11. Agora, suponha-se que por um motivo qualquer, $x_q$ altera seu valor para $d'$, enviando VALUE$(x_q, d')$ para $x_p$. $x_p$, ao receber VALUE de $x_q$, verifica que houve alteração no valor de $x_q$, reseta suas variáveis locais (linhas 35 e 36) e reinicia o processo de busca. Porém, a sub-árvore de $x_p$ computa um LB e um

UB maiores que os observados quando $x_q = d$, e envia estes valores para $x_q$. Então, $x_q$ opta por retornar seu valor para $d$. Neste momento, além de $x_q$ enviar mensagem VALUE para $x_p$, também envia o valor de *threshold* reportado anteriormente, que foi armazenado em $t(d, x_p)$ e corresponde ao valor 11, para $x_p$ através de uma mensagem THRESHOLD. O agente $x_p$ assume então este valor como sendo seu *threshold* e subdivide este valor entre si e seus *childrens*. Através deste *threshold* informado e após algumas iterações, $x_p$ rapidamente chega no valor do domínio $d_j$ que anteriormente determinou este *threshold*, pois qualquer $d_j$ com LB($d_j$) inferior a este *threshold* é considerado, não havendo necessidade de explorar todos os valores do domínio para então concluir que é preciso aumentar o valor do *threshold*.




Fonte: Adaptado de Modi (2003b, p. 44)

**Figura 2.6** – Exemplo do comportamento do *threshold*

Após uma execução passo a passo, verifica-se que o número de mensagens trocadas é significativamente grande, pois muitas das mensagens enviadas são repetidas. Para solucionar este problema, Modi e Ali (2003c) apresentam uma simples estratégia para otimizar o algoritmo. Trata-se de somente enviar mensagens VALUE, THRESHOLD e COST se houve alteração no valor do agente, *threshold* dos *childrens* e custo atual, respectivamente. Porém, os autores chamam a atenção para um detalhe importante: se

esta estratégia for utilizada, o mecanismo de comunicação deve garantir que todas as mensagens sejam entregues, pois tendo em vista o fato do algoritmo ser assíncrono, a perda de uma mensagem pode fazer com que a execução pare.

# 3 FRAMEWORK DCSP

Desenvolvido e mantido pelo GRUPO DE INTELIGÊNCIA ARTIFICIAL DA FURB (2005), o *framework* dynDCSP consiste de um conjunto de classes (desenvolvidas na linguagem Java) que implementam os algoritmos de DCSP apresentados por Yokoo (2001).

A arquitetura do *framework* dynDCSP apresenta quatro camadas distintas: de comunicação, de interface ao usuário, de implementação dos algoritmos e de gerenciamento de execução. A camada de comunicação disponibiliza uma interface de comunicação que permite ao desenvolvedor implementar a infraestrutura de comunicação entre os agentes da maneira que achar conveniente. Atualmente o *framework* oferece uma implementação de comunicação entre agentes através da infraestrutura *Simple Agent Communication Infrastructure* (SACI) (HÜBNER; SICHMAN, 2000). A camada de interface ao usuário é responsável pelo interfaceamento de informações (normalmente saída de dados) entre os agentes e o usuário do *framework*. O *framework* dynDCSP já oferece implementações de interface ao usuário em modo texto e em modo janela. Já a camada de implementação dos algoritmos, como o próprio nome sugere, oferece recursos (classes) para a implementação dos algoritmos propostos por Yokoo (2001) e para futuras implementações de algoritmos para resolução de DCSP. Por fim, a camada de gerenciamento de execução, disponibiliza um mecanismo para criar, inicializar e gerenciar a execução (*start*/*stop*) de quantos agentes forem necessários para o algoritmo de resolução escolhido, além de um mecanismo de agendamento de execuções.

A fig. 3.1 apresenta o diagrama de pacotes do *framework* dynDCSP. O pacote `dcsp.comm` contém a camada de comunicação, `dcsp.io` contém a camada de interface ao usuário, `dcsp.alg` a camada de implementação dos algoritmos e `dcsp.main` a camada de gerenciamento de execução.

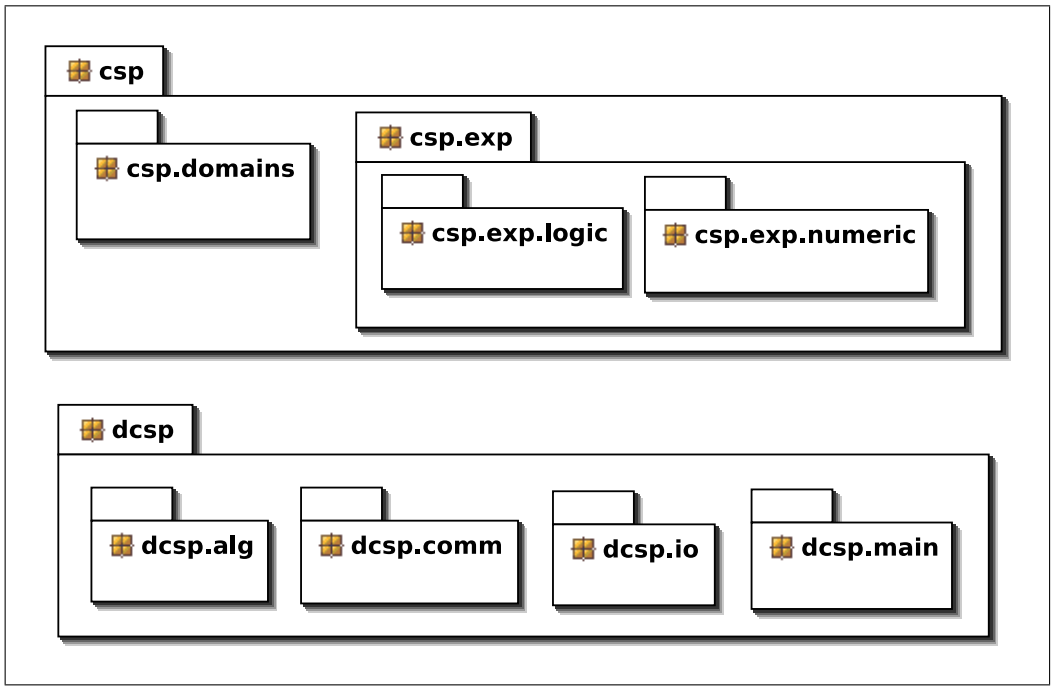



**Figura 3.1** – Pacotes disponíveis no *framework* dynDCSP

O pacote `csp` contém classes que implementam os componentes de um CSP: variáveis, domínios e restrições (sendo esta última representada por expressões lógicas ou aritméticas). As classes deste pacote são apresentadas na fig. 3.2. A classe `Variable` corresponde a uma variável do problema. À esta variável está associado um domínio, representado pela classe `Domain`. O valor que uma variável assume é representado pela classe `Value`. Uma solução (parcial ou não), formada por um conjunto de *labels* (variável-valor) é representado por `Assignment`. A classe `Constraint` representa uma restrição, e é formada por expressões (lógicas/aritméticas) representadas pela classe `Expression` (o *framework* oferece implementações de diversas expressões, lógicas ou aritméticas, como por exemplo: adição, subtração, *e* lógico, *ou* lógico, dentre outras). Um CSP, representado pela classe `CSP` é composto por variáveis (que já possuem domínios associados) e por restrições.

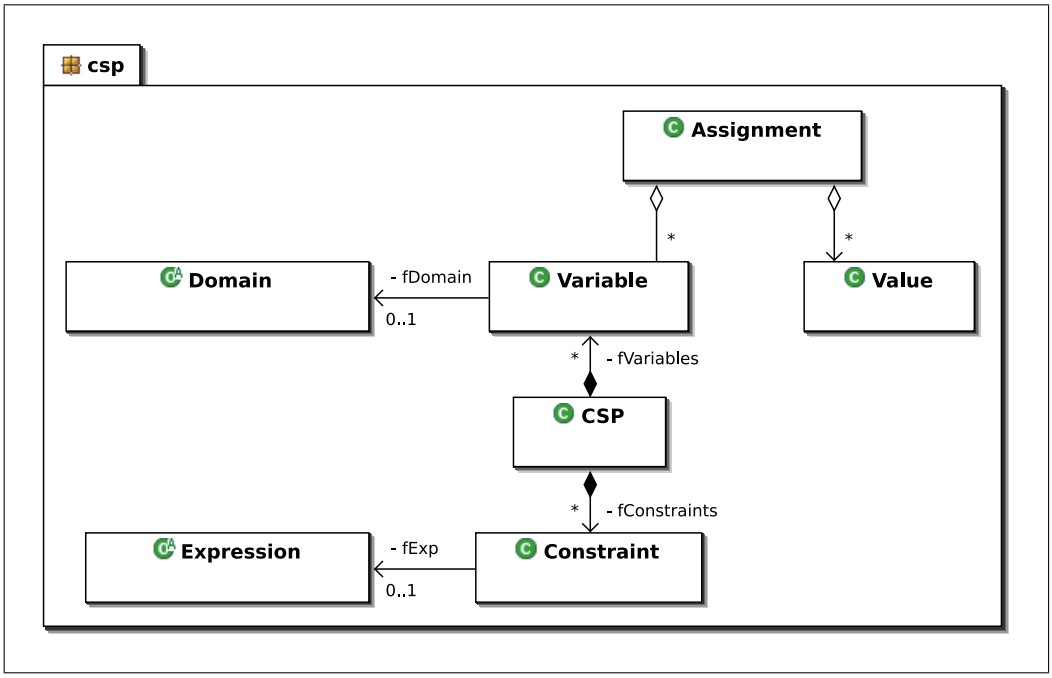



**Figura 3.2** – Classes do pacote `csp`

O pacote `dcsp.comm`, visualizado na fig. 3.3, disponibiliza uma interface (`Communication`), que deve ser implementada pela infraestrutura de comunicação que se queira utilizar. Esta interface disponibiliza, por exemplo, métodos para criação de mensagens (`createMessage()`) e envio de mensagens que geram resposta (`ask()`) e sem resposta (`tell()`). Para recepção de mensagens, a infraestrutura de comunicação utilizada deve permitir o registro de *callbacks*, e estes devem ser registrados através do método `addMsgHandlers()`. O *framework* já disponibiliza uma implementação que realiza a comunicação através do SACI na classe `SaciCommunication`. A classe de comunicação atua como um "middleware" no processo de comunicação. Quando um agente necessita enviar uma mensagem, o agente entrega esta mensagem ao objeto de comunicação que está associado a ele, que por sua vez irá entregar a mensagem ao objeto de comunicação associado ao agente de destino e este repassa ao agente propriamente dito. Este processo é ilustrado na fig. 3.4.

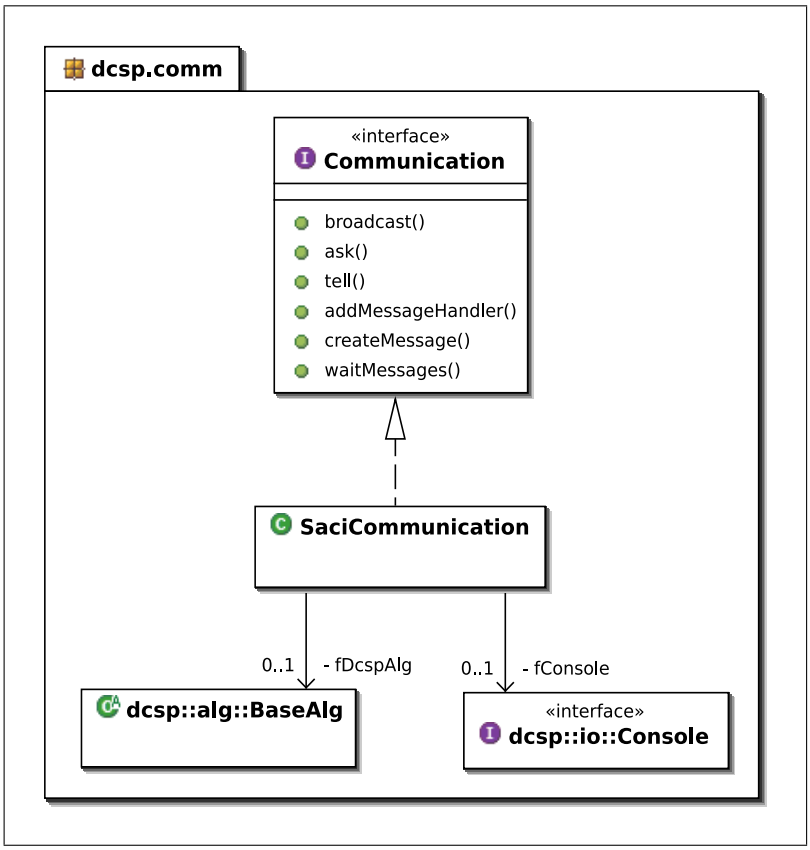



**Figura 3.3** – Classes do pacote `dcsp.comm`

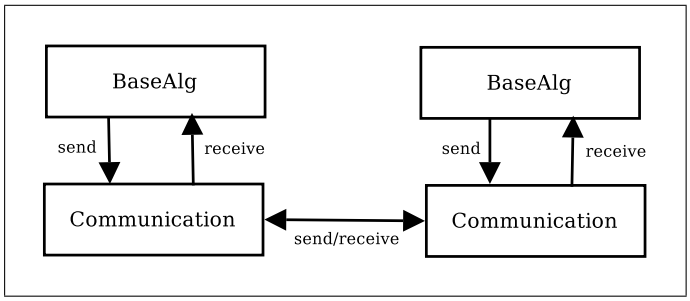



**Figura 3.4** – Objeto `Communication` e a comunicação entre agentes

Já o pacote `dcsp.io`, visto na fig. 3.5, apresenta uma interface (`Console`) que deve ser implementada pelo tipo de interface ao usuário que se queira utilizar. Esta interface contém métodos para imprimir valores (`print()` e `println()`), para exibir excessões (`showException()`) e para exibir o valor do agente a qual está associada (`showCurrentValue()`). O *framework* disponibiliza três implementações desta interface: `TextConcole` implementa interface com o usuário em modo texto; `GUIConsole` o faz em modo gráfico; e `NullConsole` não faz interface com o usuário.

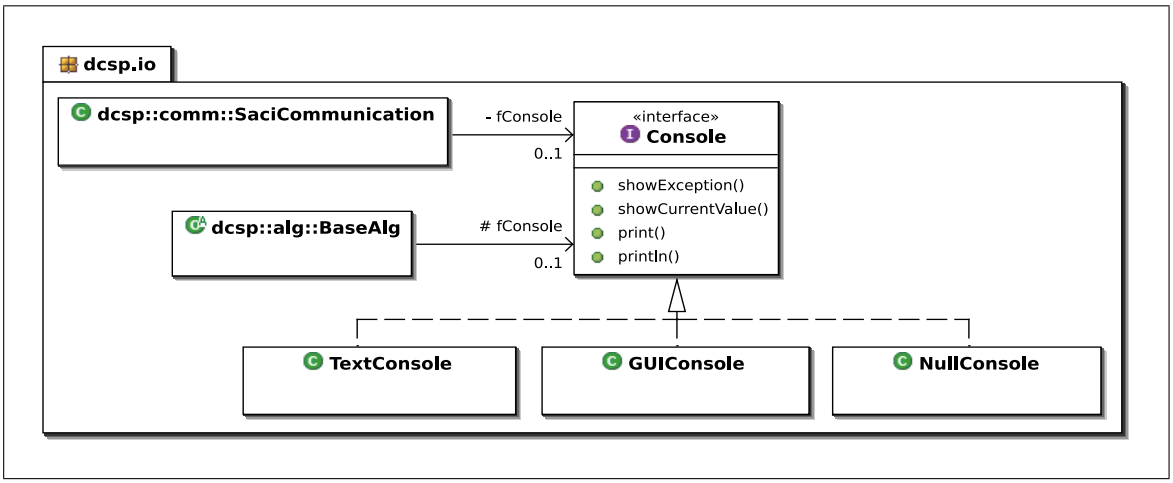



**Figura 3.5** – Classes do pacote `dcsp.io`

As classes de implementação dos algoritmos, contidas no pacote `dcsp.alg`, são apresentadas no diagrama da fig. 3.6. Conforme mencionado, o *framework* implementa os três algoritmos para resolução de DCSPs propostos por Yokoo (2001). A classe `BaseAlg` representa um agente (do algoritmo) e contém atributos e métodos comuns aos três algoritmos, como por exemplo: a variável do CSP a qual corresponde o agente e seu valor, o recebimento dos valores de domínio e das restrições fornecidos pelo agente gerenciador e verificação se uma solução parcial é consistente com as restrições do agente, dentre outros. Cada um dos três algoritmos é implementado em uma das demais classes, sendo: *Asynchronous Backtracking* na classe `AB`, *Asynchronous Weak-Commitment Search* na classe `AWT` e *Distributed Breakout* na classe `DB`.

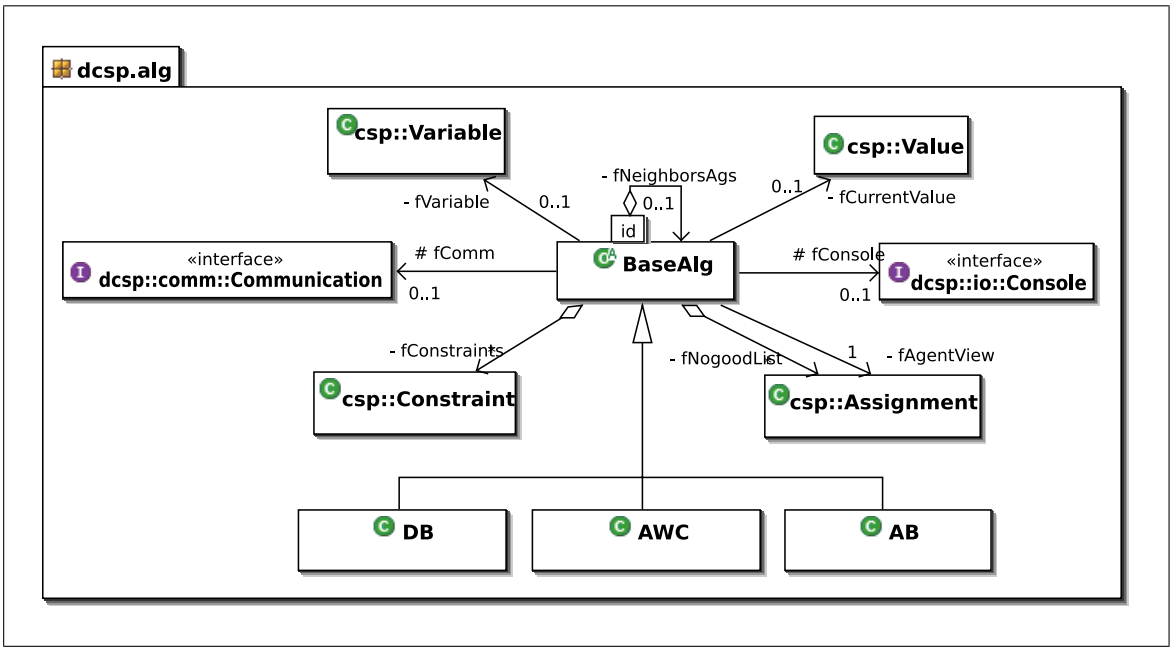



**Figura 3.6** – Classes do pacote `dcsp.alg`

Por fim, o diagrama da fig. 3.7 apresenta o pacote `dcsp.main`, responsável pelo gerenciamento da execução do(s) algoritmo(s). Este gerenciamento é feito através de um agendador e um agente gerenciador. O agendador (`Scheduler`) consiste de uma classe que recebe as requisições de execução, armazena-as em uma fila e dá início a execução quando existem *hosts* disponíveis. Assim que um *host* estiver disponível, o agendador instancia um agente gerenciador (`BaseManager`), que é responsável por instanciar e inicializar quantos agentes (do algoritmo escolhido) forem necessários. Para os algoritmos propostos por Yokoo (2001), o agente gerenciador tem o papel de instanciar um agente do algoritmo selecionado para cada variável do CSP, enviar seu domínio, suas restrições e dar início a execução do algoritmo. Devido à utilização da infraestrutura SACI para comunicação entre os agentes, há um agente gerenciador que utiliza esta infraestrutura para realizar seu papel (`SaciManager`). As classes `pdpw` e `pdpwGUI` correspondem a interface texto e gráfica, respectivamente, do *framework* dynDCSP. A classe `ManagerGUI` destina-se a exibição, ao final da execução, de informações sobre o DCSP (por exemplo: valor das variáveis e tempo de execução).

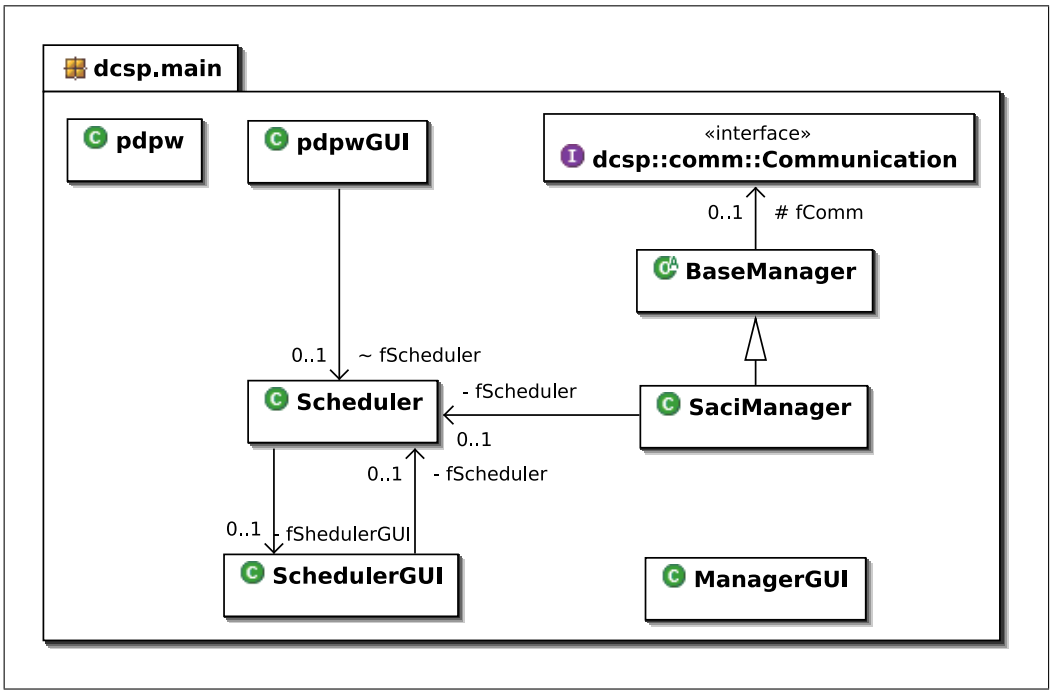



**Figura 3.7** – Classes do pacote `dcsp.main`

## 3.1 UTILIZAÇÃO DO FRAMEWORK DYNDCSP

Para utilização do *framework* é necessário primeiramente especificar um CSP a partir da classe `CSP` e em seguida adicionar o nome da classe *factory* deste CSP no arquivo de configuração utilizado pela *framework*. A seguir, é apresentado um exemplo de utilização do *framework* para solucionar o CSP visto na fig. 2.1. Para efeitos de implementação, este problema será chamado de `HelloCSP`.

