UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

APLICAÇÃO DA TÉCNICA DE SATISFAÇÃO DE RESTRIÇÕES DISTRIBUÍDAS NO SINCRONISMO DE SEMÁFOROS DE UMA MALHA VIÁRIA

MAURICIO BRUNS

BLUMENAU
2005

2004/2-37

MAURICIO BRUNS

APLICAÇÃO DA TÉCNICA DE SATISFAÇÃO DE RESTRIÇÕES DISTRIBUÍDAS NO SINCRONISMO DE SEMÁFOROS DE UMA MALHA VIÁRIA

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. Jomi Fred Hübner – Orientador

BLUMENAU
2005

2004/2-37

APLICAÇÃO DA TÉCNICA DE SATISFAÇÃO DE RESTRIÇÕES DISTRIBUÍDAS NO SINCRONISMO DE SEMÁFOROS DE UMA MALHA VIÁRIA

Por

MAURICIO BRUNS

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. Mauro Marcelo Mattos, FURB
	_____
Membro:	Prof. Dr. Paulo César Rodacki Gomes, FURB

Blumenau, 15 de Janeiro de 2005

“Algo só é impossível até que alguém duvida e acaba provando o contrário.”

Albert Einstein

AGRADECIMENTOS

Primeiramente aos meus pais, Sergio e Silvia, pelo apoio e compreensão. Aos meus irmãos, Milena e Murilo, pela amizade em todos os momentos. E a minha namorada Gabriela pelas palavras de força e carinho nas horas mais difíceis.

A todos os meus amigos, que nunca me negaram ajuda e atenção quando foi necessário.

Ao meu orientador Jomi Fred Hübner pelas palavras de incentivo e por toda ajuda prestada nos momentos mais importantes.

RESUMO

Este trabalho tem como objetivo utilizar a técnica de satisfação de restrições distribuídas para tentar solucionar o problema do sincronismo de semáforos viários. Para tanto foi utilizado o *framework* DynDCSP, desenvolvido pelo Grupo de Pesquisa em Inteligência Artificial da Fundação Universidade Regional de Blumenau (FURB). Uma contribuição importante deste trabalho foi incluir três novas funcionalidades no framework: utilização de variáveis locais, restrições dinâmicas e a buscar por todas as soluções do CSP. Para implementação foi utilizada a linguagem de programação Java e o ambiente de programação Eclipse.

Palavras Chaves: DCSP; Sistema Multi-Agentes; Semáforos viários.

ABSTRACT

The objective of this research is to use the Distributed Constraint Satisfaction technique as an approach to solve the problem of traffic lights synchronization. To reach this goal the DynDCSP framework, developed by the Artificial Intelligence Research Group of Fundação Universidade Regional de Blumenau (FURB), was used. In this framework some improvements were done to make possible the using of local variables, dynamic constraints and the search of all solutions of the CSP. The Java programming language and the Eclipse development environment were used to implement the improvements of the framework.

Key words: DCSP; Multi-Agent Systems; traffic lights.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento econômico e industrial de uma nação vem seguido de um aumento na demanda pelos meios de transportes, devido ao aumentando no número de trabalhadores deslocando-se de suas residências para seus postos de trabalho, clientes trafegando entre os locais de comércio e produtos sendo transportados dos seus locais de produção até o consumidor. O aumento na demanda pode gerar, por sua vez, a necessidade do melhoramento do sistema de transportes, que deve contribuir para a diminuição dos gastos com o transporte e para a melhoria da qualidade de vida da população.

Com o crescimento econômico e a modernização da indústria automobilística houve um aumento no número de veículos que circulam pelas ruas. Este aumento acarretou em uma rápida saturação das ruas e avenidas que não foram projetadas para receber um fluxo tão intenso de veículos.

Verifica-se então que, em função deste grande crescimento, graves problemas aflingem o trânsito dos principais centros urbanos. Os congestionamentos constantes, a coordenação dos semáforos entre cruzamentos numa malha viária, os desvios de tráfego, a poluição do ar e sonora, a segurança do motorista, demora às respostas de emergências entre outros repercutem seriamente na economia como um todo (SCHMITZ, 2002, p. 1).