Primeiramente, é necessário criar uma classe que representa este CSP. Esta classe deve estender a classe `CSP` e definir as variáveis, domínios e restrições do problema. O quadro 3.1 apresenta a classe criada para o problema da fig. 2.1. No método construtor desta classe deve-se definir os domínios, as variáveis e as restrições. Na linha 8 está sendo criado um domínio vazio composto de valores inteiros (`IntegerDomain`) de nome “oneVl”, e a linha 9 adiciona o valor 2 a este domínio (o mesmo processo é realizado nas linhas 10, 11 e 12 para criação de um outro dimínio contendo os valores 1 e 2). Em seguida, deve-se definir as variáveis do problema. Para isso, instanciam-se tantos objetos de `Variable` quantas forem as variáveis do problema, sendo que se deve atribuir um domínio para cada objeto `Variable` e adicioná-la ao CSP. A linha 14 apresenta a criação da variável $x_1$,

na linha 15 atribui-se um domínio para a variável e por último a linha 16 adiciona esta variável ao CSP. Por fim, deve-se definir as restrições entre as variáveis do problema. A linha 24 apresenta a criação da restrição $x_1 \neq x_3$ e na linha 27 esta restrição é adicionada ao CSP.

```
1 package helloCSP;

3 public class HelloCSP extends CSP {
4   /** Creates a new instance of HelloCSP */
5   public HelloCSP() {
6     super("HelloCSP");
7     // domains
8     IntegerDomain e1 = new IntegerDomain("oneVl");
9     e1.addValue(new Value(new Integer(2)));
10    IntegerDomain e2 = new IntegerDomain("twoVl");
11    e2.addValue(new Value(new Integer(1)));
12    e2.addValue(new Value(new Integer(2)));
13    // variables
14    Variable x1 = new Variable("x1");
15    x1.setDomain(e2);
16    addVariable(x1);
17    Variable x2 = new Variable("x2");
18    x2.setDomain(e1);
19    addVariable(x2);
20    Variable x3 = new Variable("x3");
21    x3.setDomain(e2);
22    addVariable(x3);
23    // constraints
24    Constraint c1 = new Constraint(new DifferentExpression(
25                    new VariableExpression(x1),
26                    new VariableExpression(x3)));
27    addConstraint(c1);

29    Constraint c2 = new Constraint(new DifferentExpression(
30                    new VariableExpression(x2),
31                    new VariableExpression(x3)));
32    addConstraint(c2);
33  }
34 }
```

**Quadro 3.1** – `CSP` para o problema `HelloCSP`

Após a criação da classe que representa o CSP, deve-se criar uma classe *factory*, que será responsável pela instanciação do objeto CSP (no caso, `HelloCSP`). Esta classe deve implementar a interfae `CSPFactory`, que define métodos para a criação (instanciação) do CSP (inclusive a partir de parâmetros do problema). A classe *factory* para o problema

`HelloCSP`, denominada `HelloCSPFactory`, é apresentada no quadro 3.2. O principal método desta classe é o `createCSP()`, visto na linha 4, que é responsável por instanciar um objeto de `HelloCSP`.

```
1 package helloCSP;

3 public class HelloCSPFactory implements CSPFactory {
4   public CSP createCSP() {
5     return new HelloCSP();
6   }
7   public CSP createCSP(java.util.Properties parameters) {
8     return createCSP();
9   }
10  public String getLabel() {
11  return "HelloCSP";
12  }
13  public Collection getParameteres() {
14    return new ArrayList(0);
15  }
16 }
```

**Quadro 3.2** – *Factory* para o problema `HelloCSP`

Para tornar o CSP definido apto para ser resolvido pelo *framework*, deve-se alterar o arquivo de configuração do *framework*. O conteúdo deste arquivo é apresentado no quadro 3.3 (página 46). Pode-se notar que este arquivo apresenta duas seções: `algorithms` e `problems`. A seção `algorithms` especifica informações relativas aos algoritmos disponíveis no *framework* para resolução de CSP. Já a seção `problems` contém os problemas aptos a serem resolvidos pelo *framework*. É nesta seção que deve ser adicionado o *factory* `HelloCSPFactory`, como se pode observar na linha 12.

Criadas as classes do CSP e alterado o arquivo de configuração, o próximo passo consiste em executar o *framework*. Para isto existem *scripts* `ant` (APACHE SOFTWARE FOUNDATION, 2005) no diretório `bin` do *framework* que devem ser executados a partir de um `shell`. Inicialmente é preciso executar o SACI através do comando `ant saci`, para então executar o *framework* através do comando `ant pdpw`. Assim que o *framework* for executado, é apresentada a interface gráfica vista na fig. 3.8.

```
1 # This file contains the problems and
2 # algoritms available for DCSP.
3 # Each value is the java class name of either
4 # the algorithm or the problem factory.

6 +algorithms
7     dcsp.alg.AWC
8     dcsp.alg.AB
9     dcsp.alg.DB

11 +problems
12     helloCSP.HelloCSPFactory
13     mSquare.MagicSquareCSPFactory
14     nqueens.NQueensCSPFactory
15     zebra.ZebraCSPFactory
```

**Quadro 3.3** – `pdpw.conf`: arquivo de configuração do *framework*

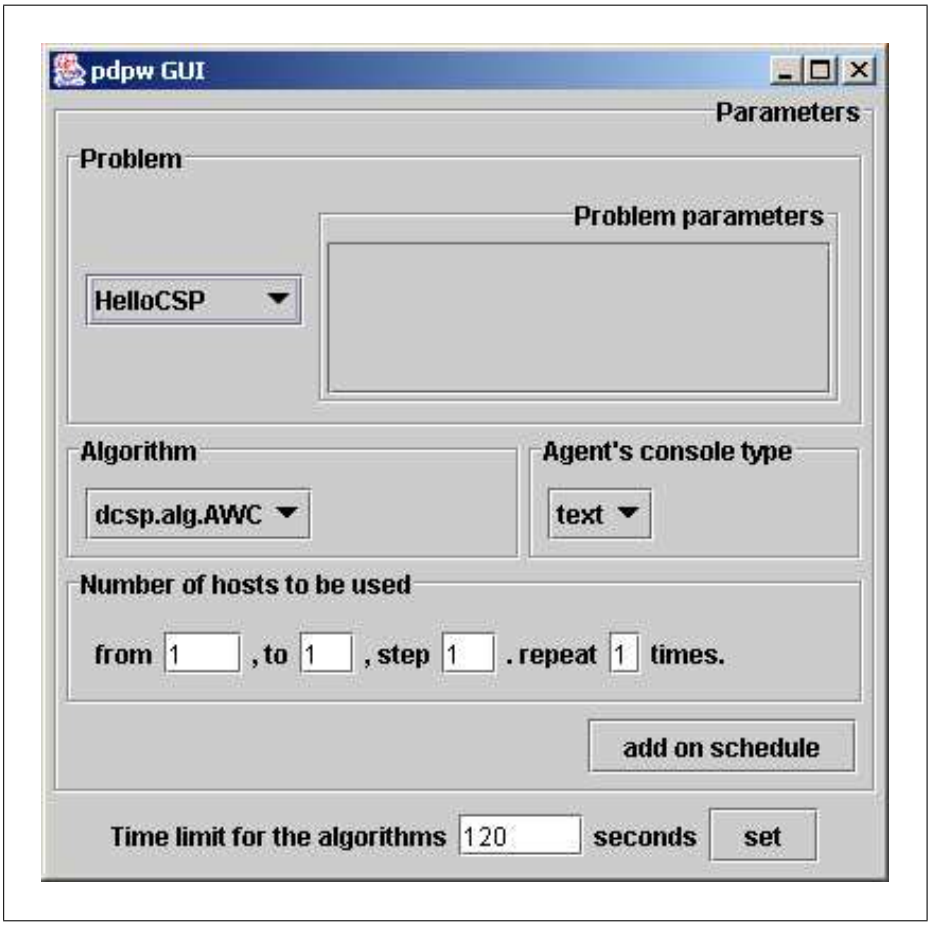



**Figura 3.8** – Interface gráfica do *framework* dynDCSP

Nesta interface é possível configurar todos os parâmetros necessários para a solução do problema utilizando determinado algoritmo. No painel *Problem* deve-se selecionar o

problema que se quer resolver, neste caso é o `HelloCSP` (criado e configurado nas etapas descritas anteriormente). Caso estivessem sido definidos parâmetros na classe *factory*, estes parâmetros iriam aparecer no painel *Problem parameters*. No painel *Algorithm* deve-se escolher qual é o algoritmo que deve ser utilizado para solucionar o problema (AB, AWC ou DB). Já no painel *Agent's console type* seleciona-se o tipo de interface ao usuário de cada agente. O painel *Numbers of hosts to be used* permite especificar a quantidade de *hosts* que se quer utilizar na execução e também quantas vezes uma mesma execução deve ser repetida. A quantidade de *hosts* deve ser configurada seguindo-se a expressão apresentada na equação (3.1).

$$for(i = minNumberOfHosts; i < maxNumberOfHosts; i+ = step) \qquad (3.1)$$

O campo `from` (fig. 3.8) corresponde a $minNumberOfHosts$, o campo `to` a $maxNumberOfHosts$ e `step` a *step*. Esta definição faz variar o número de *hosts* utilizados na execução de acordo com o valor de *step*. Por exemplo, ao preencher estes campos com os valores 1, 3 e 1, respectivamente, o *framework* irá executar três vezes o algoritmo escolhido, sendo a primeira vez com um *host*, a segunda vez com dois *hosts* e a terceira vez com três *hosts*. O campo `repeat` especifica quantas vezes será repetida a execução em cada variação do número de *hosts*. Se no exemplo anterior `repeat` estivesse com valor 2, o *framework* iria executar o algoritmo duas vezes com um *host*, duas vezes com dois *hosts* e duas vezes com três *hosts*. Por fim, o campo *Time limit for the algoritm* permite especificar o tempo que cada execução terá para execução. Após preenchimento de todos estes campos, basta pressionar o botão *add on schedule* para adicionar a execução ao agendador (*Scheduler*). A interface gráfica do agendador é mostrada na fig. 3.9.

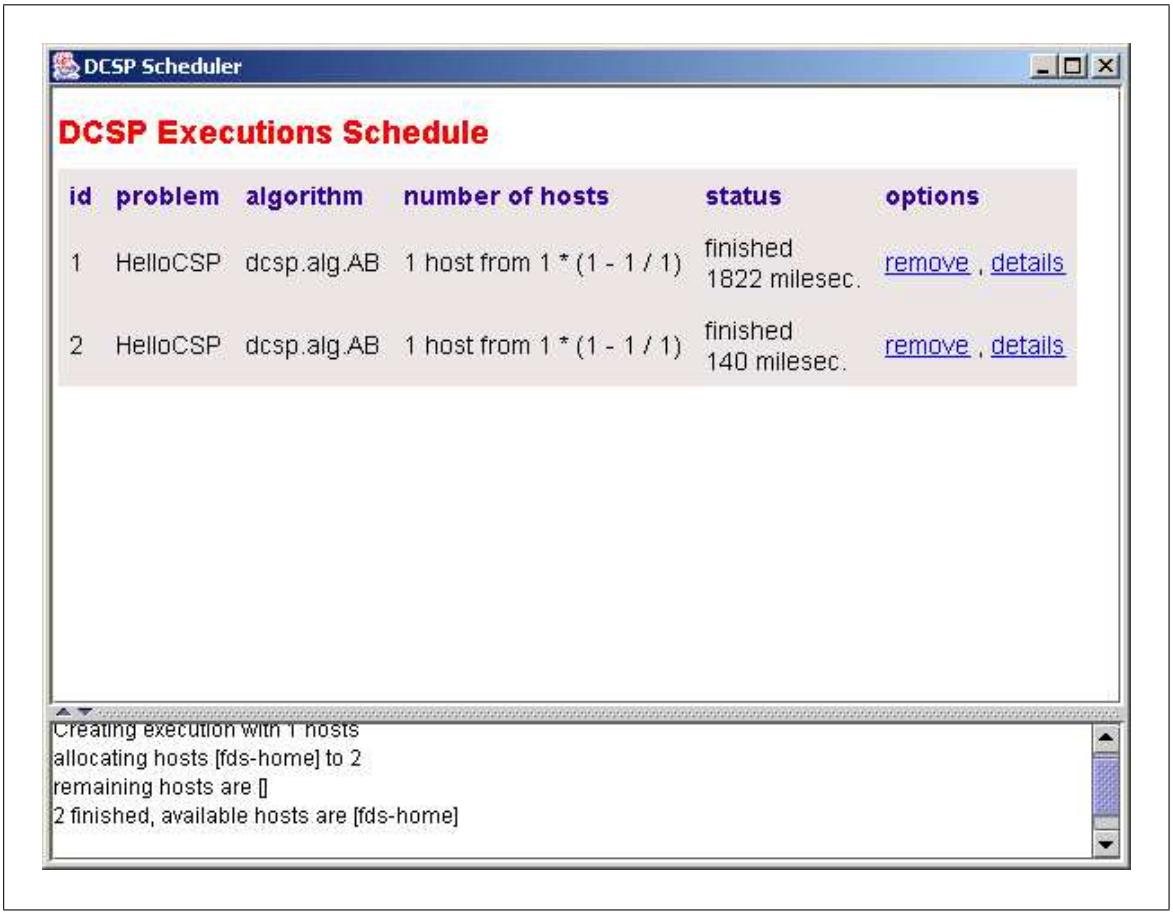



**Figura 3.9** – Interface gráfica do *scheduler*

O agendador apresenta todas as execuções programadas, em curso e finalizadas do *framework*, sendo possível para cada execução agendada, cancelá-la (se estiver programada ou em curso) e visualizar os detalhes da execução quando esta estiver finalizada. Estes detalhes são vistos através do `ManagerGUI`, onde são apresentados os valores encontrados para as variáveis (solução do problema), bem como o tempo de execução e um sumário sobre o problema solucinado. A interface gráfica do `ManagerGUI` é vista na fig. 3.10, contendo detalhes da execução do algoritmo AB para o problema `HelloCSP`.

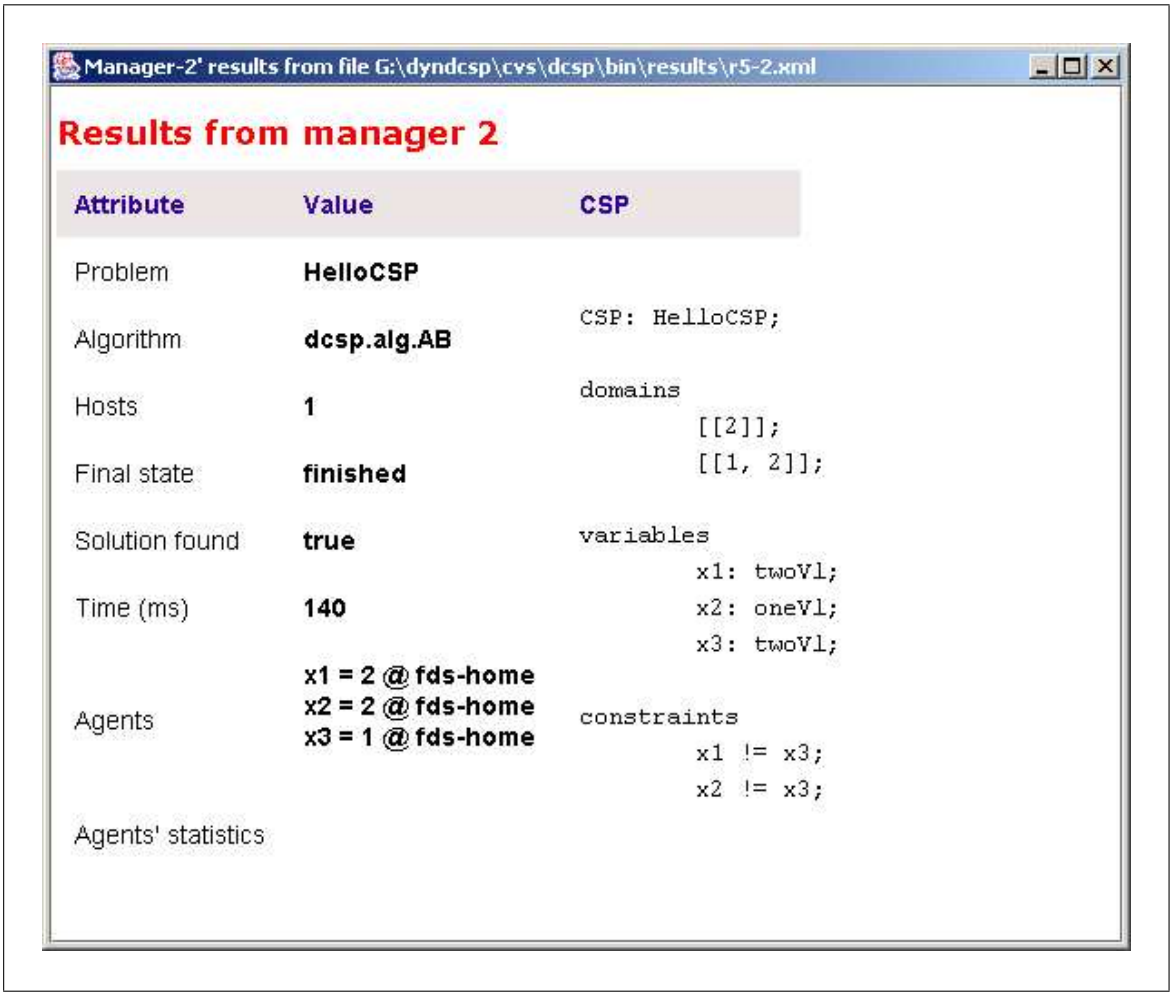



**Figura 3.10** – Interface gráfica do `ManagerGUI` com detalhes da execução

Até o momento, o *framework* dynDCSP já foi utilizado em dois trabalhos de conclusão de curso. No primeito, Tralamazza (2004) propôs alterações no algoritmo AWC (adaptação da heurística do valor menos utilizado; ordenação das restrições; e não armazenamento de *nogoods*). No segundo, Bruns (2004) adaptou o *framework* para suportar restrições dinâmicas e utilizou o algoritmo AB para sincronismo de semáforos de uma malha viária.

# 4 DESENVOLVIMENTO DO ALGORITMO ADOPT

As seções seguintes descrevem a especificação e a implementação do algoritmo Adopt a partir de extensões do *framework* dynDCSP.

## 4.1 REQUISITOS PRINCIPAIS

Os principais requisitos deste trabalho são:

a) implementação de otimização de DCOP através do algoritmo Adopt;

b) validação do algoritmo Adopt;

c) especificação do problema de modelagem molecular como um COP e posteriormente como um DCOP para verificar a viabilidade de resolução através do algoritmo Adopt;

d) portabilidade, para que seja possível sua execução em qualquer plataforma.

## 4.2 ESPECIFICAÇÃO

Em função do *framework* dynDCSP já oferecer recursos para especificar um CSP bem como para, a partir do CSP, determinar um DCSP, criando, distribuindo e gerenciando os agentes, optou-se por especificar e implementar o algoritmo Adopt para solucionar DCOP como uma extensão do *framework*. Para tanto, algumas classes precisam ser modificadas e outras incluídas.

Inicialmente, foi adicionado ao *framework* a capacidade de especificar um COP. Para tanto, foi preciso criar no pacote `csp` uma classe que estenda um CSP (para manter suas propriedades) e que permita a especificação de uma ou mais funções de custo. Esta alteração é apresentada no diagrama de classes da fig. 4.1, sendo que a classe `COP` estende `CSP` e possui funções de custo, representadas pela classe `CostFunction` que por sua vez contém uma lista de `Expression` (`costFunction`) que correspondem a função de custo. O resultado final da avaliação da função de custo através do método `eval()` corresponde ao somatório de todas as `Expression`. Também no pacote `csp` foi adicionada a classe

`Association`, que corresponde a um *label* (associação de uma variável a um valor, $\{x_i = v_i\}$).

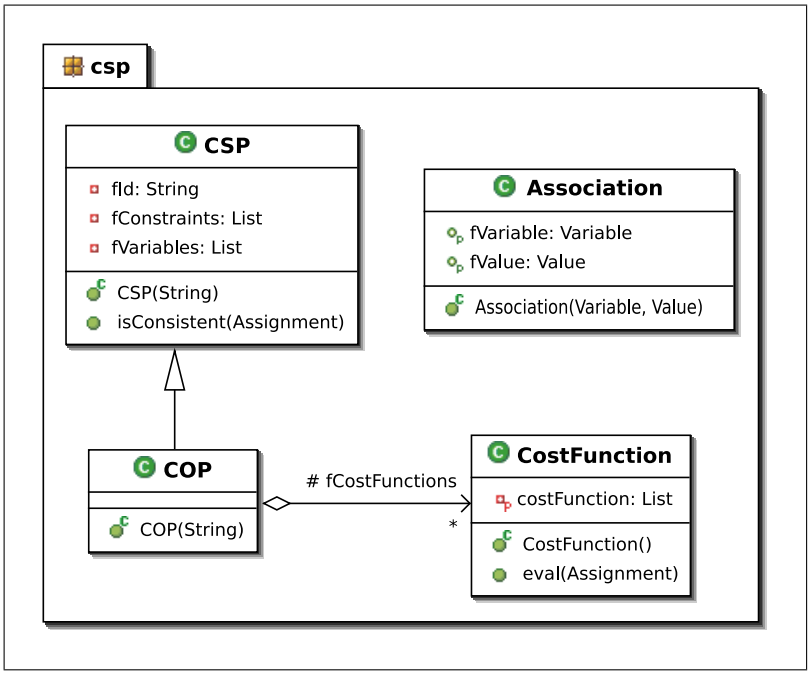



**Figura 4.1** – Alterações no pacote `csp`

O pacote `dcsp.alg` foi complementado com o algoritmo Adopt. Para tanto, necessitou-se fatorar as classes existentes neste pacote de forma a separar as propriedades e métodos de algoritmos de DCSP e DCOP. Isto foi realizado definindo uma superclasse genérica, denominada `BaseAlg`, para os dois tipos de algoritmos. Esta classe contém os atributos comuns aos dois tipos de algoritmos, como por exemplo, a variável do agente e seu valor atual, além de registrar *callbacks* utilizadas pelo agente gerenciador. Especializando esta classe, tem-se `DCSPBaseAlg` e `DCOPBaseAlg`. Cada uma destas classes também reúne atributos e métodos correspondentes ao tipo de problema que se propõe a resolver. Por fim, cada um dos algoritmos deve especializar seu respectivo `BaseAlg`. O diagrama de classe para o pacote `dcsp.alg` após estas fatorações é apresentado na fig. 4.2. Na seqüência, são apresentadas em detalhes cada uma destas novas classes criadas.

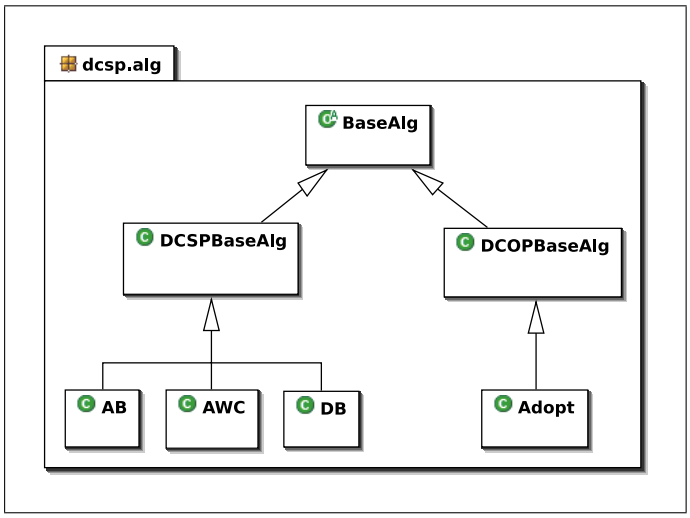



**Figura 4.2** – Alterações no pacote `dcsp.alg`

A classe `BaseAlg` é apresentada na fig. 4.3. Esta classe reúne os atributos comuns à DCSP e DCOP, sendo: a variável do agente e seu valor atual e um *flag* (`inIdleState`) que indica se o agente está em um estado *idle* (para fins de computação do término pelo agente gerenciador). Agrega uma lista de restrições e possui associado um objeto de comunicação e de interface ao usuário. Dentre os métodos desta classe, estão um método para registrar as *callbacks* de mensagens comuns a todos os dois tipos de algoritmos (`addMsgHandlers()` que registra, por exemplo, *callbacks* para receber mensagens com o domínio e as restrições do problema), e um método abstrato responsável por iniciar a execução do algoritmo (`start()`).

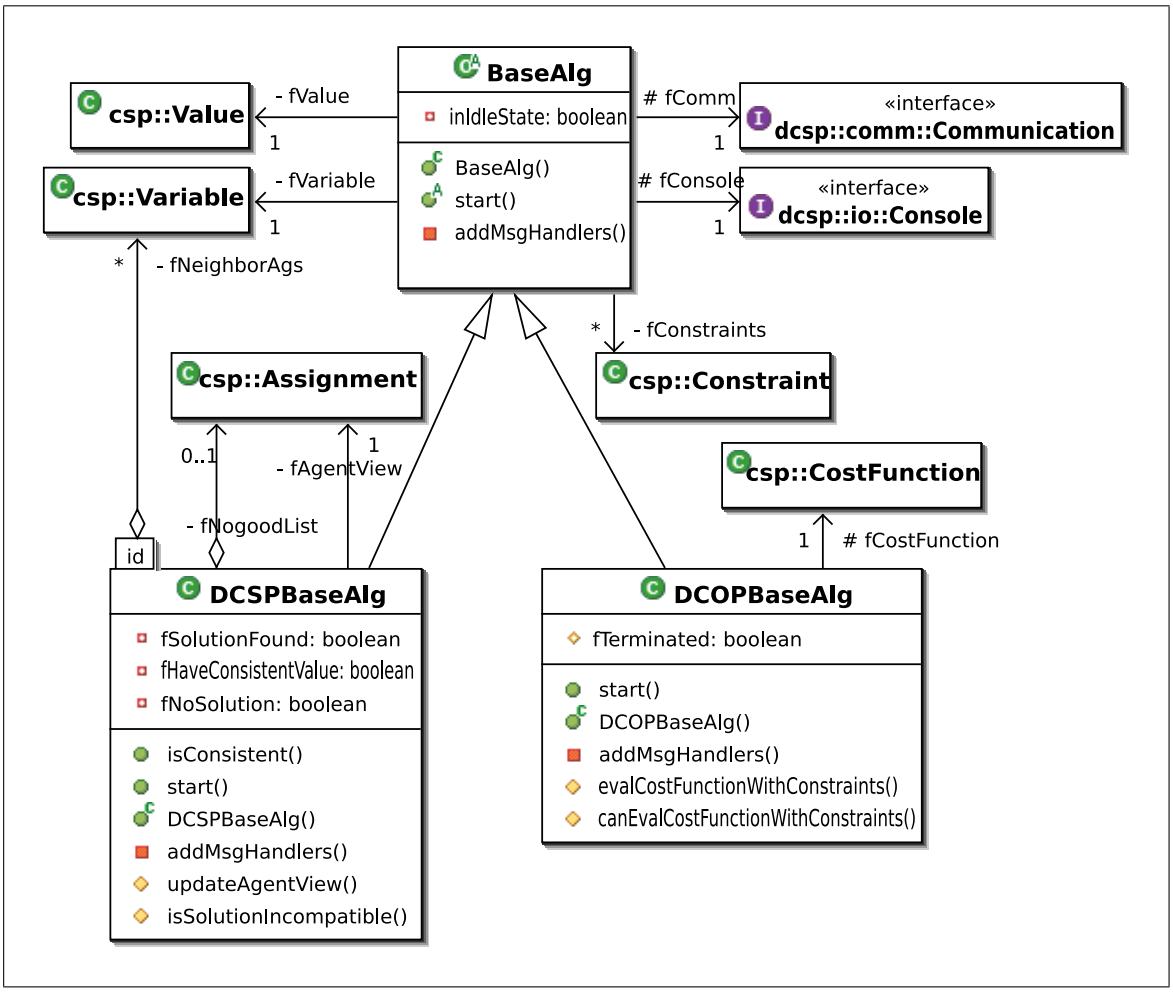



**Figura 4.3** – Detalhes das classes do pacote `dcsp.alg`

A classe `DCSPBaseAlg`, também vista na fig. 4.3, compõe-se dos atributos e métodos comuns aos algoritmos para solução de DCSP, em especial, os algoritmos propostos por Yokoo (2001) que o *framework* dynDCSP implementa. Os *flags* `fNoSolution`, `fSolutionFound` e `fHaveConsistentValue` indicam, respectivamente, se a execução do algoritmo não encontrou uma solução para o problema, se encontrou uma solução, e se o valor atual do agente é consistente. A esta classe está associado o *agentview* (que corresponde a um conjunto que contém os *labels* dos agentes *neighbors*) e agrega uma lista de *nogoods* que armazena os diversos *nogoods* (soluções parciais inconsistêntes) encontrados durante a execução, além de agregar também uma lista de variáveis que representam os agentes *neighbors*. Quanto a métodos, esta classe dispõe de `addMsgHandlers()`, para registrar *callbacks* para tratar de mensagens que informam, por exemplo, os *neighbors*

do agente. O método `isSolutionIncompatible()` verifica, dada uma solução parcial, se a mesma contém um *nogood* que invalida a solução. Já o método `updateAgentView()` destina-se a atualizar o *label* de uma determinada variável já contida no *agent view*.

Já a classe `DCOPBaseAlg`, apresentada também na fig. 4.3, contém a função de custo (um atributo comum a todos os DCOPs), e métodos relacionados a algoritmos para solução de DCOPs. Além disto, possui um *flag* para sinalizar o término da computação pelo próprio agente (`fTerminated`). Em relação a métodos, destacam-se na classe `DCOPBaseAlg` o método de registro de *callbacks* `addMsgHandlers()`, que registra *callbacks* para recepção de mensagem que informa a função de custo. O método `evalCostFunctionWithConstraints()` é responsável por avaliar a função de custo considerando as restrições definidas (se uma restrição é violada, tem-se $+\infty$ como resultado da avaliação), e o método `canEvalCostFunctionWithConstraints()` verifica se é possível avaliar a função de custo, ou seja, se o agente possui *labels* em seu $CurrentContext$ para todas as variáveis acima de si, de acordo com a DFST.

Por fim, foi especificada a classe Adopt (fig. 4.4). Esta classe apresenta os atributos e métodos necessários para a implementação do algoritmo Adopt. Na sequência são descritos estes atributos e métodos (as linhas mencionadas na descrição referem-se ao algoritmo Adopt apresentado nos algoritmos 2.1 e 2.2).

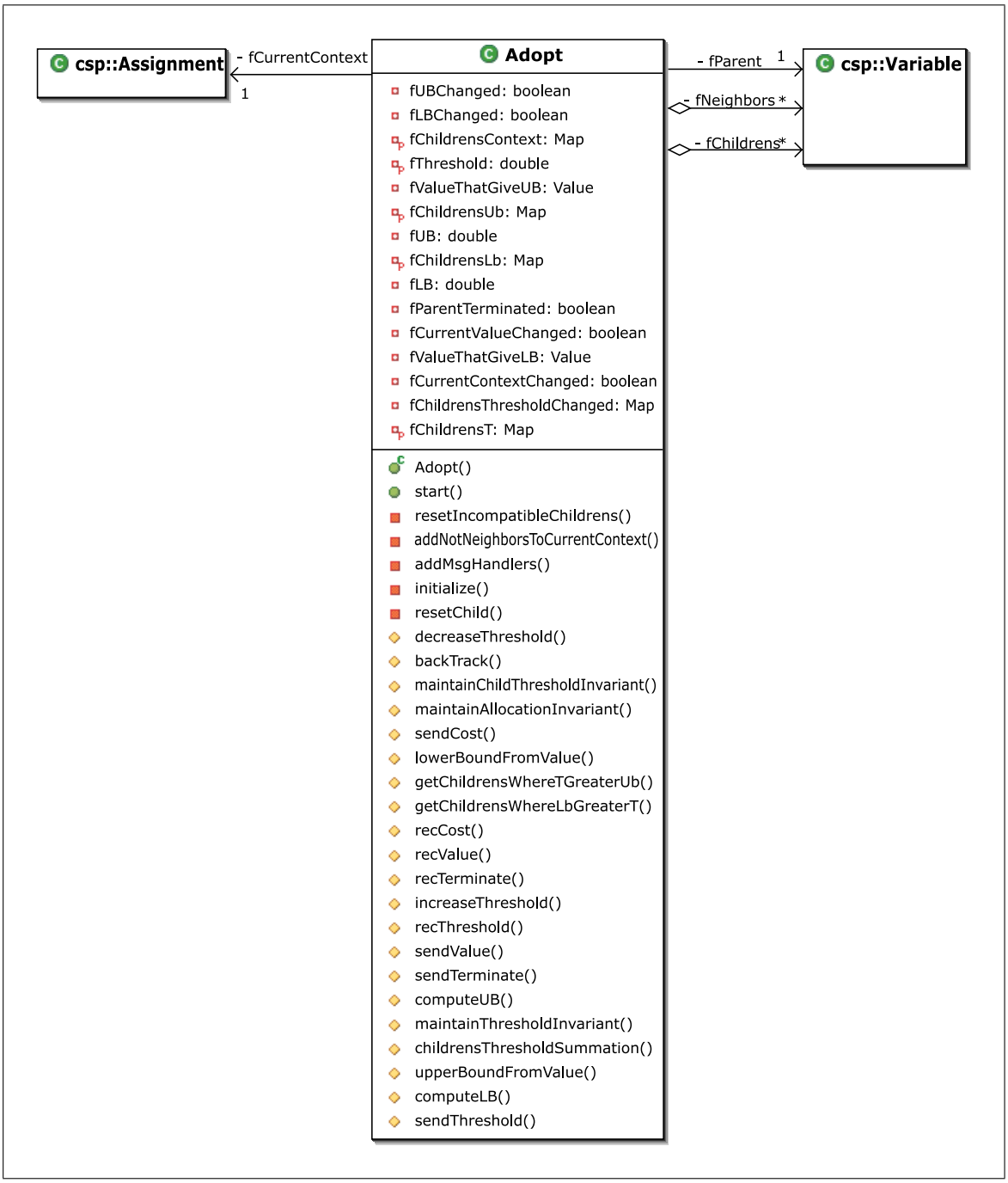



**Figura 4.4** – Classe `Adopt`

Os atributos `fChildrensLb`, `fChildrensUb`, `fChildrensT` e `fChildrensContext` correspondem respectivamente as variáveis $lb(d, x_l)$, $ub(d, x_l)$, $t(d, x_l)$ e $context(d,x_l)$. `fLB` e `fUB` são o LB e UB do agente (variável). `fValueThatGiveLB` e `fValueThatGiveUB` os valores do domínio que proporcionam LB e UB. Há uma associação com a classe `Assignment` para representar a variável *CurrentContext*. O atributo `fThreshold` representa a variável *threshold*. Para a otimização na troca de mensagens, foram cria-

dos: `fLBChanged`, `fUBChanged` e `fCurrentContextChanged` que sinalizam se houve alteração nos valores de LB, UB ou *CurrentContext* para só então enviar mensagens COST; `fChildrensThresholdChanged` que indica se houve alteração nos valores de *threshold* dos *childrens* para então enviar mensagens THRESHOLD; e `fCurrentValueChanged` indica se houve alteração no valor atual da variável para então enviar mensagens VALUE. Além disto, a classe `Adopt` apresenta uma associação com `Variable` para representar o agente (variável) *parent*, bem como duas agregações, para representar os *childrens* e os *neighbors*.

A classe `Adopt` apresenta métodos *callbacks* para recebimento das mensagens THRESHOLD, TERMINATE, VALUE e COST que são `recThreshold()`, `recTerminate()`, `recValue()` e `recCost()` respectivamente, sendo estas *callbacks* registradas através do método `addMsgHandlers()`. Também foram criados métodos para envio de mensagens, sendo eles: `sendThreshold()`, `sendTerminate()`, `sendValue()` e `sendCost()`. Cada uma das *procedures* presentes no algoritmo foram mapeadas para um método, gerando: `initialize()`, `backTrack()`, `maintainThresholdInvariant()`, `maintainAllocationInvariant()` e `maintainChildThresholdInvariant()`. Os valores de LB e UB e também os valores do domínio da variável que proporcionam LB e UB são calculados através dos métodos `computeLB()` e `computeUB()`. A linha 5 foi transformada no método `resetChild()` e as linhas 23 e 24 no método `resetIncompatibleChildrens()`. O somatório dos *thresholds* dos *childrens*, utilizado nas linhas 61 e 64 é obtido através do método `childrensThresholdSummation()`. Para as linhas 69-70 e 72-73 foram criados os métodos `getChildrensWhereLbGreaterT()` e `getChildrensWhereTGreaterUb()` respectivamente. Por fim, foram criados métodos para, dado um determinado valor do domínio *d*, calcular LB(d) e UB(d), sendo eles, `lowerBoundFromValue()` e `upperBoundFromValue()`.

A fig. 4.5 apresenta o diagrama de sequência ilustrando a troca de mensagens entre dois agentes do algoritmo Adopt. Nota-se que para uma mensagem de um agente Adopt chegar a outro agente Adopt, a mesma deve passar pelo objeto de comunicação.

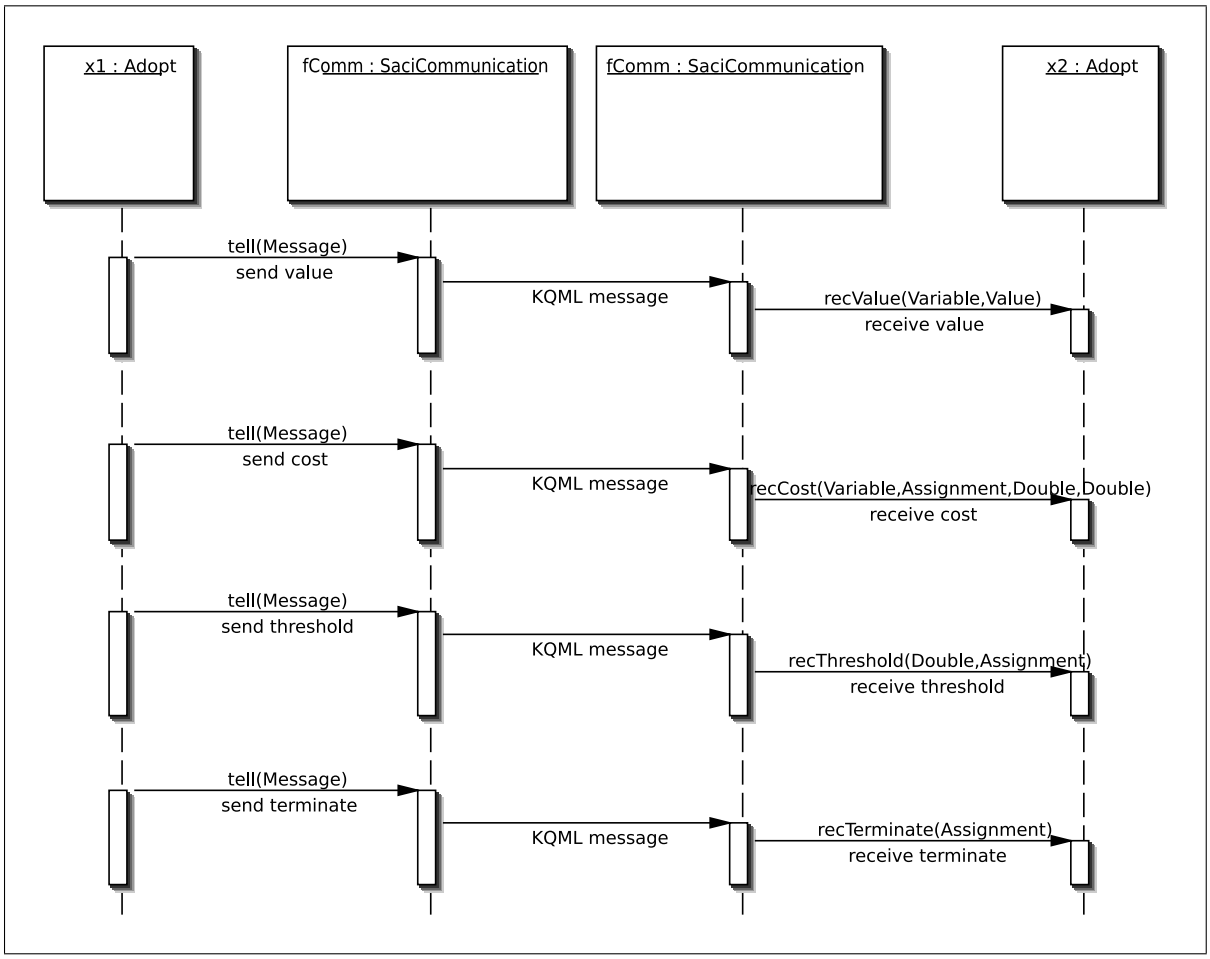



**Figura 4.5** – Mensagens na implementação do Adopt

Como o *framework* dynDCSP original continha apenas algoritmos para DCSP, a especificação do agente gerenciador de execução era voltada para estes tipos de algoritmo. Contudo, para adicionar um algoritmo de DCOP, foram necessárias alterações no agente gerenciador, contido no pacote `dcsp.main`. O diagrama da fig. 4.6 apresenta as alterações do pacote `dcsp.main`.

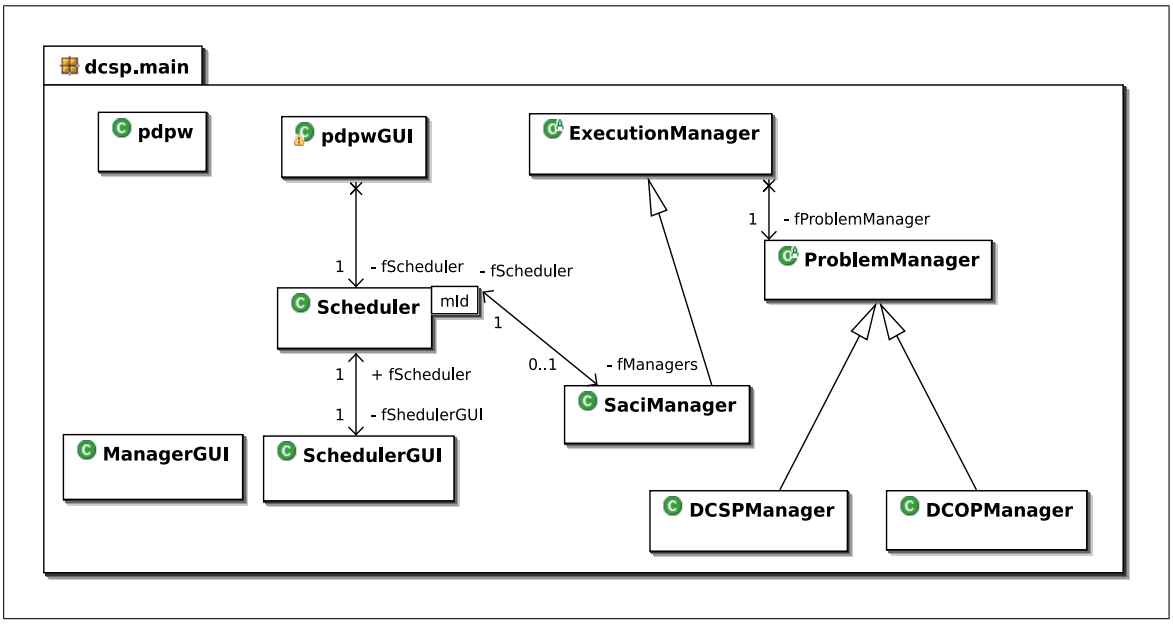



**Figura 4.6** – Alterações no pacote `dcsp.main`

Esta alteração foi motivada porque a inicialização de agentes DCOP é diferente da inicialização de agentes DCSP, pois agentes DCOP necessitam receber informações relativas a função de custo e a DFST. Para tanto, primeiramente a classe `BaseManager` foi dividida em duas: `ExecutionManager`, que contém operações unicamente relacionadas ao controle da execução (*start/stop*); e `ProblemManager`, com operações relacionadas a inicialização do problema que pretende-se resolver. Esta última foi estendida em duas outras classes: `DCSPManager`, para inicializar algoritmos que resolvem DCSP e `DCOPManager`, para inicializar algoritmos que resolvem DCOP. A seguir são apresentados detalhes destas classes.

A classe `ExecutionManager` (fig. 4.7) tem por objetivo controlar a execução do algoritmo, enviando mensagens *start* ou *stop* para os agentes, bem como contabilizar o tempo de execução. Para isto dispõe, por exemplo, dos atributos `fState`, que indica o estado da execução (*iniciando*, *executando*, *finalizado*, entre outros). Os atributos `fStartTime` e `fElapsedTime` controlam, respectivamente, a hora em que a execução foi iniciada e o tempo utilizado pela execução. Quanto a métodos, esta classe possui `addMsgHandlers()` que registra *callbacks* para indicar o término da execução de cada agente e do algoritmo. O método abstrato `createAgents()` deve ser implementado pela infraestrutura de co-

municação utilizada para criar efetivamente os agentes do algoritmo. Já `sendStart()` é responsável por enviar uma mensagem *start* para os agentes, indicando que devem iniciar a execução do algoritmo. Esta classe também dispõe do método abstrato `finished()` que pode ser implementado pela infraestrutura de comunicação utilizada para realizar algum procedimento quando a execução é finalizada (por exemplo, destruir os agentes). Por fim, esta classe apresenta o método `run()`, que além de registrar as *callbacks* e enviar *start* (métodos descritos anteriormente), executa o `start()` do `ProblemManager` específico do problema.

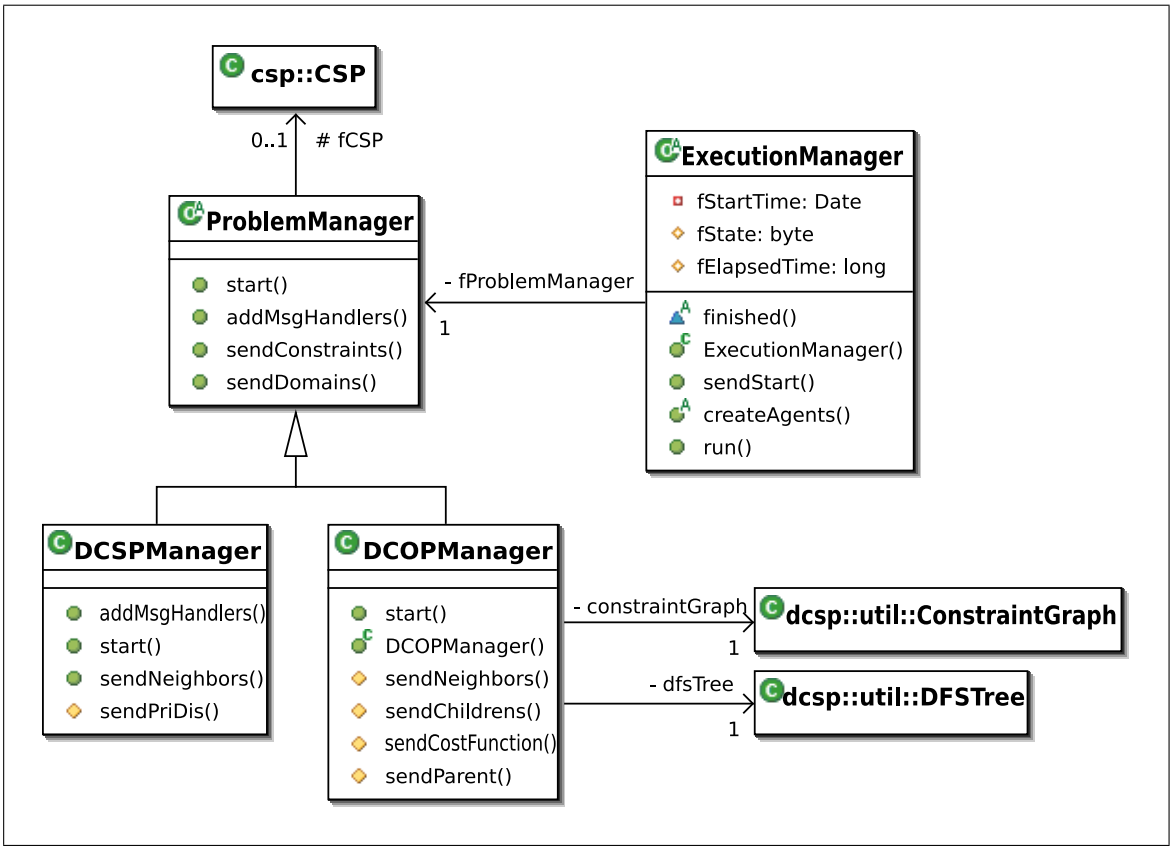



**Figura 4.7** – Detalhes das classes do pacote `dcsp.main`

A classe `ProblemManager` (fig. 4.7) reúne as rotinas de inicialização comuns aos dois tipos de algoritmo. À esta classe está associada uma instância de `CSP`, de onde retira informações para o algoritmo, como as variáveis, os domínios e as restrições. Esta classe também possui o método `addMsgHandlers()` que registra uma *callback* para processar mensagens que requisitam a solução final do problema. Há também o método `sendConstraints()` e `sendDomains()` que enviam, respectivamente, as restrições e os

domínios para os agentes, e por fim o método `start()`, que tem por finalidade executar os métodos descritos anteriormente.

A classe `DCSPManager` (fig. 4.7) tem por objetivo inicializar unicamente agentes de algoritmos de DCSP. O método `addMsgHandlers()` registra uma *callback* para receber mensagens indicando que não há solução do problema (caso algum agente descobre que o problema não tem solução, este informa os outros para que a execução seja parada). Já o método `sendNeighbors()` envia os agentes *neighbors* para um determinado agente e o método `sendPriDis()` envia a prioridade de cada variável e a distância máxima entre duas variáveis em relação ao grafo de restrições (estas informações só são processadas pelo agente se ele registrar uma *callback* para tratá-las, como ocorre no algoritmo DB). Há também o método `start()`, que executa os métodos descritos.

Por fim, a classe `DCOPManager` (fig. 4.7) é responsável por inicializar os agentes do algorimo Adopt. Para tanto, à esta classe estão associados um grafo de restrições e uma DFST. Os métodos `sendCostFunction()`, `sendChildrens()`, `sendNeighbors()` e `sendParent()` enviam para um agente, respectivamente, a função de custo, os *childrens*, os *neighbors* e o *parent*.

Conforme verificado na seção 2.4.1.1, para implementar o algoritmo Adopt é necessário definir uma DFST a partir de um grafo de restrições. Para tanto, foi criado o pacote `dcsp.util`, que contém classes que especificam estas duas estruturas. A fig. 4.8 apresenta o diagrama de classes deste pacote, sendo `ConstraintGraph` a classe que representa um grafo de restrições e `DFSTree` a que representa uma DFST. Para criação de uma DFST, é necessário primeiramente ter um grafo de restrições. Este grafo de restrições é criado a partir das restrições/funções de custo definidas na classe COP do problema em questão. Ao criar uma árvore pode-se optar por construir os ramos da árvore através do nome das variáveis, ou levando-se em consideração o grau de cada vértice do grafo, sendo esta última forma a recomendada por Modi (2003b, p. 27) pois possibilita um melhor fluxo de comunicação entre os agentes.

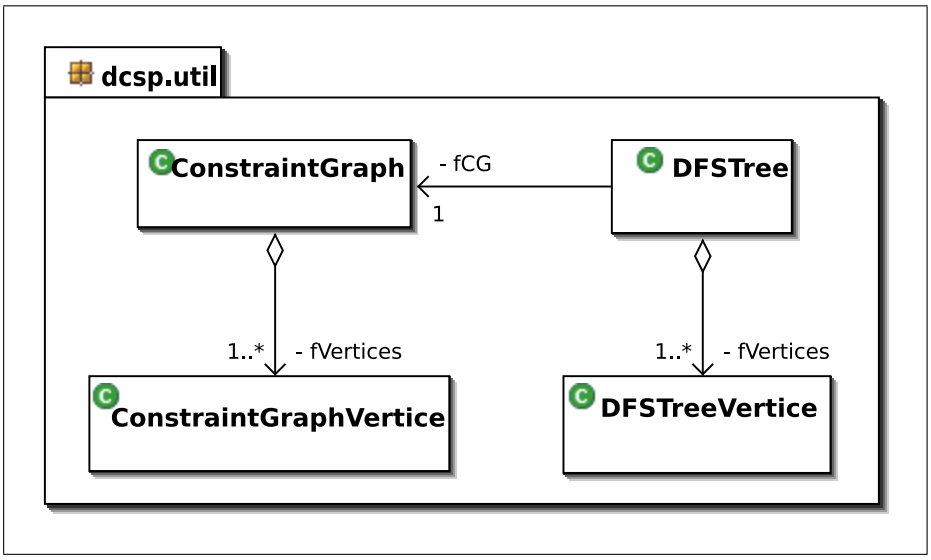



**Figura 4.8** – Classes do pacote `dcsp.util`

## 4.3 IMPLEMENTAÇÃO

Para implementação, foram utilizados a linguagem de programação Java versão 1.4, pois o *framework* é implementado nesta linguagem, e o ambiente Eclipse versão 3.0. A seguir são apresentados alguns trechos de código ilustrando a implementação nesta linguagem.

O registro de *callbacks* é apresentado no quadro 4.1. Para registrar uma *callback* é necessário instanciar um objeto de `MessageHandler` (cuja classe está no pacote SACI) e implementar o método `processMessage()`, que deve conter o tratamento da mensagem recebida. No quadro 4.1 é apresentado o registro de *callbacks* para tratar os tipos de mensagens existentes no algoritmo Adopt: THRESHOLD, TERMINATE, VALUE e COST.

```
1 private void addMsgHandlers() {
2   // add a message handler to receive THRESHOLD messages
3   fComm.addMessageHandler("threshold", new MessageHandler() {
4     public boolean processMessage(Message m) {
5       recThreshold((Double) m.get("t"),
6             (Assignment)m.get("context"));
7       return true;
8     }
9   });

11  // add a message handler to receive TERMINATE messages
12  fComm.addMessageHandler("terminate", new MessageHandler() {
13    public boolean processMessage(Message m) {
14      recTerminate((Assignment) m.get("context"));
15      return true;
16    }
17  });

19  // add a message handler to receive VALUE messages
20  fComm.addMessageHandler("value", new MessageHandler() {
21    public boolean processMessage(Message m) {
22      recValue((Variable) m.get("xj"), (Value) m.get("dj"));
23      return true;
24    }
25  });

27  // add a message handler to receive COST messages
28  fComm.addMessageHandler("cost", new MessageHandler() {
29    public boolean processMessage(Message m) {
30      recCost((Variable) m.get("xk"),
31          (Assignment) m.get("context"),
32          (Double) m.get("lb"),
33          (Double) m.get("ub"));
34      return true;
35    }
36  });
37 }
```

**Quadro 4.1** – Adopt: registro de *callbacks*

Os métodos que enviam mensagens VALUE, THRESHOLD, COST e TERMINATE são apresentados nos quadros 4.2, 4.3, 4.4 e 4.5. Nota-se que para enviar uma mensagem deve-se solicitar ao objeto `Communication` a criação da mesma, adicionar os "parâmetros" da mensagem para em seguida solicitar ao objeto `Communication` o envio da mesma, através do método `tell()` ou `ask()`, dependendo do caso.

```
1 protected void sendValue() {
2   // for all neighbors
3   Iterator itNeighbors = getNeighbors().iterator();
4   while (itNeighbors.hasNext()) {
5     Message mValue = fComm.createMessage("value");
6     // adding message parameters
7     mValue.put("xj", getVariable());
8     mValue.put("dj", getCurrentValue());
9     // adding message receiver
10    Variable neighbor  = (Variable)itNeighbors.next();
11    mValue.put("receiver", neighbor.getId());
12    // sending message
13    fComm.tell(mValue);
14  }
15 }
```

**Quadro 4.2** – Adopt: envio de mensagem VALUE

```
1 protected void sendThreshold() {
2   // for all childrens
3   Iterator itChildrens = getChildrens().iterator();
4   while (itChildrens.hasNext()) {
5     Variable child = (Variable)itChildrens.next();
6     // verifies if children threshod was changed
7     if ( isChildrenThresholdChanged(child) ){
8       Message mThreshold = fComm.createMessage("threshold");
9       // adding message parameters
10      mThreshold.put("context", getCurrentContext().clone());
11      Association ass = new Association(child, getCurrentValue());
12      mThreshold.put("t", fChildrensT.get(ass));
13      // adding message receiver
14      mThreshold.put("receiver", child.getId());
15      // sending message
16      fComm.tell(mThreshold);
17      // signal that this children threshold was send
18      setChildrenThresholdChanged(child, false);
19    }
20  }
21 }
```

**Quadro 4.3** – Adopt: envio de mensagem THRESHOLD

```
1 protected void sendCost() {
2   if ( getParent() != null ){
3     Message mCost = fComm.createMessage("cost");
4     // adding message parameters
5     mCost.put("xk", getVariable());
6     mCost.put("context", getCurrentContext().clone());
7     mCost.put("lb", new Double(getLB()));
8     mCost.put("ub", new Double(getUB()));
9     // adding message receiver
10    mCost.put("receiver", getParent().getId());
11    // sending message
12    fComm.tell(mCost);
13  }
14 }
```

**Quadro 4.4** – Adopt: envio de mensagem COST

```
1 protected void sendTerminate() {
2   Iterator itChildrens = getChildrens().iterator();
3   while (itChildrens.hasNext()) {
4     Message mTerminate = fComm.createMessage("terminate");
5     // adding message parameters
6     mTerminate.put("context", getCurrentContextPlusMe());
7     // adding message receiver
8     Variable child = (Variable)itChildrens.next();
9     mTerminate.put("receiver", child.getId());
10    // sending message
11    fComm.tell(mTerminate);
12  }
13 }
```

**Quadro 4.5** – Adopt: envio de mensagem TERMINATE

Cada uma das *procedures* existentes no algoritmo Adopt foi transformada em um método da classe Adopt. Os quadros 4.6, 4.7, 4.8 e 4.9 apresentam estes métodos.

```
1 // see backtrack procedure (Adopt pseudo-code)
2 protected void backTrack() {
3   if ( getThreshold() == getUB() )
4     setCurrentValue( getValueThatGiveUB() );
5   else{
6     if (lowerBoundFromValue(getCurrentValue()) > getThreshold())
7       setCurrentValue(getValueThatGiveLB());
8   }
9   // alg. optimization: send only when current value was changed
10  if ( isCurrentValueChanged() ){
11    sendValue();
12    setCurrentValueChanged(false);
13  }
14  maintainAllocationInvariant();
15  if ( getThreshold() == getUB() ){
16    if ( isParentTerminated() || getParent() == null){
17      setTerminated(true);
18      sendTerminate();
19    }
20  }
21  if ( ! isTerminated() )
22    // alg. optimization: send only when something was changed
23    if (isCurrentContextChanged()||isLBChanged()||isUBChanged()){
24      sendCost();
25      setCurrentContextChanged(false);
26      setLBChanged(false);
27      setUBChanged(false);
28    }
29 }
```

**Quadro 4.6** – Adopt: método `backtrack()`

```
1 // see maintainThresholdInvariant procedure (Adopt pseudo-code)
2 protected void maintainThresholdInvariant() {
3   if ( getThreshold() < getLB() ){
4     setThreshold( getLB() );
5   }
6   if ( getThreshold() > getUB() )
7     setThreshold( getUB() );
8   }
9 }
```

**Quadro 4.7** – Adopt: método `maintainThresholdInvariant()`

```
1 // see maintainAllocationInvariant procedure (Adopt pseudo-code)
2 protected void maintainAllocationInvariant(){
3   if ( getChildrens().size() > 0 ){
4     double avaibleThreshold = getThreshold()-
5       evalCostFunctionWithConstraints(getCurrentContextPlusMe());
6     double childrensThreshold = childrensThresholdSummation();
7     if ( avaibleThreshold > childrensThreshold){
8       double avaibleToIncreaseThreshold =
9             sub(avaibleThreshold, childrensThreshold);
10      Iterator itChilds = getChildrens().iterator();
11      while(itChilds.hasNext() && avaibleToIncreaseThreshold>0){
12        Variable child = (Variable)itChilds.next();
13        avaibleToIncreaseThreshold =
14          increaseThreshold(child,avaibleToIncreaseThreshold);
15      }
16    }else{
17      if ( avaibleThreshold < childrensThreshold){
18        double avaibleToDecreaseThreshold =
19            sub(childrensThreshold, avaibleThreshold);
20        Iterator itChilds = getChildrens().iterator();
21        while (itChilds.hasNext() &&
22            avaibleToDecreaseThreshold>0){
23          Variable child = (Variable)itChilds.next();
24          avaibleToDecreaseThreshold =
25            decreaseThreshold(child,
26                      avaibleToDecreaseThreshold);
27        }
28      }
29    }
30  }
31  sendThreshold();
32 }
```

**Quadro 4.8** – Adopt: método `maintainAllocationInvariant()`

```
1 // see maintainChildThresholdInvariant procedure (Adopt pseudo-code)
2 protected void maintainChildThresholdInvariant() {
3   Iterator itDomain = getVariable().getDomain().iterator();
4   while(itDomain.hasNext()){
5     Value vl = (Value)itDomain.next();
6     List childs = getChildrensWhereLbGreatherT(vl);
7     Iterator itThresh = childs.iterator();
8     while (itThresh.hasNext()){
9       Variable var = (Variable)itThresh.next();
10      Association ass = new Association(var, vl);
11      Double lbValue = (Double)fChildrensLb.get(ass);
12      fChildrensT.put(ass, new Double(lbValue.doubleValue()));
13      setChildrenThresholdChanged(var, true);
14    }
15    childs = getChildrensWhereTGreaterUb(vl);
16    itThresh = childs.iterator();
17    while (itThresh.hasNext()){
18      Variable var = (Variable)itThresh.next();
19      Association ass = new Association(var, vl);
20      Double ubValue = (Double)fChildrensUb.get(ass);
21      fChildrensT.put(ass, new Double(ubValue.doubleValue()));
22      setChildrenThresholdChanged(var, true);
23    }
24  }
25 }
```

**Quadro 4.9** – Adopt: método `maintainChildThresholdInvariant()`

Por fim, os quadros 4.10 e 4.11 apresentam a implementação dos métodos `computeLB()` e `computeUB()` respectivamente, que calculam o *lower bound* e o *upper bound* da variável (agente).

```
1 protected void computeLB(){
2   double minLbD = -1;
3   Value valueThatGiveMinLb = null;
4   // for all domain values
5   Iterator itDomain = getVariable().getDomain().iterator();
6   while (itDomain.hasNext()){
7     Value vl = (Value)itDomain.next();
8     // computes lower bound from this domain value
9     double lb_d = lowerBoundFromValue(vl);
10    if ( valueThatGiveMinLb == null){
11      valueThatGiveMinLb = vl;
12      minLbD = lb_d;
13    }else{
14      if ( lb_d < minLbD ){
15        minLbD = lb_d;
16        valueThatGiveMinLb = vl;
17      }
18    }
19  }
20  // set current LB and the domain value that give this LB
21  setLB(minLbD);
22  setValueThatGiveLB(valueThatGiveMinLb);
23 }
```

**Quadro 4.10** – Adopt: método `computeLB()`

```
1 protected void computeUB(){
2   double minUbD = -1;
3   Value valueThatGiveMinUb = null;
4   // for all domain values
5   Iterator itDomain = getVariable().getDomain().iterator();
6   while (itDomain.hasNext()){
7     Value vl = (Value)itDomain.next();
8     // compute upper bound from this domain value
9     double ub_d = upperBoundFromValue(vl);
10    if ( valueThatGiveMinUb == null){
11      valueThatGiveMinUb = vl;
12      minUbD = ub_d;
13    }else{
14      if ( ub_d < minUbD ){
15        minUbD = ub_d;
16        valueThatGiveMinUb = vl;
17      }
18    }
19  }
20  // set current UB and the domain value that give this UB
21  setUB(minUbD);
22  setValueThatGiveUB(valueThatGiveMinUb);
23 }
```

**Quadro 4.11** – Adopt: método `computeUB()`

Em função das alterações que adicionaram ao *framework* a capacidade de resolver DCOPs, houve a necessidade de modificar o arquivo de configuração do *framework* para que o mesmo especifique os dois tipos de algoritmos implementados: que solucionam DCOP e que solucionam DCSP. O novo formato do arquivo de configuração é apresentado no quadro 4.12. Agora o arquivo de configuração apresenta duas seções distintas. A primeira, denominada *algorithm-types* especifica os tipos de problemas que o *framework* resolve, os algoritmos disponíveis para o tipo de problema e qual *Manager* deve ser utilizado para o tipo de problema. A segunda seção não sofreu alterações, e especifica os problemas "cadastrados" no *framework*.

```
1 # This file contains the problems and
2 # algoritms available for DCSP.
3 # Each value is the java class name of either
4 # the algorithm or the problem factory.

6 +algorithm-types
7 +type
8     DCSP
9 +algorithms
10     dcsp.alg.AB
11     dcsp.alg.AWC
12     dcsp.alg.DB
13 +manager
14     dcsp.main.DCSPManager

16 +type
17     DCOP
18 +algorithms
19     dcsp.alg.Adopt
20 +manager
21     dcsp.main.DCOPManager

23 +problems
24     helloCSP.HelloCSPFactory
25     mSquare.MagicSquareCSPFactory
26     nqueens.NQueensCSPFactory
27     zebra.ZebraCSPFactory
28     helloAdopt.HelloAdoptCSOPFactory
29     fepk.FePKCSOPFactory
30     molecules.WaterCSOPFactory
31     molecules.EthaneCOPFactory
32     graphColoring.GraphColoring8VerticesCOPFactory
```

**Quadro 4.12** – Novo formato do arquivo de configuração do *framework*

# 5 ESTUDO DE CASO: COLORAÇÃO DE GRAFO

Para validar a implementação do algoritmo Adopt em ambiente distribuído e utilizando SACI, optou-se por utilizar o problema de coloração de grafos. Isto porque Modi (2003b) utiliza este mesmo problema para validar seu algoritmo, e os grafos utilizados pelo autor em sua validação estão disponíveis, juntamente com uma implementação não distribuída em linguagem Java (desenvolvida pelo autor) em Modi (2003a), o que torna possível, além de validar a implementação realizada neste trabalho, comparar com a existente.

Para exemplificação de como é especificado um COP para coloração de grafos, será utilizado um grafo de oito vértices e onze arestas. Este grafo está representado graficamente na fig. 5.1. Cada vértice do grafo representa uma variável que possui um domínio de cores, sendo estas cores representadas por números inteiros. As arestas representam relações entre estas variáveis. Para cada uma destas relações existe uma função de custo, sendo o objetivo encontrar um valor para cada variável de forma a minimizar estas funções de custo.

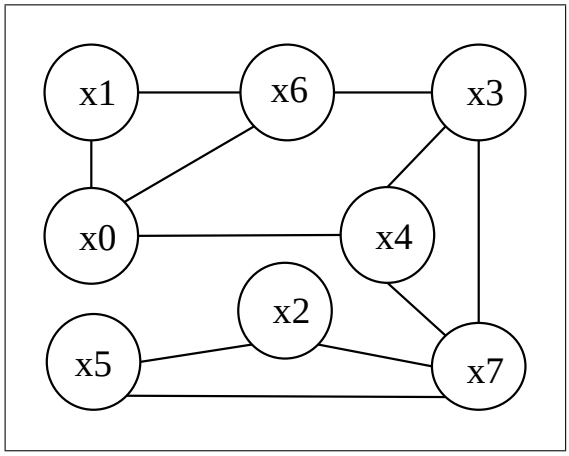



**Figura 5.1** – Problema de coloração de grafo

O grafo da fig. 5.1 foi retirado de Modi (2003a), onde está descrito de forma textual através de termos que denotam um COP. Esta descrição é utilizada pela implementação do

algoritmo Adopt disponível também em Modi (2003a) e especifica as variáveis, as funções de custo e a quantidade de agentes que devem ser criados para execução do algoritmo na implementação disponibilizada pelo autor (onde cada agente pode possuir mais de uma variável). A descrição textual apresentada no quadro 5.1 define um COP para o grafo da fig. 5.1. A linhas 1-8 definem os agentes que devem ser criados na implementação de Modi (2003a). As linhas 9-16 definem oito variáveis, sendo esta definição apresentada pela quadrupla $(VARIABLE, id, agent, domain)$, onde a palavra VARIABLE define uma variável, *id* coresponde ao identificador da variável (0-7), *agent* especifica qual agente será responsável por esta variável e *domain* define a quantidade de elementos contidos no domínio desta variável, sendo este domínio composto por números naturais na forma: $(0, 1, ..., domain-1)$. As próximas linhas descrevem as restrições binárias. Uma restrição é descrita pela palavra CONSTRAINT seguida de dois números $x_1$ e $x_2$, que correspondem aos identificadores do par de variáveis a qual esta restrição se aplica. Na sequência, é estabelecida uma lista de valores que este par de variáveis não pode assumir através da palavra NOGOOD seguida de dois números $v_1$ e $v_2$, que especificam os valores que as variáveis $x_1$ e $x_2$ não podem assumir, respectivamente.

```
1  AGENT 1
2  AGENT 2
3  AGENT 3
4  AGENT 4
5  AGENT 5
6  AGENT 6
7  AGENT 7
8  AGENT 8
9  VARIABLE 0 1 3
10 VARIABLE 1 2 3
11 VARIABLE 2 3 3
12 VARIABLE 3 4 3
13 VARIABLE 4 5 3
14 VARIABLE 5 6 3
15 VARIABLE 6 7 3
16 VARIABLE 7 8 3
17 CONSTRAINT 0 4
18 NOGOOD 2 1
19 NOGOOD 0 1
20 NOGOOD 1 0
21 CONSTRAINT 3 6
22 NOGOOD 2 1
23 NOGOOD 0 2
24 NOGOOD 0 0
25 CONSTRAINT 2 5
26 NOGOOD 0 1
27 NOGOOD 0 2
28 NOGOOD 2 1
29 CONSTRAINT 1 6
30 NOGOOD 0 0
31 NOGOOD 1 2
32 NOGOOD 0 1
33 CONSTRAINT 0 1
34 NOGOOD 0 2
35 NOGOOD 2 2
36 NOGOOD 1 2
37 CONSTRAINT 3 7
38 NOGOOD 2 1
39 NOGOOD 1 0
40 NOGOOD 0 1
41 CONSTRAINT 0 6
42 NOGOOD 0 0
43 NOGOOD 2 2
44 NOGOOD 1 2
45 CONSTRAINT 3 4
46 NOGOOD 2 0
47 NOGOOD 0 2
48 NOGOOD 0 0
49 CONSTRAINT 4 7
50 NOGOOD 2 2
51 NOGOOD 2 0
52 NOGOOD 0 1
53 CONSTRAINT 2 7
54 NOGOOD 1 2
55 NOGOOD 2 2
56 NOGOOD 0 2
57 CONSTRAINT 5 7
58 NOGOOD 1 0
59 NOGOOD 2 2
60 NOGOOD 1 2
```

Fonte: adaptado de Modi (2003a)

**Quadro 5.1** – Descrição textual de grafo para coloração

O cálculo das funções de custo é realizado a partir da avaliação de cada NOGOOD (verificando se $x_1 = v_1$ e $x_2 = v_2$ é verdadeiro ou falso). Se a avaliação for falsa, a função de custo retorna o valor zero, se verdadeira, retorna um valor maior que zero. Esta função de custo pode ser representada por uma expressão, conforme apresenta a equação (5.1).

$$f(x_1, x_2) = \begin{cases} 1 & \text{if } (x_1 = v_1, x_2 = v_2) \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

Compreendida a maneira de apresentação de um COP de coloração de grafo na implementação de Modi (2003a), elaborou-se uma descrição textual do COP de coloração de grafo mais compreensível ao leitor. Esta descrição, apresentada no quadro 5.2 (página 74), serve de ponto de partida para especificação da classe `COP`. A partir desta descrição, pode-se observar que este problema possui um domínio, que foi denominado *colors*, composto pelos valores inteiros 0, 1 e 2 (seção *Domains*). Há também oito variáveis (seção *Variables*), $x_0$, $x_1$, $x_2$, $x_3$, $x_4$, $x_5$, $x_6$, $x_7$, sendo *colors* o domínio de cada uma delas. Por fim, existem onze funções de custo (seção *Cost Functions*).

A classe criada para este problema de coloração chama-se `GraphColoring8VerticesCOP`, estende de `COP` e destina-se a definir um COP para o problema de coloração apresentados na fig. 5.1 e no quadro 5.2.

O quadro 5.3 (página 75), contém a declaração da classe `GraphColoring8Vertices`. Inicialmente esta classe define duas `Expression` constantes (provenientes do pacote `csp.exp`, linhas 4 e 8) que correspondem ao valor de retorno da função de custo. Na sequência é definido o domínio `colors` (linha 16) para todas as variáveis. A definição da variável $x_0$ é vista na linha 19 sendo na sequência atribuído o domínio `colors` para esta variável e adicionada no problema COP (a definição das sete outras variáveis seguem o mesmo princípio, e por isto são omitidas). Por fim, são definidas as funções de custo entre as variáveis. Para definir estas funções de custo, optou-se por criar um método chamado `createCostExpression()` que retorna uma função de custo a partir das expressões do quadro 5.2 (este método é detalhado a seguir). Portanto, para adicionar as funções de custo ao COP basta, para cada expressão do quadro 5.2, chamar `addCostFunction()` com a função de custo criada por `createCostExpression()` (linha 26).

**Graph Coloring COP**

**Domains**

colors: $\{1, 2, 3\}$

**Variables**

$x_0$, $x_1$, $x_2$, $x_3$, $x_4$, $x_5$, $x_6$, $x_7$ : colors;

**Cost Functions**

$$f(x_0, x_4) = \begin{cases} 1 & \text{if } (x_0 = 2, x_4 = 1) \text{ or } (x_0 = 0, x_4 = 1) \text{ or} (x_0 = 1, x_4 = 0) \\ 0 & \text{otherwise} \end{cases}$$

$$f(x_3, x_6) = \begin{cases} 1 & \text{if } (x_3 = 2, x_6 = 1) \text{ or } (x_3 = 0, x_6 = 2) \text{ or} (x_3 = 0, x_6 = 0) \\ 0 & \text{otherwise} \end{cases}$$

$$f(x_2, x_5) = \begin{cases} 1 & \text{if } (x_2 = 0, x_5 = 1) \text{ or } (x_2 = 0, x_5 = 2) \text{ or} (x_2 = 2, x_5 = 1) \\ 0 & \text{otherwise} \end{cases}$$

$$f(x_1, x_6) = \begin{cases} 1 & \text{if } (x_1 = 0, x_6 = 0) \text{ or } (x_1 = 1, x_6 = 2) \text{ or} (x_1 = 0, x_6 = 1) \\ 0 & \text{otherwise} \end{cases}$$

$$f(x_0, x_1) = \begin{cases} 1 & \text{if } (x_0 = 0, x_1 = 2) \text{ or } (x_0 = 2, x_1 = 2) \text{ or} (x_0 = 1, x_1 = 2) \\ 0 & \text{otherwise} \end{cases}$$

$$f(x_3, x_7) = \begin{cases} 1 & \text{if } (x_3 = 2, x_7 = 1) \text{ or } (x_3 = 1, x_7 = 0) \text{ or} (x_3 = 0, x_7 = 1) \\ 0 & \text{otherwise} \end{cases}$$

$$f(x_0, x_6) = \begin{cases} 1 & \text{if } (x_0 = 0, x_6 = 0) \text{ or } (x_0 = 2, x_6 = 2) \text{ or} (x_0 = 1, x_6 = 2) \\ 0 & \text{otherwise} \end{cases}$$

$$f(x_3, x_4) = \begin{cases} 1 & \text{if } (x_3 = 2, x_4 = 0) \text{ or } (x_3 = 0, x_4 = 2) \text{ or} (x_3 = 0, x_4 = 0) \\ 0 & \text{otherwise} \end{cases}$$

$$f(x_4, x_7) = \begin{cases} 1 & \text{if } (x_4 = 2, x_7 = 2) \text{ or } (x_4 = 2, x_7 = 0) \text{ or} (x_4 = 0, x_7 = 1) \\ 0 & \text{otherwise} \end{cases}$$

$$f(x_2, x_7) = \begin{cases} 1 & \text{if } (x_2 = 1, x_7 = 2) \text{ or } (x_2 = 2, x_7 = 2) \text{ or} (x_2 = 0, x_7 = 2) \\ 0 & \text{otherwise} \end{cases}$$

$$f(x_5, x_7) = \begin{cases} 1 & \text{if } (x_5 = 1, x_7 = 0) \text{ or } (x_5 = 2, x_7 = 2) \text{ or} (x_5 = 1, x_7 = 2) \\ 0 & \text{otherwise} \end{cases}$$

**Quadro 5.2** – Descrição compreensível de grafo para coloração

```
1 public class GraphColoring8VerticesCOP extends COP {

3   // cost value to violated NOGOOD
4   private Expression violatedCost =
5             new ConstantExpression(new Value(new Integer(1)));

7   // cost value to non violated NOGOOD
8   private Expression nonViolatedCost =
9             new ConstantExpression(new Value(new Integer(0)));

11  public GraphColoring8VerticesCOP (){

13    super("Graph Coloring with 8 vertices");

15    // creating the domain
16    IntegerDomain colors = new IntegerDomain("Colors", 0, 2);

18    // creating variable x0 and adding it to COP
19    Variable x0 = new Variable("x0");
20    x0.setDomain(colors);
21    addVariable(x0);

23    // definition of another 7 variables...

25    // creating the cost function between x0 and x4
26    addCostFunction(
27        createCostExpression(x0, x4, 2, 1, 0, 1, 1, 0));

29    // definition of another cost functions
30  }
31 }
```

**Quadro 5.3** – `GraphColoring8VerticesCOP`: definição da classe e domínio

Para criação das funções de custo, foram definidas expressões condicionais para cada uma das expressões do quadro 5.2. Desta maneira, a expressão condicional que define, por exemplo, a função de custo entre as variáveis $x_0$ e $x_4$, pode ser vista no quadro 5.4. A partir destas expressões condicionais e utilizando as `Expression` contidas no pacote `csp.exp`, são definidas as funções de custo para o `GraphColoring8VerticesCOP`. Os quadros 5.5 e 5.6 apresentam o método `createCostExpression()`, responsável por criar a expressão condicional que representa a função de custo, dado um par de variáveis e os valores que este par de variáveis não pode assumir.

```
if ($x_0$==2 && $x_4$==1)
  return 1
else
  if ($x_0$==0 && $x_4$==1)
    return 1
  else
    if ($x_0$==1 && $x_4$==0)
      return 1
    else
      return 0
```

**Quadro 5.4** – Expressão condicional como função de custo

```
32 private Expression createCostExpression(Variable xi, Variable xj,
33                         int xiv1, int xjv1,
34                         int xiv2, int xjv2,
35                         int xiv3, int xjv3){

37   EqualsExpression xieqxiv1 =
38         new EqualsExpression(
39           new VariableExpression(xi),
40           new ConstantExpression(new Value(new Integer(xiv1))));

42   EqualsExpression xieqxiv2 =
43         new EqualsExpression(
44           new VariableExpression(xi),
45           new ConstantExpression(new Value(new Integer(xiv2))));

47   EqualsExpression xieqxiv3 =
48         new EqualsExpression(
49           new VariableExpression(xi),
50           new ConstantExpression(new Value(new Integer(xiv3))));

52   EqualsExpression xjeqxjv1 =
53         new EqualsExpression(
54           new VariableExpression(xj),
55           new ConstantExpression(new Value(new Integer(xjv1))));

57   EqualsExpression xjeqxjv2 =
58         new EqualsExpression(
59           new VariableExpression(xj),
60           new ConstantExpression(new Value(new Integer(xjv2))));

62   EqualsExpression xjeqxjv3 =
63         new EqualsExpression(
64           new VariableExpression(xj),
65           new ConstantExpression(new Value(new Integer(xjv3))));
```

**Quadro 5.5** – Método `createCostExpression()`, parte 1

```
66   AndExpression xieqxiv1andxjeqxjv1 =
67         new AndExpression(xieqxiv1, xjeqxjv1);

69   AndExpression xieqxiv2andxjeqxjv2 =
70         new AndExpression(xieqxiv2, xjeqxjv2);

72   AndExpression xieqxiv3andxjeqxjv3 =
73         new AndExpression(xieqxiv3, xjeqxjv3);

75   IfExpression ifxieqxiv3andxjeqxjv3 =
76         new IfExpression(
77           xieqxiv3andxjeqxjv3,
78           violatedCost,
79           nonViolatedCost);

81   IfExpression ifxieqxiv2andxjeqxjv2 =
82         new IfExpression(
83           xieqxiv2andxjeqxjv2,
84           violatedCost,
85           ifxieqxiv3andxjeqxjv3);

87   IfExpression ifxieqxiv1andxjeqxjv1 =
88         new IfExpression(
89           xieqxiv1andxjeqxjv1,
90           violatedCost,
91           ifxieqxiv2andxjeqxjv2);

93   return ifxieqxiv1andxjeqxjv1 ;
94 }
```

**Quadro 5.6** – Método `createCostExpression()`, parte 2

Após a definição de uma classe *factory* para este problema de coloração, denominada `GraphColoring8VerticesCOPFactory`, e efetuadas as devidas alterações no arquivo de configuração do *framework* (estas etapas não são descritas em virtude da simplicidade e semelhança com o exemplo descrito no capítulo 3), este problema de coloração está apto para ser solucionado pela implementação do algoritmo Adopt realizada neste trabalho.

O fator utilizado para geração da DFST foi o grau de cada vértice. Isto garantiu uma árvore não linear, o que otimiza a comunicação entre os agentes. A DFST gerada para o COP de coloração com oito vértices é vista na fig. 5.2.

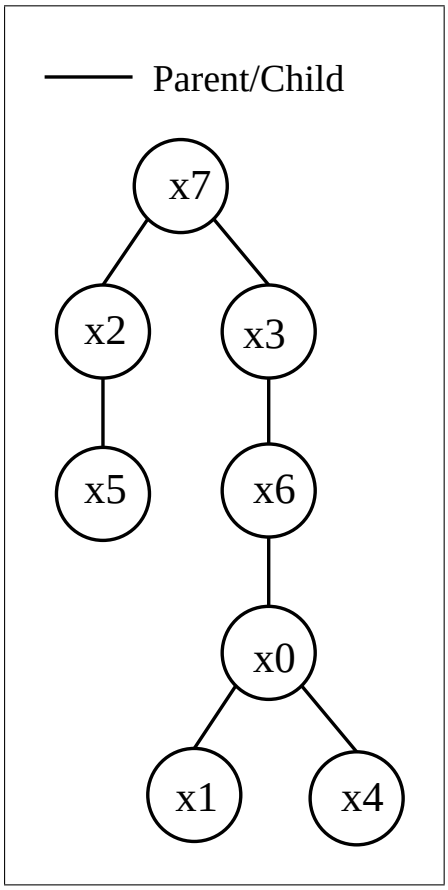



**Figura 5.2** – DFST para o grafo de oito vértices

A resolução do COP de coloração através do algoritmo Adopt com um *host* foi realizada com sucesso, sendo que a solução obtida pela implementação do algoritmo Adopt deste trabalho é a mesma que a solução obtida pela implementação disponível em Modi (2003a), o que valida a implementação realizada para este estudo caso. A fig. 5.3 apresenta o `ManagerGUI` com os resultados obtidos para este problema em uma execução do algoritmo Adopt. Estatísticas sobre a execução através da implementação do algoritmo Adopt deste trabalho são apresentadas na tab. 5.1 (página 80). Para validação foram realizadas dez execuções (coluna *Execução*), e para cada execução foram computados o tempo gasto (coluna *Tempo*) e a quantidade de mensagens trocadas (coluna *Mensagens trocadas*). A quantidade de mensagens trocadas corresponde ao somatório da quantidade de mensagens VALUE, COST, THRESHOLD E TERMINATE enviadas pelos agentes do algoritmo Adopt durante a execução.

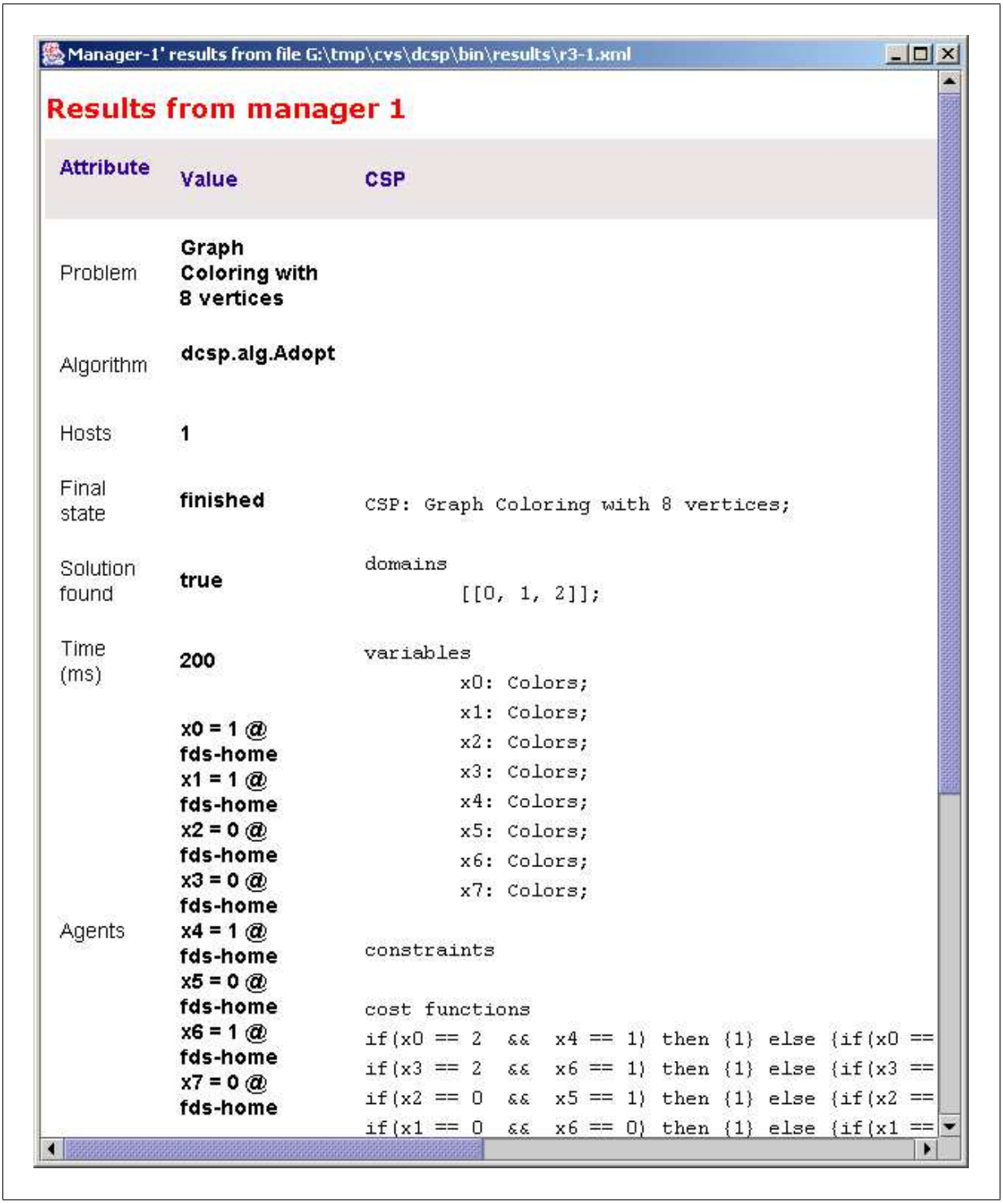



**Figura 5.3** – `ManagerGUI` com resultados do problema de coloração de grafo

**Tabela 5.1** – Adopt para coloração de grafo: *framework* dynDCSP

| *Execução* | *Mensagens Trocadas* | *Tempo (milisegundos)* |
|---|---|---|
| 1 | 47 | 239 |
| 2 | 51 | 202 |
| 3 | 42 | 167 |
| 4 | 46 | 102 |
| 5 | 48 | 112 |
| 6 | 52 | 92 |
| 7 | 51 | 117 |
| 8 | 53 | 109 |
| 9 | 51 | 96 |
| 10 | 47 | 90 |
| *Média* | 48,2 | 1574,2 |

O leitor atento irá verificar que a quantidade de mensagens trocadas entre os agentes do algoritmo Adopt não é a mesma nas dez execuções. Isto ocorre porque o algoritmo Adopt é assíncrono, o que o torna não-determinístico, pois seu funcionamento depende da ordem em que as mensagens são recebidas (MODI, 2003a).

Os tempos de execução obtidos utilizando a implementação de Modi (2003a) para o mesmo COP de coloração (descritos na tab. 5.2), são superiores aos obtidos com a implementação deste trabalho, o que comprova a eficiência da implementação e da infra-estrutura de comunicação. Já a quantidade de mensagens apresentou uma significativa diferença. Isto se deve ao fato de que a implementação do algoritmo Adopt deste trabalho já faz uso das otimizações propostas por Modi e Ali (2003c).

**Tabela 5.2** – Adopt para coloração de grafo: Modi (2003a)

| *Execução* | *Mensagens Trocadas* | *Tempo (milisegundos)* |
|---|---|---|
| 1 | 191 | 1886 |
| 2 | 181 | 1274 |
| 3 | 209 | 1548 |
| 4 | 160 | 1419 |
| 5 | 203 | 1318 |
| 6 | 156 | 1368 |
| 7 | 177 | 2102 |
| 8 | 182 | 1468 |
| 9 | 141 | 935 |
| 10 | 134 | 1071 |
| *Média* | 173,4 | 1438,9 |

Para verificar o desempenho do algoritmo Adopt em um ambiente efetivamente distribuído, foram realizadas execuções com diferentes números de *hosts*. O número máximo de *hosts* utilizado foi oito, que corresponde ao número de variáveis do problema, o que significa que havia um agente em cada *host*. Para esta situação, também foram realizadas dez execuções e tomados os tempos e as quantidades de mensagens. Na sequência o mesmo processo foi executado com sete, seis, cinco, quatro, três e dois *hosts*, sendo que conforme o número de *hosts* diminui, a quantidade de agentes por *hosts* aumenta. Ao final do processo, foram confeccionados dois gráficos.

O gráfico da fig. 5.4 apresenta o tempo gasto pela execução em função do número de *hosts* utilizados. Como se pode verificar, houve um pequeno aumento do tempo quando utilizou-se três e sete *hosts*. Porém, como esta diferença é pequena (menos de vinte milisegundos), pode-se dizer que o tempo de execução foi constante e independente da quantidade de *hosts*.

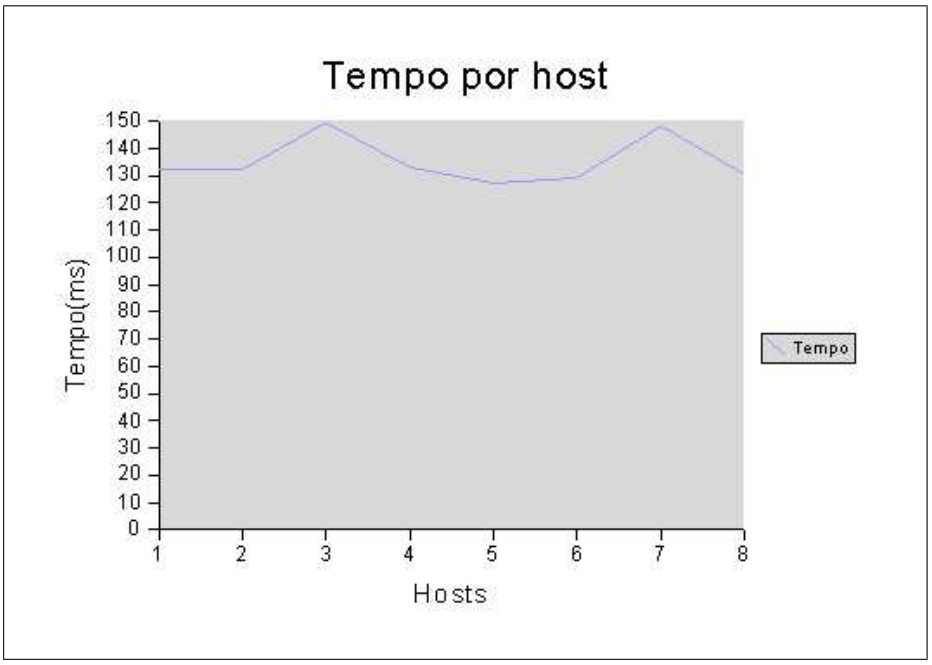



**Figura 5.4** – Adopt para coloração de grafo: Tempo por *hosts*

Já o gráfico da fig. 5.5 apresenta o número de mensagens trocadas em função do número de hosts. Assim como ocorreu com o tempo de execução, o número médio de mensagens trocadas se manteve constante.

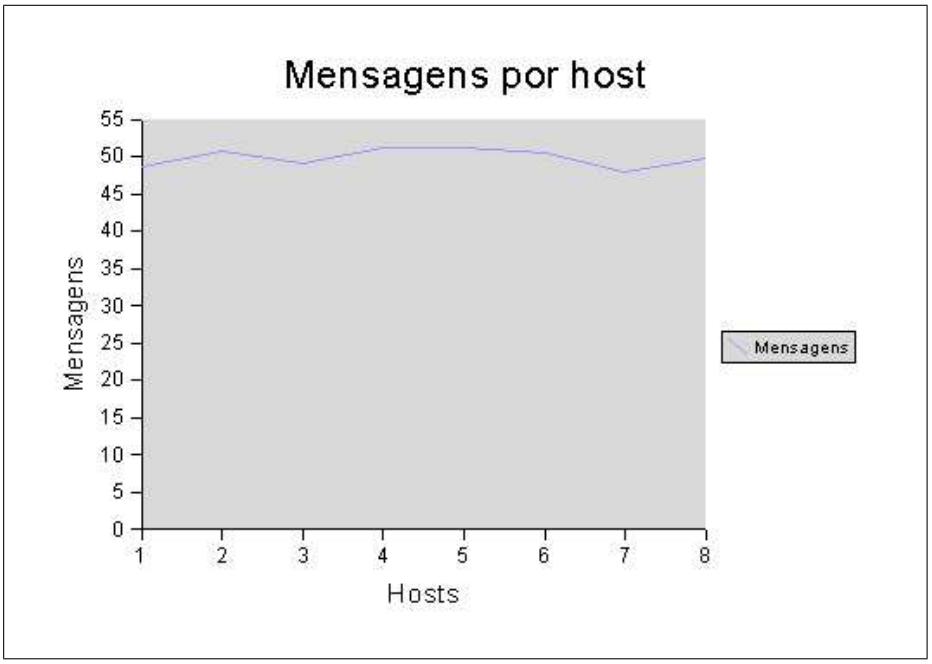



**Figura 5.5** – Adopt para coloração de grafo: Mensagens por *hosts*

O fato de que o tempo de execução se manteve constante neste estudo de caso,

independente da quantidade de *hosts* utilizados, não é suficiente para afirmar que a execução do algoritmo Adopt em um ambiente distribuído não apresenta ganho em relação a execução centralizada, pois talvez este ganho só seja percebido em problemas com um número maior de variáveis.

Conforme já mencionado, em Modi (2003a) encontra-se uma implementação do algoritmo Adopt, realizada pelo autor. Trata-se de uma implementação não distribuída, onde cada agente é constituído por um processo (*thread*) local. Nesta implementação, não é utilizada uma infraestrutura de comunicação entre agentes de terceiros: o próprio autor implementou a sua infraestrutura de comunicação. Agora, com a implementação realizada do algoritmo Adopt utilizando a infraestrutura de comunicação entre agentes SACI, é possível executar o algoritmo em ambientes efetivamente distribuídos.

# 6 ESTUDO DE CASO: MODELAGEM MOLECULAR

Para realização da modelagem molecular é fundamental obter conhecimentos da área de química, definindo átomos e moléculas, bem como a maneira com que as ligações entre os átomos de uma molécula são especificadas no processo de modelagem molecular. As seções a seguir pretendem apresentar estes conceitos.

## 6.1 ÁTOMOS E MOLÉCULAS

Toda matéria é formada por partículas extremamente pequenas denominadas átomos. Todo átomo é constituído de três partes: prótons, nêutrons e elétrons. Os prótons e nêutrons compõem conjuntamente o núcleo do átomo. Os elétrons estão dispostos (e em movimento) ao redor do núcleo em uma região denominada eletrosfera. A eletrosfera, por sua vez, é dividida em níveis de energia. Segundo Fonseca (1993, p. 32), um nível de energia é uma região do átomo onde o elétron pode movimentar-se sem perder nem ganhar energia. A cada nível corresponde uma quantidade de energia, que varia de acordo com o afastamento do nível energético em relação ao núcleo, assumindo valores menores quando o nível está próximo do núcleo, e maiores quando está afastado.

Para cada nível de energia $n$ está associada uma determinada quantidade de elétrons. Cada um destes níveis é designado por um número inteiro iniciando de 1 e crescente a medida que a quantidade de energia do nível aumenta. Cada um destes níveis pode ser dividido em subníveis de energia, designados pelas letras $s, p, d, f$. Da mesma maneira que ocorre com os níveis de energia, cada subnível é ocupado por uma quantidade máxima de elétrons. A tab. 6.1 apresenta alguns níveis, subníveis e o número máximo de elétrons em cada um dos níveis/sub-níveis.

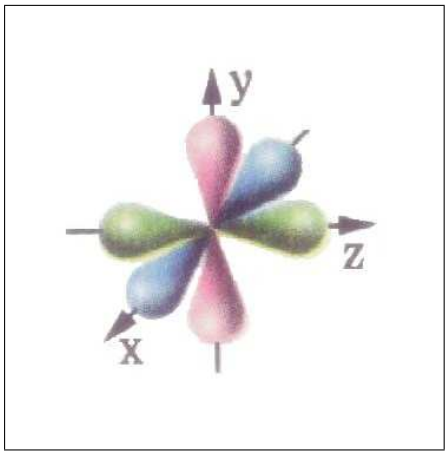
Os elétrons movimentam-se em torno do núcleo em regiões específicas, determinadas orbitais. Segundo Fonseca (1993), um orbital é a região do átomo onde a probabilidade de encontrar um elétron é máxima, visto que não se sabe exatamente o trajeto descrito pelo elétron ao movimentar-se, mas sabe-se apenas as regiões que a trajetória

**Tabela 6.1** – Disposição dos elétrons nos níveis de energia

| **Nível** | **Máx. de elétrons** | **Subníveis** | **Máx. de elétrons** |
|---|---|---|---|
| 1 | 2 | $s$ | 2 |
| 2 | 8 | $s, p$ | 2, 6 |
| 3 | 18 | $s, p, d$ | 2, 6, 10 |
| 4 | 32 | $s, p, d, f$ | 2, 6, 10, 14 |

Fonte: adaptado de Fonseca (1993, p. 34)

pode apresentar. Um orbital pode ser compartilhado por no máximo dois elétrons. A representação geométrica tri-dimensional de um orbital varia de acordo com o subnível de energia. Para o subnível de energia $p$, que comporta até no máximo seis elétrons, a representação geométrica é apresentada na fig. 6.1. Como cada orbital pode comportar até dois elétrons, tem-se três orbitais diferentes para o subnível $p$.




Fonte: Fonseca (1993, p. 36)

**Figura 6.1** – Orbitais do subnível $p$

A estabilidade de um átomo é obtida quando este atinge o número máximo de elétrons no último subnível de energia que o compõe. Este subnível é denominado de camada de valência, e é a partir dos elétrons da camada de valência que os átomos estabelecem ligações entre si. As ligações estabelecidas entre os átomos tem por objetivo buscar a estabilidade. Neste processo, os elétrons da camada de valência são compartilhados com os de outro átomo, dando origem a moléculas (FONSECA, 1993, p. 81).

É importante observar que existem determinados tipos de átomos em que o número de ligações estabelecidas não corresponde ao número de ligações previstas na configuração

eletrônica (disposição dos elétrons nos níveis energéticos) destes átomos. Nestes casos, diz-se que houve a hibridação do átomo em questão. (FONSECA, 1993, p. 88).

A hibridação ocorre quando elétrons de um determinado orbital são ativados, misturando-se, e passando a fazer parte de um novo orbital. A hibridação explica, por exemplo, o fato de um átomo de carbono (que em seu estado fundamental, sua configuração eletrônica apresenta dois elétrons no subnível $s$ do nível 2 e dois elétrons no subnível $p$ do nível 2) formar três tipos híbridos de átomos de carbono, que são: carbono $sp^3$, cujos elétrons (um no subnível $s$ do nível 2 e três no subnível $p$ do nível 2) misturam-se para formar um novo subnível chamado $sp^3$ que permite ao carbono $sp^3$ fazer ligação simples (compartilhar um elétron) com quatro átomos diferentes; carbono $sp^2$, que possui a mesma configuração eletrônica do carbono $sp^3$, porém, apenas três elétrons agrupam-se para formar o novo subnível chamado $sp^2$, ficando um elétron "puro" (não híbrido) permitindo ao carbono $sp^2$ fazer uma ligação dupla (compartilhar dois elétrons) com um determinado átomo; e carbono $sp$, com a mesma configuração eletrônica do carbono $sp^3$, porém, apenas dois elétrons agrupam-se para formar o novo subnível, chamado $sp$, ficando dois elétrons "puros", fato que permite ao carbono $sp$ fazer duas ligações duplas ou uma ligação tripla (compartilhar três elétrons) com um determinado átomo.

Segundo Jensen (1999, p. 1), moléculas são tradicionalmente consideradas como compostos de átomos, ou, de uma forma geral, como uma coleção de partículas elétricamente carregadas, com núcleos positivos e elétrons negativos. As moléculas diferem entre si pois contém diferentes números de núcleos e elétrons podendo estar arranjados de diferentes maneiras.

Fonseca (1993, p. 83) afirma que existem três fórmulas de representação de uma molécula: fórmula eletrônica, fórmula estrutural e fórmula molecular. A fórmula eletrônica (fig. 6.2(A)) representa, além do símbolo do átomo, todos os elétrons da camada de valência através de pequenos círculos dipostos ao redor do símbolo do átomo, de forma que os elétrons compartilhados fiquem lado a lado. A fórmula estrutural (fig. 6.2(B)) evidencia a estrutura da ligação, sendo cada par de elétrons compartilhados representado

por um traço. Já a fórmula molecular (fig. 6.2(C)) mostra apenas a quantidade de átomos que formam a molécula.

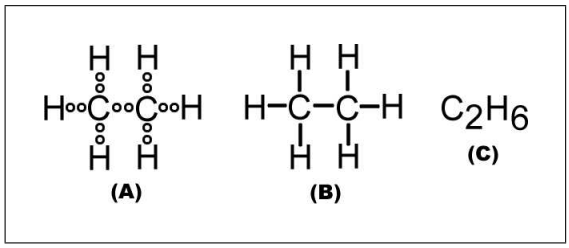



**Figura 6.2** – Fórmulas de representação molecular

Quando as ligações entre os átomos são estabelecidas e a estabilidade é atingida, formando uma molécula, os átomos estarão dispostos em um espaço tri-dimensional, atingindo um estado de conformação, denominado modelo molecular. Segundo Leach (1996, p. 413), as propriedades químicas, físicas e biológicas de toda molécula dependem criticamente da estrutura tri-dimensional adotada.

O processo de determinação de modelos moleculares é denominado modelagem molecular. Tsai (2002, p. 285) define modelagem molecular como a aplicação de computadores para gerar, manipular, calcular e predizer estruturas moleculares realísticas e suas propriedades.

O avanço da tecnologia computacional nas últimas décadas possibilitou o uso intensivo dos computadores para modelagem molecular. Young (2001, p. 3) afirma que o uso de computadores permite, de forma rápida e eficiente, a determinação e o estudo de modelos moleculares sem a (ou antes da) síntese da molécula. Esta síntese, além de requerer matéria-prima e meses de laboratório, pode resultar em resíduos tóxicos.

Leach (1996) apresenta as duas maneiras existentes para realização da modelagem molecular: mecânica quântica e mecânica molecular. A mecânica quântica considera explicitamente a presença dos elétrons e suas movimentações para confecção do modelo molecular. Por um lado, este fato permite o estudo preciso de propriedades que dependem diretamente da distribuição eletrônica e a investigação de reações químicas em que ligações são formadas ou quebradas. Por outro lado, considerar os elétrons na determinação do modelo molecular requer grande quantidade de cálculos, consumindo tempo considerável.

Já a mecânica molecular, que será apresentada com mais detalhes na seção 6.2, desconsidera a movimentação dos elétrons e considera apenas a posição dos núcleos dos átomos para determinação do modelo molecular, simplificando desta forma os cálculos e reduzindo o tempo de processamento computacional.

## 6.2 MECÂNICA MOLECULAR

A mecânica molecular, também chamada de modelo de campo de força, especifica as relações entre os átomos que compõem a molécula através de conceitos da mecânica clássica (por exemplo: sistemas massa-mola). Diferentemente da mecânica quântica, a mecânica molecular ignora a movimentação dos elétrons, e, segundo Tsai (2002, p. 288), é baseada nos seguintes princípios: 1) o núcleo e os elétrons de cada átomo são agrupados em uma única partícula e tratados como bolas esféricas; 2) ligações entre estas partículas são vistas como molas; 3) interações entre estas partículas são tratadas utilizando funções derivadas da mecânica clássica; 4) cada função atua sob parâmetros experimentais, que descrevem as interações entre determinados tipos de átomos; 5) a soma das interações entre os átomos da molécula determina o estado de conformação do modelo molecular.

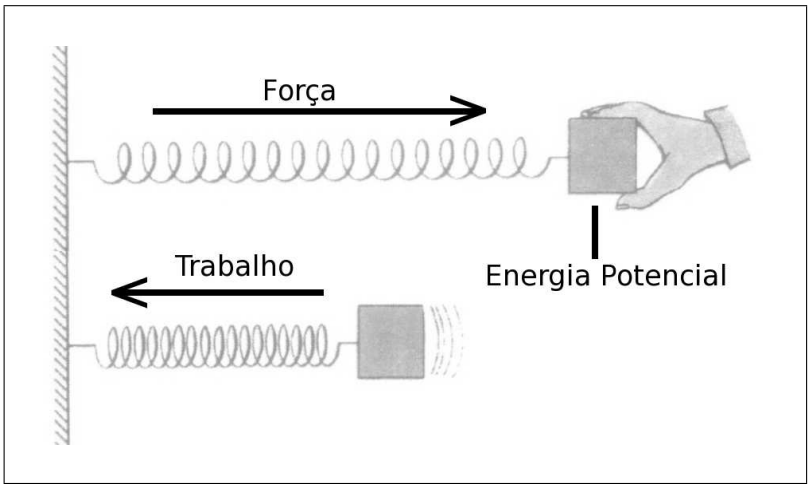
A partir destes princípios, é possível especificar a equação (6.1) que corresponde a equação geral que determina o modelo molecular de uma série de moléculas (limitadas apenas pela existência ou não de parâmetros experimentais).

$$E = E_{str} + E_{bend} + E_{tors} + E_{non-bonded} \tag{6.1}$$

Nesta equação, o termo $E$ representa energia potencial, logo, infere-se que a energia potencial total da molécula ($E$) é igual a soma das energias de estiramento das ligações entre dois átomos ($E_{str}$), de dobramento de ligação entre três átomos ($E_{bend}$), de torção entre planos ($E_{tors}$) e de interações entre átomos não ligados diretamente ($E_{non-bonded}$) (LEACH, 1996, p. 132). Estas energias são detalhadas nas seções 6.2.1, 6.2.2, 6.2.3 e 6.2.4, respectivamente.

Inicialmente, é preciso apresentar o conceito de energia potencial. Bonjorno et al.

(1993, p. 139) definem energia potencial como sendo a energia armazenada pelos corpos devido as suas posições. Como exemplo, cita-se um corpo preso a uma mola sendo "puxado" por uma determinada força (fig. 6.3). Quanto mais afastado o corpo está, mais esticada está a mola, e consequentemente, mais energia potencial o corpo possui. Assim que o objeto é solto, a energia armazenada fará o corpo se mover, produzindo trabalho. Uma propriedade importante que deve ser destacada é a constante elástica da mola que determina sua "flexibilidade". A quantidade de energia potencial é diretamente dependente do valor da constante elástica e da força aplicada para deslocar o corpo.




Fonte: adaptado de Bonjorno et al. (1993, p. 144)

**Figura 6.3** – Energia potencial

Da mesma forma que o sistema corpo(massa)-mola apresentado anteriormente, as interações entre os átomos de uma molécula apresentam constantes de referência. No caso da ligação entre, por exemplo, dois átomos de carbono $sp^3$, verifica-se experimentalmente que estes tem como distância de ligação de referência 1,523 Å(a unidade de medida de distância entre dois átomos é o angstrom, de símbolo Å, que corresponde a bilionésima parte de um metro) (LEACH, 1996, p. 138). Sendo assim, quando estes dois átomos têm como distância de ligação um valor diferente desta referência, apresentam energia potencial, analogamente à mola esticada/comprimida.

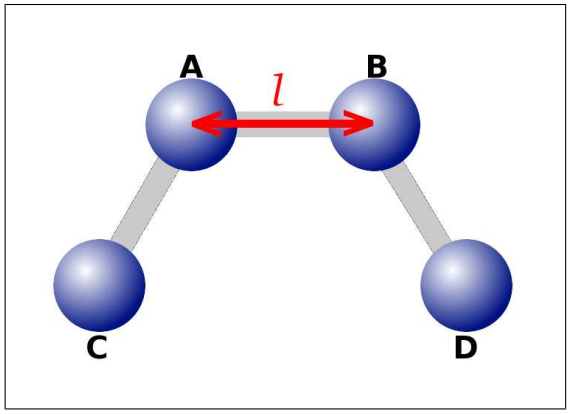
Sob a ótica da mecânica molecular, a modelagem molecular objetiva encontrar uma disposição espacial dos átomos de forma a estabilizar o sistema, ou seja, um arranjo que minimiza a energia potencial total da molécula (JENSEN, 1999, p. 8).

### 6.2.1 Energia de Estiramento

Todos os pares de átomos A e B, quando ligam-se diretamente, tendem a ficar separados por uma determinada distância $l$, que varia em função do tipo de átomo A e B. Quando esta distância é aumentada ou diminuída (em ambos os casos, diz-se estirada) em relação a distância de referência, forma-se energia potencial. A equação (6.2) define a energia de estiramento entre um par de átomo A e B (JENSEN, 1999, p. 8).

$$E_{str} = k^{AB}(l^{AB} - l_0^{AB})^2 \tag{6.2}$$

Esta equação apresenta um termo variável ($l^{AB}$) e duas constantes ($l_0^{AB}$ e $k^{AB}$). O termo $l^{AB}$ corresponde a distância $l$ entre os átomos A e B. $l_0^{AB}$ é a distância de referência para a ligação A-B. $k^{AB}$ é a constante de força para a ligação A-B. A fig. 6.4 ilustra graficamente a distância de ligação entre os átomos A e B, denotada por $l$, de onde surge a energia de estiramento. Apesar de estar sendo representada apenas entre os átomos A e B, a energia de estiramento ocorre entre todos os pares de átomos ligados.




Fonte: adaptado de CENTER FOR MOLECULAR MODELING (1996)

**Figura 6.4** – Energia de estiramento

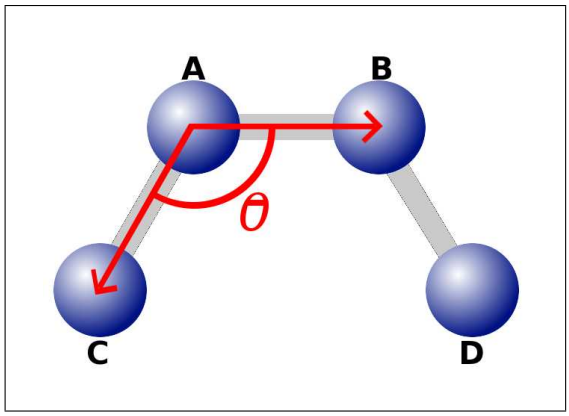
### 6.2.2 Energia de Dobramento

Entre cada três átomos A, B e C ligados, forma-se um ângulo. A energia de dobramento é a energia formada quando há um aumento ou diminiução (diz-se dobramento) deste ângulo em relação ao ângulo de referência para os átomos em questão. A

equação (6.3) define a energia de dobramento entre três átomos A, B e C (JENSEN, 1999, p. 11).

$$E_{bend} = k^{ABC}(\theta^{ABC} - \theta_0^{ABC})^2 \tag{6.3}$$

O termo $\theta^{ABC}$ corresponde ao ângulo $\theta$ formado entre os átomos A, B e C. $\theta_0^{ABC}$ é o ângulo de referência para os átomos A-B-C. $k^{ABC}$ é a constante de força para o ângulo formado entre os átomos A-B-C. A fig. 6.5 ilustra gráficamente o ângulo formado entre os átomos A, B e C, denotada por $\theta$, de onde surge a energia de dobramento. Apesar de estar sendo representada apenas entre os átomos A, B e C, a energia de dobrameto ocorre a cada conjunto de três átomos ligados.




Fonte: adaptado de CENTER FOR MOLECULAR MODELING (1996)

**Figura 6.5** – Energia de dobramento

### 6.2.3 Energia de Torção

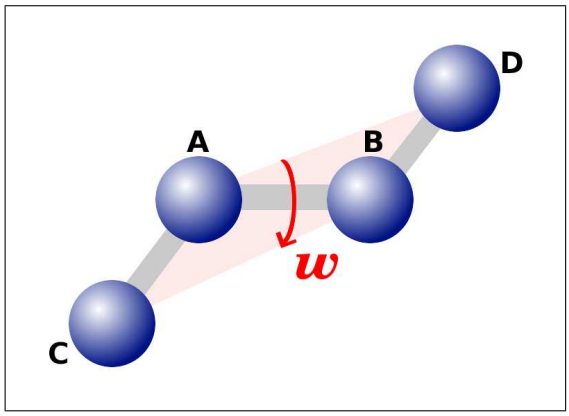
Da mesma forma que se forma energia quando há variação no ângulo de ligação entre os átomos com relação ao ângulo de referência, também há formação de energia quando há variação do ângulo formado entre os diversos planos existentes em um modelo molecular.

A cada conjunto de quatro átomos ligados A, B, C e D, é possível formar, geometricamente, dois planos. O primeiro plano é formado pelos átomos A, B e C e o segundo pelos átomos B, C e D. O ângulo de torção (também chamado de ângulo diedro) é o ângulo formado por estes dois planos (LEACH, 1996, p. 3). Jensen (1999, p. 15) observa

um aspecto importante da energia de torção: ela deve ser periódica, ou seja, a energia retorna a um mesmo valor quando os planos são rotacionados um determinado período (por exemplo, $360^o$). A equação (6.4) define a energia de torção formada entre os átomos A, B, C e D.

$$E_{tors} = \frac{V_1}{2}(1 + cos\omega) + \frac{V_2}{2}(1 + cos2\omega) + \frac{V_3}{2}(1 + cos3\omega) \tag{6.4}$$

Esta equação analisa a energia de torção em três períodos: $360^o$, $180^o$ e $120^o$. O termo $\omega$ corresponde ao ângulo formado entre os planos A-B-C e B-C-D. Os termos $V_1$, $V_2$ e $V_3$ são constantes e correspondem ao máximo de energia encontrado ao rotacionar o ângulo de torção em cada um dos respectivos períodos. A fig. 6.6 ilustra graficamente o ângulo de torção formado entre os átomos A, B, C e D, denotada por $\omega$, de onde surge a energia de torção, que ocorre a cada conjunto de quatro átomos ligados.




Fonte: adaptado de CENTER FOR MOLECULAR MODELING (1996)

**Figura 6.6** – Energia de ângulo de torção

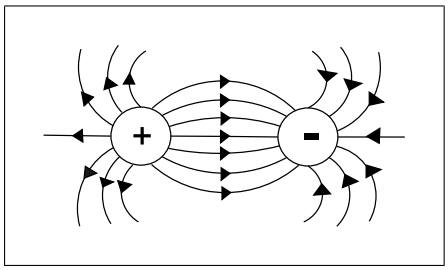
### 6.2.4 Energia de interações entre átomos não ligados

A energia formada a partir de interações entre átomos não ligados diretamente é composta por dois tipos de energia (conforme apresenta a equação (6.5)): de van der Waals($E_{vdw}$) e eletrostática ($E_{el}$). Ambas energias estão associadas a atração/repulsão dos átomos, apenas diferenciam-se pelos fatores que realizam a atração/repulsão.

$$E_{nonbonded} = E_{vdw} + E_{el} \tag{6.5}$$

A energia de van der Waals é formada a partir da repulsão e atração dos átomos. Os elétrons, por serem carregados negativamente, tendem a repelirem-se. Logo, quando dois átomos aproximan-se, há repulsão por parte de seus elétrons. Porém, a medida que os átomos se afastam, surge atração entre os mesmos. Esta atração é explicada em função de momentos dipolares.

Um dipolo consiste de um par de cargas elétricas de sinais opostos (uma positiva e outra negativa). Estas duas cargas produzem ao seu redor um campo de força, conforme é apresentado na fig. 6.7. Nesta figura, a direção da seta indica a ação do campo de força (carga negativa "atrai" carga positiva e a carga positiva deixa-se "atrair" pela carga negativa). No caso de um átomo, um dipolo é formado pelos elétrons (negativos) e pelos prótons (positivos). Como os elétrons estão sempre em movimento, a ação do campo de força também está. Dessa forma, em determinado momento (quando o "lado positivo" do átomo está próximo do "lado negativo" de outro átomo) acontece atração.




Fonte: adaptado de Bonjorno et al. (1993, p. 373)

**Figura 6.7** – Dipolo elétrico

A equação (6.6) define a energia de van der Waals entre um par de átomos A e B não ligados (LEACH, 1996, p. 174). Nesta equação, o termo $r_{AB}$ corresponde a distância entre os átomos A e B. $\sigma_{AB}$ é constante e representa a distância de ligação entre os átomos A e B quando a energia de van der Waals atinge o valor zero. $\epsilon$ também é constante e representa a "maciez" (quão próximos podem ficar) dos átomos em questão.

$$E_{vdw} = 4\epsilon\left[\left(\frac{\sigma_{AB}}{r_{AB}}\right)^{12} - \left(\frac{\sigma_{AB}}{r_{AB}}\right)^{6}\right] \tag{6.6}$$

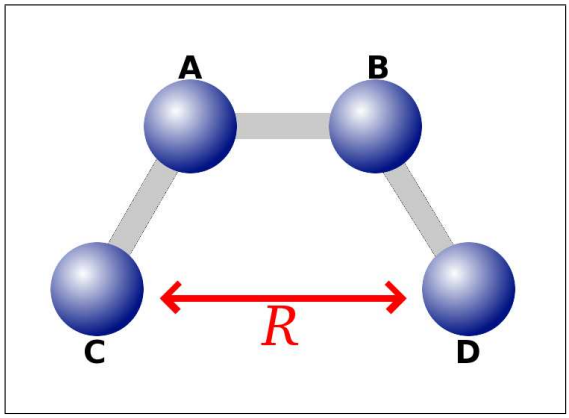
A energia eletrostática forma-se em função da capacidade que um átomo tem de

atrair ou repelir elétrons, capacidade esta determinada pela tendência do átomo de compartilhar elétrons da camada de valência. A equação (6.7) define a energia eletrostática entre um par de átomos A e B (LEACH, 1996, p. 148).

$$E_{el} = \frac{q_A q_B}{4\pi\epsilon_0 r_{AB}} \tag{6.7}$$

O termo $r_{AB}$ corresponde a distância entre os átomos A e B. Os termos $q_A$ e $q_B$ são constantes e representam a carga (capacidade) que os átomos A e B, respectivamente, possuem de atrair elétrons. O termo $\epsilon_0$ corresponde a constante dielétrica, que no vácuo, corresponde ao valor 1,0.

A fig. 6.8 apresenta a representação gráfica da ocorrência de energia entre átomos não ligados, enfatizando justamente o fato de que este tipo de energia somente ocorre entre átomos não ligados e separados por no mínimo três ligações.




Fonte: adaptado de CENTER FOR MOLECULAR MODELING (1996)

**Figura 6.8** – Energia de átomos não ligados

### 6.2.5 Parametrização

Conforme verifica-se nas equações apresentadas nas seções anteriores, para determinar a energia molecular, além da necessidade de conhecer a atual disposição geométrica dos átomos da molécula (para desta forma determinar os valores das distâncias e ângulos de ligação, por exemplo), é preciso também conhecer os diversos parâmetros (constantes) envolvidos em cada equação, para desta forma, poder "parametrizar" corretamente as

equações.

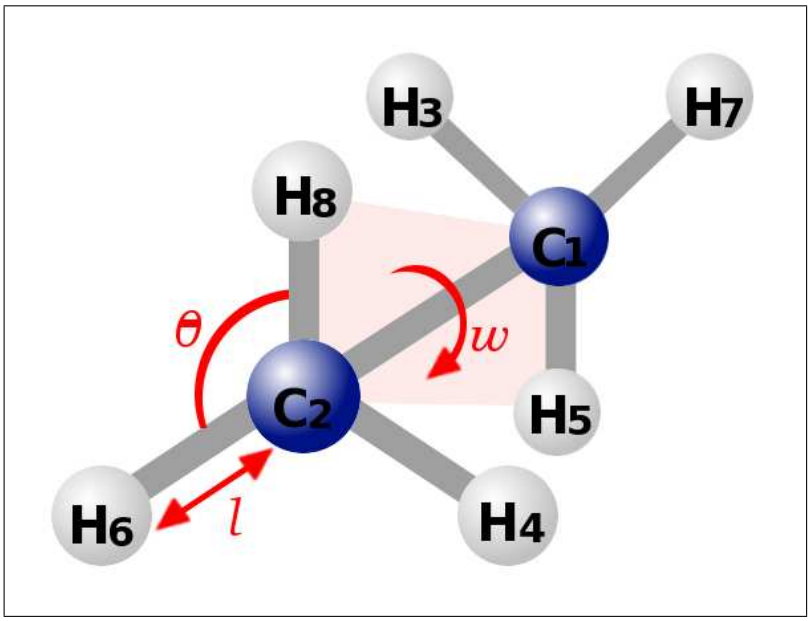
Um importante conceito para a parametrização é o de *tipo de átomo*. O tipo de átomo é determinado em função das ligações efetuadas. Sendo assim, se um átomo de um determinado elemento pode realizar ligações de diferentes maneiras (é o caso da hibridação do carbono, apresentada na seção 6.1), devem haver diferentes tipos de átomo para representar o átomo em questão (LEACH, 1996, p. 135).

Leach (1996, p. 135) observa uma propriedade importante dos parâmetros das equações: a transferibilidade. A transferibilidade consiste do fato de que um determinado parâmetro pode sempre ser utilizado para representar a relação entre determinados átomos, independente da molécula em questão. Por exemplo, os parâmetros que especificam a relação dos átomos A1-A2 em uma molécula M1 são os mesmos que (transferíveis para) a relação A1-A2 em uma molécula M2. Leach (1996, p. 135) também observa que os conjuntos de parâmetros mais amplamente utilizados para determinação de modelos moleculares para moléculas pequenas são: Molecular Mechanics 2 (MM2) e Molecular Mechanics 3 (MM3).

Os parâmetros MM2 e MM3 foram desenvolvidos pelo ALLINGER'S MOLECULAR MECHANICS RESEARCH LAB (1995). No conjunto MM2 existem 69 tipos de átomos. O conjunto MM3, que é um aprimoramento e extensão do conjunto MM2 apresenta 156 tipos de átomos. Apenas para o elemento carbono, o conjunto MM3 especifica 14 tipos de átomo.

### 6.2.6 Representação Geométrica

Segundo Leach (1996, p. 5), o modelo geométrico tri-dimensional mais utilizado para representar um modelo molecular é o de "bolas e palitos", onde cada átomo é representado por uma bola (esfera) e as ligações entre os átomos são representadas por "palitos" conectando as esferas. Um exemplo de modelo molecular tri-dimensional é apresentado na fig. 6.9. Trata-se de uma molécula de etano, de fórmula molecular $C_2H_6$, gerado a partir dos parâmetros do conjunto MM2.




Fonte: adaptado de Leach (1996, p. 3)

**Figura 6.9** – Modelo molecular do etano

A maneira mais apropriada para especificar um modelo molecular tri-dimensional é utilizar uma matriz Z. Nesta matriz, as interações entre os átomos são representadas utilizando coordenadas internas, onde a posição de cada átomo é descrita em relação aos outros átomos. Em uma matriz Z, cada linha representa as interações de um determinado átomo (LEACH, 1996, p. 2). A tab. 6.2 apresenta um exemplo de matriz Z, que corresponde ao modelo molecular do etano (representado graficamente na fig. 6.9) gerado a partir dos parâmetros do conjunto MM2.

**Tabela 6.2** – Matriz Z para o modelo molecular do etano

| $a_1$ | $l(a_1 - a_2)$ | $a_2$ | $\theta(a_1 - a_2 - a_3)$ | $a_3$ | $\omega(a_1 - a_2 - a_3 - a_4)$ | $a_4$ |
|---|---|---|---|---|---|---|
| C1 | | | | | | |
| C2 | 1,54 | C1 | | | | |
| H3 | 1,0 | C1 | 109,5 | C2 | | |
| H4 | 1,0 | C2 | 109,5 | C1 | 180,0 | H3 |
| H5 | 1,0 | C1 | 109,5 | C2 | 60,0 | H4 |
| H6 | 1,0 | C2 | 109,5 | C1 | -60,0 | H5 |
| H7 | 1,0 | C1 | 109,5 | C1 | 180,0 | H6 |
| H8 | 1,0 | C2 | 109,5 | C1 | 60,0 | H7 |

Fonte: adaptado de Leach (1996, p. 2)

Na primeira linha da matriz Z é definido o átomo C1, que corresponde a um átomo

de carbono. O átomo C2 também é um carbono e está a uma distância de 1,54Å do átomo C1 (colunas 2 e 3). O átomo H3 é um hidrogênio e está separado do átomo C1 por 1,0Å. O ângulo formado pelos átomos C2-C1-H3 é $109,5^o$, informação esta que é especificada nas colunas 4 e 5. O quarto átomo é um hidrogênio, distanciado 1,0Å do átomo C2, o ângulo H4-C2-C1 é $109,5^o$ e o ângulo de torção dos átomos H4-C2-C1-H3 é $180^o$.

## 6.3 ESPECIFICAÇÃO E IMPLEMENTAÇÃO

Para verificar a viabilidade da resolução do problema de modelagem molecular em um ambiente distribuído e considerando-se o fato de que a mesma consiste de um problema de otimização de restrições, optou-se por desenvolver um estudo de caso aplicando o algoritmo Adopt para obter o modelo molecular da molécula de etano, apresentada na fig. 6.9. Considerando-se as interações entre os átomos apresentadas na seção 6.2, a molécula do etano apresenta 37 interações. Estas iterações são apresentadas no quadro 6.1.

| **Estiramento** | **Dobramento** | **Torção** | **Não ligados** |
|---|---|---|---|
| C1-H3 | C1-C2-H4 | H3-C1-C2-H4 | H3-C1-C2-H4 |
| C1-H7 | C1-C2-H8 | H3-C1-C2-H6 | H3-C1-C2-H6 |
| C1-H5 | C1-C2-H6 | H3-C1-C2-H8 | H3-C1-C2-H8 |
| C1-C2 | C2-C1-H3 | H5-C1-C2-H4 | H5-C1-C2-H4 |
| C2-H4 | C2-C1-H5 | H5-C1-C2-H6 | H5-C1-C2-H6 |
| C2-H6 | C2-C1-H7 | H5-C1-C2-H8 | H5-C1-C2-H8 |
| C2-H8 | H3-C1-H5 | H7-C1-C2-H4 | H7-C1-C2-H4 |
| | H3-C1-H7 | H7-C1-C2-H6 | H7-C1-C2-H6 |
| | H5-C1-H7 | H7-C1-C2-H8 | H7-C1-C2-H8 |
| | H4-C2-H6 | | |
| | H4-C2-H8 | | |
| | H6-C2-H8 | | |

**Quadro 6.1** – Interações entre os átomos da molécula de etano

Quando a energia da molécula de etano é minimizada, gerando seu modelo molecular tri-dimensional, os átomos da molécula estão dispostos de acordo com a matriz Z da tab. 6.3. Esta disposição, obtida através do *software* TINKER (PONDER, 2004a) utilizando o conjunto de parâmetros MM3, servirá de referencial para os valores obtidos a partir da resolução do COP de modelagem molecular do etano através do algoritmo Adopt.

**Tabela 6.3** – Matriz Z de referência para o modelo molecular do etano

| $a_1$ | $l(a_1 - a_2)$ | $a_2$ | $\theta(a_1 - a_2 - a_3)$ | $a_3$ | $\omega(a_1 - a_2 - a_3 - a_4)$ | $a_4$ |
|---|---|---|---|---|---|---|
| C1 | | | | | | |
| C2 | 1,5313 | C1 | | | | |
| H3 | 1,1131 | C1 | 111,408 | C2 | | |
| H4 | 1,1131 | C2 | 111,408 | C1 | 179,996 | H3 |
| H5 | 1,1131 | C1 | 111,408 | C2 | 59,997 | H4 |
| H6 | 1,1131 | C2 | 111,408 | C1 | 300,000 | H5 |
| H7 | 1,1131 | C1 | 111,408 | C1 | 179,999 | H6 |
| H8 | 1,1131 | C2 | 111,408 | C1 | 59,998 | H7 |

Fonte: adaptado de Ponder (2004a)

Tendo definidas as interações entre os átomos, pode-se então especificar um COP para a modelagem molecular. Como para cada uma destas interações existe uma equação (que define a energia), e que em cada uma destas equações existe um termo variável, optou-se por especificar cada uma destas interações como uma variável do COP, sendo que o domínio de cada uma destas variáveis dependerá do tipo da relação (por exemplo, o domínio das variáveis das interações de dobramento será dado em graus). A função de custo que irá atuar sobre as variáveis são as equações que definem a energia, pois estas são minimizadas quando cada variável apresenta seu valor ideal.

Antes de iniciar a especificação do COP, faz-se necessário comentar os parâmetros utilizados em cada uma das equações e sua relação com os domínios. Os parâmetros utilizados foram os do conjunto de parâmetros MM3, retirados de Ponder (2004b), onde estão dispostos de forma tabular, separados por tipos de átomos e de interação. Conforme descrito na seção 6.2.5, o conjunto de parâmetros MM3 especifica 156 tipos de átomos. Cada um destes tipos de átomo é representado por um valor inteiro. O quadro 6.2 apresenta a definição dos tipos de átomos de carbono e hidrogênio que formam a molécula de etano.

| *Tipo do átomo* | *Símbolo* | *Observação* |
|---|---|---|
| 1 | C | CSP3 ALKANE |
| 5 | H | EXCEPT ON H, O, S |

Fonte: adaptado de Ponder (2004b)

**Quadro 6.2** – Tipos de átomo no conjunto MM3

Os parâmetros para as interações de estiramento são apresentados em Ponder (2004b) conforme ilustra o quadro 6.3. Os parâmetros para as demais interações são dispostos tabularmente da mesma forma como os deste quadro. Devido ao fato de que, para as interações entre átomos não ligados, o conjunto de parâmetros MM3 apresenta parâmetros não diretamente aplicáveis nas equações apresentadas na seção 6.2.4 sem sofrerem transformações através de equações não descritas neste trabalho, estas interações não serão consideradas, acarretando na obtenção de um modelo molecular aproximado, mas não inválido para a avaliação da especificação da modelagem molecular como COP e sua resolução com o algoritmo Adopt.

| *Tipo do átomo* | *Tipo do átomo* | $k$ | $l_0$ |
|---|---|---|---|
| 1 | 1 | 4,4900 | 1,5247 |
| 1 | 5 | 4,7400 | 1,1120 |

Fonte: adaptado de Ponder (2004b)

**Quadro 6.3** – Parâmetros de estiramento

Em Ponder (2004b), para cada tipo de interação pode existir um Fator de Multiplicação (FM), que deve ser multiplicado pelo valor do parâmetro para obter seu valor real. Para o exemplo apresentado no quadro 6.3, o FM para o parâmetro $k$ é 71,94. Desta maneira, o valor real do parâmetro $k$ para a interação entre os tipos de átomos 1-1 é 323,0106 e para a interação entre os tipos de átomos 1-5 é 340,9956. O parâmetro $l_0$ não apresenta FM.

Considerando o fato de que na modelagem molecular as variáveis das equações assumem valores reais (ou seja, seus valores são obtidos a partir de domínios contínuos e infinitos) e que o algoritmo Adopt opera somente com valores inteiros para os domínios das variáveis de um COP (ou seja, com domínios discretos e finitos), torna-se necessário

"discretizar" os domínios das variáveis das equações para então serem utilizados como domínios das variáveis do COP. Para isto, a cada variável de equação foi definido um Fator de Discretização (FD) que "transforma" um valor contínuo em discreto. Nas equações em que o termo variável é determinado em função de um parâmetro que especifica o valor de referência para o mesmo (o que ocorre nas equações de energia de estiramento e dobramento), o FD é determinado a partir das casas decimais do parâmetro, com o objetivo de "removê-las". Já para a equação de energia de torção (que não apresenta parâmetro com valor de referência), optou-se por definir um FD que proporciona-se uma precisão de dois décimos.

A tab. 6.4 apresenta os parâmetros de estiramento para as interações na molécula de etano, já considerados o FM e apresentando o FD. Nesta tabela, o termo $FM_k$ corresponde ao FM de $k$. O termo $k * FM_k$ é $k$ multiplicado por $FM_k$. Já o termo $FD_l$ é o FD do termo variável $l$, que define os valores dos domínios (correspondentes as distâncias que o estiramento pode tomar) com precisão de quatro décimos, conforme o parâmetro de referência $l_0$.

**Tabela 6.4** – Parâmetros de estiramento com FM e FD para molécula de etano

| *interação* | $k$ | $l_0$ | $FM_k$ | $k * FM_k$ | $FD_l$ |
|---|---|---|---|---|---|
| 1-1 | 4,4900 | 1,5247 | 71,94 | 323,0196 | $10^4$ |
| 1-5 | 4,7400 | 1,1130 | 71,94 | 340,9956 | $10^4$ |

Na tab. 6.5 são vistos os parâmetros de dobramento para as interações na molécula de etano, com os respectivos FM ($FM_K$) e FD ($FD_l$), sendo este último utilizado para especificar os valores dos domínios (que correspondem aos ângulos que a interação pode assumir) com precisão de três décimos, conforme o parâmetro de referência $\theta_0$.

**Tabela 6.5** – Parâmetros de dobramento com FM e FD para molécula de etano

| *interação* | $k$ | $\theta_0$ | $FM_k$ | $k * FM_k(\cong)$ | $FD_\theta$ |
|---|---|---|---|---|---|
| 1-1-5 | 0,590 | 109,800 | 0,02191418 | 0.0129293662 | $10^3$ |
| 5-1-5 | 0,550 | 107,600 | 0,02191418 | 0.012052799 | $10^3$ |

Por fim, a tab. 6.6 contém os parâmetros de torção para as interações na molécula de etano, sendo $FM_{V_n}$ o FM dos termos $V_n$ e $FD_\omega$ o fator de discretização utilizado para especificar os valores dos domínios, que correspondem ao ângulo de torção que a interação pode assumir com uma precisão de dois décimos.

**Tabela 6.6** – Parâmetros de torção com FM para molécula de etano

| *interação.* | $V_1$ | $V_2$ | $V_3$ | $FM_{V_n}$ | $V_1 * FM$ | $V_2 * FM$ | $V_3 * FM$ | $FD_\omega$ |
|---|---|---|---|---|---|---|---|---|
| 5-1-1-5 | 0,0 | 0,0 | 0,238 | 0,5 | 0,0 | 0,0 | 0,119 | $10^2$ |

Após a definição das variáveis, dos domínios e dos parâmetros, foi possível especificar uma classe para o COP de modelagem molecular. Esta classe foi chamada de `EthaneCOP`. Os parâmetros e os FD foram especificados como atributos estáticos (constantes) nesta classe, conforme apresenta o quadro 6.4 (página 102).

No método construtor desta classe são definidos os domínios, as variáveis e as funções de custo. A definição dos domínios é mostrada no quadro 6.5 (página 102). São definidos um domínio para cada interação entre diferentes tipos de átomos, sendo que em todos os casos, os valores dos domínios consideram o FD. Devido ao fato de que o algoritmo Adopt requer domínios finitos, optou-se por compor um intervalo de valores para os domínios com base no valor do parâmetro de referência (nos casos de estiramento e dobramento) e com base também nos valores para as variáveis das equações determinados por Ponder (2004a) para o modelo molecular do etano (já que verificou-se que os mesmos, no caso do etano, não se distanciam muito do valor de referência). Portanto, foram considerados para os limites inferior e superior deste intervalo, 20% acima e abaixo do valor de referência. Para o domínio da energia de torção, como esta não apresenta ângulo de referência, fica o intervalo do círculo trigonométrico (em graus) como sendo o domínio das variáveis que representam este tipo de energia.

```
1 public class EthaneCOP extends COP {
2   //  stretch parameters for 1-1 atoms interactions
3   private static final double C_C_K = 323.0106;  // k*FM
4   private static final double C_C_L = 1.5247;    // l0
5   private static final int C_C_DIVIDER = 10000;  // FD

7   //  stretch parameters for 1-5 atoms interactions
8   private static final double C_H_K = 340.9956;  // k*FM
9   private static final double C_H_L = 1.1120;    // l0
10  private static final int C_H_DIVIDER = 10000;  // FD

12  // bend parameters for 1-1-5 atoms interactions
13  private static final double H_C_C_K = 0.0129293662;  // k*FM
14  private static final double H_C_C_T = 109.800;       // θ0
15  private static final int  H_C_C_DIVIDER = 1000;      // FD

17  // bend parameters for 1-5-1 atoms interactions
18  private static final double H_C_H_K = 0.012052799;  // k*FM
19  private static final double H_C_H_T = 107.600;      // θ0
20  private static final int H_C_H_DIVIDER = 1000;      // FD

22  // torsion parameters for 5-1-1-5 atom interactions
23  private static final double H_C_C_H_V1 = 0.0;      // V1
24  private static final double H_C_C_H_V2 = 0.0;      // V2
25  private static final double H_C_C_H_V3 = 0.119;    // V3
26  private static final int H_C_C_H_V3_DIVIDER = 1;   // FD

28  // the constant term 1 present in the torsinal equation
29  private static final int TORSION_EXPANSION_CONSTANT = 1;
```

**Quadro 6.4** – `EthaneCOP`: definição da classe e parâmetros MM3

```
29  public EthaneCOP(){
30    super("EthaneCOP Molecule");
31    // domain for stretch interactions between 1-1 atom types
32    IntegerDomain C_C_STRETCH_DOMAIN =
33                  new IntegerDomain("Bond C-C", 12197, 18296);
34    // domain for stretch interactions between 1-5 atom-types
35    IntegerDomain H_C_STRETCH_DOMAIN =
36                  new IntegerDomain("Bond H-C", 8896, 13344);
37    // domain for bend interactions between 1-1-5 atom types
38    IntegerDomain H_C_C_BEND_DOMAIN =
39              new IntegerDomain("Angle H-C-C", 87840, 131760);
40    // domain for bend interactions between 5-1-5 atom types
41    IntegerDomain H_C_H_BEND_DOMAIN =
42              new IntegerDomain("Angle H-C-H", 86080, 129120);
43    // domain for torsion interactions between 5-1-1-5 atom types
44    IntegerDomain H_C_C_H_TORSION_DOMAIN =
45              new IntegerDomain("Torsion H-C-C-H", 0, 36000);
```

**Quadro 6.5** – `EthaneCOP`: definição dos domínios das variáveis

A definição das variáveis é apresentada, em partes, no quadro 6.6. Para cada variável está associado um domínio, definido anteriormente. Este quadro mostra apenas a definição de uma variável para cada tipo de domínio, sendo que as demais seguem o mesmo princípio e por isto foram omitidas.

```
45      // stretch between c1-c2 atoms
46      Variable S_C1_C2 = new Variable("S_C1_C2");
47      S_C1_C2.setDomain(C_C_STRETCH_DOMAIN);
48      addVariable(S_C1_C2);

50      // stretch between c1-h3 atoms
51      Variable S_C1_H3 = new Variable("S_C1_H3");
52      S_C1_H3.setDomain(H_C_STRETCH_DOMAIN);
53      addVariable(S_C1_H3);

55      // bend between h3-c1-c2 atoms
56      Variable B_H3_C1_C2 = new Variable("B_H3_C1_C2");
57      B_H3_C1_C2.setDomain(H_C_C_BEND_DOMAIN);
58      addVariable(B_H3_C1_C2);

60      // bend between h3-c1-h5 atoms
61      Variable B_H3_C1_H5 = new Variable("B_H3_C1_H5");
62      B_H3_C1_H5.setDomain(H_C_H_BEND_DOMAIN);
63      addVariable(B_H3_C1_H5);

65      // torsion between h3-c1-c2-h4 atoms
66      Variable T_H3_C1_C2_H4 = new Variable("T_H3_C1_C2_H4");
67      T_H3_C1_C2_H4.setDomain(H_C_C_H_TORSION_DOMAIN);
68      addVariable(T_H3_C1_C2_H4);
```

**Quadro 6.6** – `EthaneCOP`: definição das variáveis

As funções de custo são definidas em função das equações que especificam as interações. Sendo o objetivo minimizar o somatório destas funções de custo, na abordagem utilizada (de definir uma interação como variável do COP) cada agente é responsável por calcular o seu custo considerando o valor da sua variável e o valor da variável do agente acima de si na DFST (apresentada na fig. 6.10). Como os valores das variáveis do COP correspondem ao valor da variável da equação (que determina a posição geométrica dos átomos relacionados) e não ao valor da energia calculada pela equação, cada agente deve conhecer as funções de custo (equações) dos agentes acima de si para então calcular (e conseqüentemente poder minimizar) a energia. Estas funções de custo são separadas (de

acordo com a necessidade do agente conhecê-la ou não) e enviadas pelo agente gerenciador.

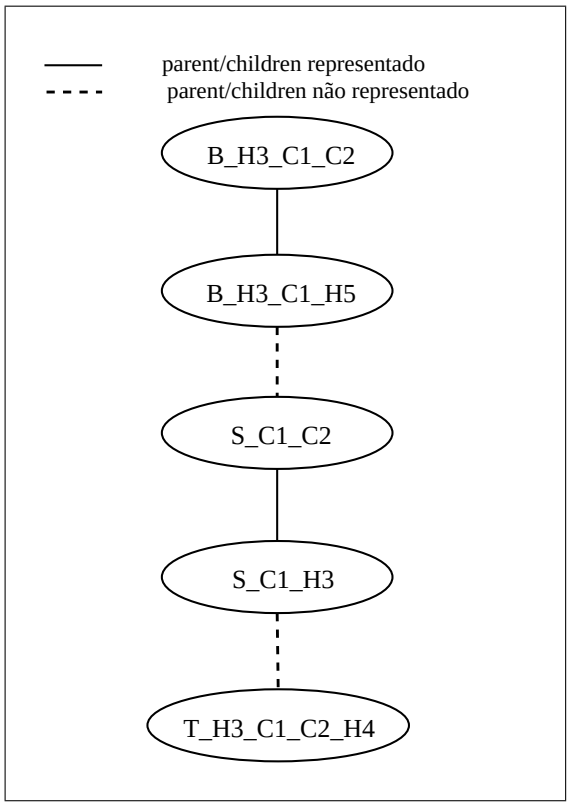



**Figura 6.10** – DFST para COP da molécula do etano

O último passo para especificar a modelagem molecular como COP é definir as funções de custo na classe `EthaneCOP`. De acordo com a mecânica molecular, a energia de estiramento é descrita pela equação (6.8).

$$E_{str} = k^{AB}(l^{AB} - l_0^{AB})^2 \tag{6.8}$$

A equação (6.9) apresenta a equação (6.8) (no caso, para interações entre os átomos C1 e C2) utilizando as constantes e a variável já definidas na classe `EthaneCOP`. Como a elevação da subtração ao quadrado é utilizada apenas para tornar eventuais valores negativos em positivos, esta operação foi substituída pelo módulo da subtração. Esta equação é codificada na classe `EthaneCOP` a partir das `Expression` disponíveis no pacote `csp.exp.numeric`, conforme apresenta o quadro 6.7.

$$E_{str} = C\_C\_K * |(S\_C1\_C2 \div C\_C\_DIVIDER) - C\_C\_L| \tag{6.9}$$

```
68      // variable value divided by FD
69      DivisionExpression c1c2value =
70            new DivisionExpression(
71              new VariableExpression(S_C1_C2),
72              new ConstantExpression(
73                new Value( new Double(C_C_DIVIDER))));

75      // variabel value subtracted by l0 param
76      SubtractionExpression c1c2l =
77            new SubtractionExpression(
78              c1c2value,
79              new ConstantExpression(
80                new Value(new Double(C_C_L))));

82      // ''turning off'' negative values
83      AbsExpression c1c2labs = new AbsExpression(c1c2l);

85      // multiplying by the k param
86      MultiplicationExpression c1c2stretch =
87            new MultiplicationExpression(
88              new ConstantExpression(
89                new Value( new Double(C_C_K)),
90              c1c2labs);

92      // create a cost function to add into COP
93      AdditionExpression cost12 =
94            new AdditionExpression(cost11, c1c2stretch);
95      addCostFunction(cost12);
```

**Quadro 6.7** – `EthaneCOP`: definição das funções de custo de estiramento

A energia de dobramento é descrita na mecânica molecular pela equação (6.10). Na equação (6.11) tem-se a equação (6.10), para interações entre os átomos H3, C1 e C2, expressa utilizando-se as constantes e a variável já definidas na classe `EthaneCOP`. A codificação da equação (6.11) na classe `EthaneCOP` é apresentada no quadro 6.8.

$$E_{bend} = k^{ABC}(\theta^{ABC} - \theta_0^{ABC})^2 \tag{6.10}$$

$$E_{bend} = H\_C\_C\_K * |(B\_H3\_C1\_C2 \div H\_C\_C\_DIVIDER - H\_C\_C\_T| \tag{6.11}$$

```
96      // variable value divided by FD
97      DivisionExpression h3c1c2value =
98            new DivisionExpression(
99              new VariableExpression(B_H3_C1_C2),
100             new ConstantExpression(
101               new Value( new Double(H_C_C_DIVIDER))));

103     // variabel value subtracted by θ0 param
104     SubtractionExpression h3c1c2t =
105           new SubtractionExpression(
106             h3c1c2value,
107             new ConstantExpression(
108               new Value( new Double( H_C_C_T))));

110     // ''turning off'' negative values
111     AbsExpression h3c1c2tabs = new AbsExpression(h3c1c2t);

113     // multiplying by the k param
114     MultiplicationExpression h3c1c2bend =
115           new MultiplicationExpression(
116             new ConstantExpression(
117               new Value( new Double(H_C_C_K))),
118               h3c1c2tabs);
```

**Quadro 6.8** – `EthaneCOP`: definição das funções de custo de dobramento

Por fim, a energia de torção, dada pela equação (6.12), é apresentada, considerando-se as constantes e a variável definidas na classe `EthaneCOP` (para interações entre os átomos H3, C1, C2 e H4) na equação (6.13). Conforme visto na tab. 6.6, os termos $V_1$ e $V_2$ são zero, logo, não foram considerados. A divisão por 2 do valor de $V_3$, conforme apresenta a equação (6.12), foi desconsiderada, não ocasionando danos ao processo de minimização da energia molecular. A respectiva codificação desta equação na classe `EthaneCOP` é vista no quadro 6.9.

$$E_{tors} = \frac{V_1}{2}(1 + cos\omega) + \frac{V_2}{2}(1 + cos2\omega) + \frac{V_3}{2}(1 + cos3\omega) \tag{6.12}$$

$$\begin{aligned} E_{tors} = H\_C\_C\_H\_V3 * ( & TORSION\_EXPANSION\_CONSTANT \\ & + cos(3 * (T\_H3\_C1\_C2\_H4 \div H\_C\_C\_H\_V3\_DIVIDER)) ) \end{aligned} \tag{6.13}$$

```
118     //  variable value divided by FD
119     DivisionExpression h3c1c2h4value =
120           new DivisionExpression(
121             new VariableExpression(T_H3_C1_C2_H4),
122             new ConstantExpression(
123               new Value( new Double(H_C_C_H_V3_DIVIDER))));

125     // variable value multiplied by the constant 3
126     MultiplicationExpression h3c1c2h4x3 =
127           new MultiplicationExpression(
128             new ConstantExpression(new Value(new Double(3.0))),
129             h3c1c2h4value);

131     // cossine of the variable value * 3
132     IntDegCosExpression h3c1c2h4x3cos =
133           new IntDegCosExpression(h3c1c2h4x3);

135     // adding constant 1
136     AdditionExpression h3c1c2h4x3cosplus1 =
137           new AdditionExpression(
138             new ConstantExpression(
139               new Value( new Double(TORSION_EXPANSION_CONSTANT))),
140             h3c1c2h4x3cos);

142     // multiplying by the V3 value
143     MultiplicationExpression h3c1c2h4v3 =
144           new MultiplicationExpression(
145             new ConstantExpression(
146               new Value( new Double(H_C_C_H_V3))),
147             h3c1c2h4x3cosplus1);

149     // create a cost function to add into COP
150     AdditionExpression cost19 =
151           new AdditionExpression(cost18, h3c1c2h4v3);
152     addCostFunction(cost19);
153   }
154 }
```

**Quadro 6.9** – `EthaneCOP`: definição das funções de custo de torção

Nos quadros 6.7 e 6.9, nas linhas 93 e 150, respectivamente, nota-se que é definido uma expressão de adição, que corresponde ao somatório das funções de custo dos agentes acima do em questão. Este somatório corresponde ao objeto `CostFunction` que deve ser enviado, pelo agente gerenciador, para o agente que contém a variável da função de custo. No quadro 6.8 não se verifica esta adição, pois esta variável será o agente raíz da DFST.

O próximo passo foi constuir uma classe *factory* e alterar o arquivo de configuração

do *framework*. Estes passos, detalhados no capítulo 3, serão omitidos nesta seção por não apresentarem diferença significativa ao exemplo apresentado na seção referenciada.

Concluidas estas etapas, o próximo passo foi a execução, que consiste basicamente em selecionar o problema, o tipo de algoritmo, a quantidade de *hosts* e adicionar a execução ao agendador. Após a execução do algoritmo Adopt sobre o problema de modelagem molecular do etano, algumas características foram observadas, as quais serão descritas a seguir.

Na primeira execução realizada, foram efetuadas alterações nos domínios das variáveis com o intuito de testar se a abordagem utilizada (de definir um agente para cada interação) funcionaria. Estas alterações consistiam na redução do intervalo de cada domínio para agilizar o processo de busca. Os domínios utilizados nesta primeira execução são apresentados no quadro 6.10.

```
30  public EthaneCOP(){
31    super("EthaneCOP Molecule");

33    // domain for stretch interactions between 1-1 atom types
34    IntegerDomain C_C_STRETCH_DOMAIN =
35                  new IntegerDomain("Bond C-C", 15246, 15247);
36    // domain for stretch interactions between 1-5 atom-types
37    IntegerDomain H_C_STRETCH_DOMAIN =
38                  new IntegerDomain("Bond H-C", 11119, 11120);

40    // domain for bend interactions between 1-1-5 atom types
41    IntegerDomain H_C_C_BEND_DOMAIN =
42              new IntegerDomain("Angle H-C-C", 109799, 109800);
43    // domain for bend interactions between 5-1-5 atom types
44    IntegerDomain H_C_H_BEND_DOMAIN =
45              new IntegerDomain("Angle H-C-H", 107599, 107600);

47    // domain for torsion interactions between 5-1-1-5 atom types
48    IntegerDomain H_C_C_H_TORSION_DOMAIN =
49              new IntegerDomain("Torsion H-C-C-H");
50    H_C_C_H_TORSION_DOMAIN.addValue(new Value ( new Integer(60)));
51    H_C_C_H_TORSION_DOMAIN.addValue(new Value ( new Integer(180)));
52    H_C_C_H_TORSION_DOMAIN.addValue(new Value ( new Integer(300)));
```

**Quadro 6.10** – `EthaneCOP`: domínios para teste do Adopt

Em todas as execuções (realizadas com apenas um *host*), o algoritmo Adopt convergiu para a solução apresentada na matriz Z da tab. 6.7. Comparando-se com a matriz

Z apresentada na tab. 6.3, nota-se que os valores são bastantes semelhantes, não sendo idênticos pois não estão sendo consideradas as interações entre átomos não ligados. Apesar de aparentemente haver grande diferença nos valores de $\omega(a_1 - a_2 - a_3 - a_4)$, esta diferença não é real, pois, como a energia de torção é periódica, ao ser aplicada uma rotação de $120^o$ nas interações H4-C2-C1-H3, H6-C2-C1-H5 e H8-C2-C1-H7 são obtidos valores próximos aos apresentados na tab. 6.3.

**Tabela 6.7** – Matriz Z para o modelo molecular do etano determinado com Adopt

| $a_1$ | $l(a_1 - a_2)$ | $a_2$ | $\theta(a_1 - a_2 - a_3)$ | $a_3$ | $\omega(a_1 - a_2 - a_3 - a_4)$ | $a_4$ |
|---|---|---|---|---|---|---|
| C1 | | | | | | |
| C2 | 1,5247 | C1 | | | | |
| H3 | 1,1120 | C1 | 109,800 | C2 | | |
| H4 | 1,1120 | C2 | 109,800 | C1 | 60,000 | H3 |
| H5 | 1,1120 | C1 | 109,800 | C2 | 60,000 | H4 |
| H6 | 1,1120 | C2 | 109,800 | C1 | 180,00 | H5 |
| H7 | 1,1120 | C1 | 109,800 | C1 | 180,00 | H6 |
| H8 | 1,1120 | C2 | 109,800 | C1 | 300,00 | H7 |

Em relação ao tempo de execução e a quantidade de mensagens trocadas, o algoritmo Adopt, resolvendo o COP de modelagem molecular do etano nesta situação de teste, apresentou os valores descritos na tab. 6.8, onde é possível observar a elevada quantidade de mensagens trocadas e considerável tempo utilizado.

**Tabela 6.8** – Estatísticas de execução do Adopt para etano

| *Execução* | *Mensagens Trocadas* | *Tempo (milisegundoss)* |
|---|---|---|
| 1 | 426435 | 419351 |
| 2 | 309767 | 330654 |
| 3 | 340376 | 387586 |
| *Média* | 358859,33 | 379197 |

Considerando este simples caso onde os domínios estão reduzidos, o tempo utilizado pelo algoritmo Adopt não foi satisfatório. Isto ocorreu porque, em função da abordagem utilizada para o problema de modelagem molecular do etano onde cada interação foi transformada em variável, a DFST gerada é linear, o que prejudica a comunicação entre

os agentes e torna a execução do algoritmo análoga a uma simples busca em profundidade com *backtrack*, não fazendo uso do paralelismo que uma execução distribuída pode oferecer. Esta abordagem também resultou em uma sobrecarca de funções de custo a serem analizadas pelos agentes conforme a distância que o agente apresenta em relação ao raíz, sendo o agente folha o maior prejudicado, tendo que computar uma função de custo gigantesca que considera os valores de todos os agentes acima de si. Estas conclusões são corroboradas quando tentou-se executar o algoritmo Adopt sobre o COP do etano considerando os valores originalmente concebidos para os domínios das variáveis, onde o Adopt não convergiu para uma solução, mesmo sendo executado por tempo aproximado de uma hora.

Existe no mercado uma significativa quantidade de *softwares* para modelagem molecular em estações de trabalho. Um deles, de distribuição gratuita (inclusive com códigos fonte) é o TINKER (PONDER, 2004a). Este software apresenta uma interface gráfica que, a partir de um arquivo contendo a disposição geométrica em coordenadas tri-dimensionais de cada átomo, gera e exibe o modelo molecular (com a correta disposição tri-dimensional) da molécula. Este *sofware* especifica o problema de modelagem molecular como sistemas de equações e utiliza métodos de aproximação interativos, computados centralizadamente, para resolvê-lo. A princípio, pensou-se que resolver o problema de modelagem molecular em um ambiente distribuído poderia reduzir o tempo utilizado para resolução através de computação centralizada. Porém, o estudo de caso apresentado provou que, dada a abordagem utilizada, a resolução do problema de modelagem molecular de maneira distribuída através do algoritmo Adopt é inviável em função do tempo computacional necessário.

# 7 CONCLUSÕES

O *framework* dynDCSP, devido a sua arquitetura em camadas, mostrou-se bastante flexível a alterações. A comprovação disto está no fato de que, para adicionar ao *framework* a capacidade de resolver COPs, bastaram poucas refatorações nas classes existentes e a inclusão de algumas novas classes. Outro ponto positivo do *framework* é a utilização de expressões lógicas e aritméticas para especificar restrições e funções de custo pois, por serem especificadas de maneira análoga as expressões utilizadas no universo matemático, sua compreensão é rápida e a utilização facilitada. A utilização da infraestrutura de comunicação SACI torna transparente ao desenvolvedor o processo de comunicação entre os agentes, realizando esta tarefa com sucesso. Em função da implementação ser feita na linguagem Java, o requisito de portabilidade é seguramente atendido.

A implementação do algoritmo Adopt foi realizada com sucesso. Testes realizados a partir de estudo de caso com problema de coloração de grafo corroboraram uma implementação correta e efetivamente funcional, já que apresentou as mesmas soluções da implementação de Modi (2003a) (o autor do algoritmo). Pesquisa realizada em Pearce (2005) (um repositório de implementações do algoritmo Adopt) mostrou que a implementação do algoritmo Adopt realizada neste trabalho é a primeira a operar em um ambiente distribuído formado por vários *hosts*. A respeito da capacidade do algoritmo Adopt resolver DCOP, algumas considerações merecem ser feitas.

Conforme Modi et al. (2003d) afirmam, o algoritmo Adopt garante solução ótima em situações onde o grafo de restrições é esparso, ou seja, onde o grau de cada vértice é pequeno. Nestas situações o funcionamento do algoritmo Adopt é muito bom, convergindo rapidamente para uma solução. Esta característica é comprovada com o estudo de caso do problema de coloração de grafo apresentado neste trabalho, composto por oito vértices sendo o maior grau igual a quatro havendo oito vértices. Neste estudo de caso, o tempo utilizado pela implementação do algoritmo Adopt deste trabalho é menor que o utilizado

pela implementação de Modi (2003a), além da quantidade de mensagens enviadas, sendo bastante superior na implementação de Modi (2003a) pois a mesma não otimiza o envio de mensagens.

Já para problemas onde o grafo de restrições é mais denso, o desempenho do algoritmo Adopt não é satisfatório. Modi (2003b) observa que, no pior caso (o de um grafo completo), o desempenho do algoritmo Adopt é exponencial ao número de variáveis, isto porque os COP fazem parte do conjunto de problemas NP-completos. Este fato foi verificado no estudo de caso do problema de modelagem molecular.

Sendo o problema de modelagem molecular definido como um somatório de energias de interações onde a solução é encontrada ao minimizar este somatório, a abordagem utilizada neste trabalho, de definir cada interação entre os átomos como uma variável do COP e as funções de custo como as equações que definem cada uma destas interações, tornou o grafo de restrições do problema muito denso, o que inviabilizou a sua resolução através do algoritmo Adopt devido ao tempo que seria necessário para resolvê-lo.

## 7.1 EXTENSÕES

A extensão mais importante deste trabalho é a realização de uma depuração cuidadosa da implementação do algoritmo Adopt. Isto porque, quando utilizados grafos de testes disponíveis em Modi (2003a), em algumas situações a execução do algoritmo pára. Este fato não acontece em todas as execuções do algoritmo em função de que o mesmo é assíncrono, o que torna a execução não determinística. A causa deste problema possívelmente está relacionada às otimizações no envio de mensagens, o que implica em uma revisão e possíveis alterações destas otimizações.

A possibilidade de especificação do problema de modelagem molecular como COP para então utilizar o algoritmo Adopt para sua resolução ainda não está totalmente descartada. A abordagem utilizada neste trabalho se mostrou inviável, mas é apenas uma das possíveis para este problema. Como sugestão de extensão, fica a tentativa de utilizar outra abordagem para especificar o problema de modelagem molecular. De início, citam-se duas possibilidades de abordagem.

A primeira delas consiste em considerar como variável cada tipo de interação entre os átomos, ou seja, cada tipo de energia. Sendo assim, haveria uma variável que representaria a energia de estiramento, outra para a energia de dobramendo, e assim sucessivamente. Esta abordagem possivelmente iria necessitar de significativas alterações na mecânica de escolha de novos valores para as variáveis do algoritmo Adopt, já que cada agente seria responsável por gerenciar interações entre diferentes tipos de átomos, com diferentes parâmetros e conseqüentemente diferentes "domínios" para as variáveis das equações.

A segunda abordagem diz respeito ao conteúdo das mensagens VALUE. Ao invés de enviar o valor que a variável assuma (que corresponde ao valor da variável da equação/função de custo), poderia ser enviado o valor computado para a função de custo. Esta abordagem otimizaria a análise do custo, já que a quantidade de funções analisadas não cresceria em relação à proximidade do agente em relação ao folha da DFST. Esta provavelmente é a abordagem que poderia ser implementada mais facilmente, já que não requer grandes alterações em relação a abordagem utilizada, porém, não eliminaria a densidade do grafo de restrições.

Ao se optar por uma ou outra sugestão de abordagem descrita anteriormente, uma extensão complementar é adicionar no *COP* de modelagem molecular os termos que correspondem a energia formada a partir das interações entre os átomos não ligados. Estes termos garantiriam uma proximidade maior dos resultados obtidos em relação aos reais valores.

Outra sugestão consiste do desenvolvimento de um mecanismo de coleta de informações dos agentes para então realizar uma animação da execução dos algoritmos do *framework* dynDCSP. Um benefício imediato desta extensão está relacionado ao auxílio que proporcionaria no processo de depuração dos algoritmos.

Por fim, uma sugestão não tão pertinente à este trabalho, mas bastante relevante ao avanço científico, é adicionar no algoritmo Adopt a capacidade de operar em domínios efetivamente contínuos, sem a necessidade de discretização.

# REFERÊNCIAS BIBLIOGRÁFICAS

ALLINGER'S MOLECULAR MECHANICS RESEARCH LAB. **Allinger's research lab home page**. Athens; Georgia, 1995. Disponível em: <http://europa.chem.uga.edu-/index.htm>. Acesso em: 26 maio 2005.

APACHE SOFTWARE FOUNDATION. **Apache Ant**. Forest Hill, Maryland, 2005. Disponível em: <http://ant.apache.org>. Acesso em: 30 jun. 2005.

BONJORNO, Regina Azenha et al. **Física fundamental**. São Paulo: FTD, 1993. 886 p.

BRUNS, Maurício. **Aplicação da técnica de satisfação de restrições distribuídas no sincronismo de semáforos de uma malha viária**. 2004. 57 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

CENTER FOR MOLECULAR MODELING. **Molecular mechanics**. Bethesda, Maryland, 1996. Disponível em: <http://cmm.info.nih.gov/modeling/guide_documents-/molecular_mechanics_document.htm>. Acesso em: 26 maio 2005.

FONSECA, Martha R. M. **Química integral**. São Paulo: FTD, 1993. 624 p.

GRUPO DE INTELIGÊNCIA ARTIFICIAL DA FURB. **DynDCSP project**. Blumenau, 2005. Disponível em: <http://www.inf.furb.br/gia/dynDCS>. Acesso em: 27 maio 2005.

HÜBNER, Jomi Fred; SICHMAN, Jaime Simão. SACI: uma ferramenta para implementação e monitoração da comunicação entre agentes. In: IBERAMIA/SBIA, 7., 2000, Atibaia, São Paulo. **Proceedings...** São Carlos: ICMC/USP, 2000. p. 47–56.

JENSEN, Frank. **Introduction to computational chemistry**. Chichester, Weinheim: Wiley, 1999. 429 p.

LEACH, Andrew R. **Molecular modelling**: principles and applications. Essex: Longman, 1996. 595 p.

MODI, Pragnesh Jay. **Adopt**. Los Angeles, California, 2003a. Disponível em: <http://www-2.cs.cmu.edu/~pmodi/adopt>. Acesso em: 30 maio 2005.

MODI, Pragnesh Jay. **Distributed constraint optimization for multiagent systems**. 2003b. 156 f. PhD Thesis – Department of Computer Science, University of Southern California, Los Angeles, California.

MODI, Pragnesh Jay; ALI, Syed Muhammad. **Distributed constraint reasoning under unreliable communication**. Los Angeles, California, 2003c. Disponível em: <http://www.cs.cmu.edu/~pmodi/papers /modi-dcr03.pdf>. Acesso em: 30 jun. 2005.

MODI, Pragnesh Jay et al. An asynchronous complete method for distributed constraint optimization. In: INTERNATIONAL JOINT CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, 2., 2003, Melbourne, Australia. **Proceedings...** New York, NY, USA: ACM Press, 2003d. p. 161–168.

PEARCE, Jonathan. **Distributed constraint optimization problem (DCOP)**. Los Angeles, California, 2005. Disponível em: <http://teamcore.usc.edu/dcop>. Acesso em: 30 jun. 2005.

PONDER, Jay William. **TINKER**: software tools for molecular design. Saint Louis, 2004a. Disponível em: <http://dasher.wustl.edu/tinke>. Acesso em: 30 maio 2005.

PONDER, Jay William. **mm3.prm**: arquivo com conjunto de parâmetros mm3. Saint Louis, 2004b. Disponível em: <ftp://dasher.wustl.edu/pub/tinker/params/mm3.prm>. Acesso em: 30 jun. 2005.

RUSSEL, Stuart J.; NORVIG, Peter. **Artificial intelligence**: a modern approach. 2nd. ed. New Jersey: Prentice Hall, 2003. 1132 p.

TRALAMAZZA, Daniel Moser. **Desenvolvimento de um algoritmo para problema de satisfação de restrição distribuída**. 2004. 37 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

TSAI, C. Stan. **An introduction to computational biochemistry**. 1st. ed. New York: Wiley-Liss, 2002. 368 p.

TSANG, Edward. **Foundations of constraint satisfaction**. London: Academic Press, 1993. 421 p.

YOKOO, Makoto. **Distributed constraint satisfaction**: foundations of cooperation in multi-agent systems. Berlin: Springer, 2001. 143 p.

YOUNG, David C. **Computational chemistry**: a pratical guide for applying techniques to real world problems. New York: John Wiley & Sons, 2001. 381 p.