Segundo SincMobil (SINCMOBIL, 2002), um bom ajuste do controle de tráfego (semáforos em particular) é o ponto de partida para o bom deslocamento de veículos em malhas viárias urbanas. O cálculo manual deste ajuste é oneroso, pois envolve contagens periódicas de fluxos veiculares. Ainda assim, o desempenho não é ótimo, pois os ajustes semafóricos não variam de acordo com a variação do tráfego, e sim em função da hora programada, impedindo a correção de variações repentinas de fluxo. Por isso, têm sido adotados mundialmente sistemas automáticos que efetuam contagens em tempo real, atuando rapidamente em resposta aos padrões de fluxo. Os benefícios destes sistemas são da ordem de 10% a 15% com relação à semáforos de plano fixo bem ajustados.

Uma possível abordagem para solucionar o problema do sincronismo dos semáforos é a modelagem e resolução deste problema como um problema de satisfação de restrições.

Um problema de satisfação de restrições, em inglês Constraint Satisfaction Problem (CSP), é uma forma simples de se representar alguns problemas na Inteligência Artificial (IA). Os CSPs constituem uma classe de problemas que pode ser expressa por um conjunto de variáveis ligadas por um conjunto de restrições. As variáveis representam o estado do problema e seus domínios podem ser finitos (enumerações) ou infinitos (conjunto dos números inteiros, por exemplo). Uma restrição pode ser tanto uma simples igualdade quanto uma fórmula matemática complexa e seu papel é de restringir o valor das variáveis. A resolução de um CSP consiste em encontrar e atribuir um valor para cada variável respeitando todas as restrições impostas. Caso seja encontrado, este valor é dito consistente (TSANG, 1993, p. 1).

A utilização de Distributed Constraint Satisfaction Problem (DCSP) visa solucionar o problema Constraint Satisfaction Problem (CSP) de forma distribuída, utilizando agentes. Considerando que as principais características do problema levantado são a distribuição e a constante alteração das informações necessárias à gerência e controle de trânsito, o desenvolvimento de uma ferramenta de apoio centralizada é de difícil realização (SCHMITZ; HÜBNER, 2002, p. 2).

Contudo, pretende-se neste trabalho modelar o problema de sincronismo de sinais como um problema de satisfação de restrições e utilizar técnicas de satisfação de restrições distribuídas para resolvê-lo. Eliminando assim a necessidade de se estabelecer planos fixos para os semáforos, onde ocorre a necessidade de troca dos tempos dos semáforos para que os mesmos se adaptem as condições do tráfego conforme os horários do dia. Isto causaria uma diminuição no tempo de espera nos cruzamentos de um sistema viário, amenizando o problema dos congestionamentos.

1.1 OBJETIVOS

Este trabalho tem como objetivo principal aplicar a técnica de satisfação de restrições distribuídas como alternativa para solucionar o problema de sincronismo de semafóricos em uma malha viária.

1.2 ESTRUTURA DO TRABALHO

Após uma breve introdução apresentada neste capítulo, o próximo capítulo apresentará, na seção 2.1, algumas informações sobre o trânsito, suas vias, tipos de circulação e classificações. A seção 2.2 é destinada a apresentação dos conceitos sobre sinalização semafórica. Uma introdução sobre CSP será apresentada na seção 2.3 e em seguida (seção 2.4 e 2.5) será apresentada a teoria de DCSP e o algorítmo *Asynchronous Backtracking* (AB).

No capítulo 3 é descrito o desenvolvimento do trabalho, o que foi implementado e como se chegou aos objetivos propostos. Por fim o capítulo de conclusões apresenta os principais resultados e aponta possíveis extensões para o trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Nesse capítulo serão abordados assuntos relacionados com o foco do trabalho proposto, enfatizando aspectos utilizados para definição e implementação do trabalho. Serão abordados os seguintes assuntos: trânsito e suas características, sinalização semafórica, CSP, DCSP e o algoritmo AB.

2.1 TRÂNSITO

O Código de Trânsito Brasileiro (CTO) considera trânsito com sendo a utilização das vias por pessoas, veículos e animais, isolados ou em grupos, conduzidos ou não, para fins de circulação, parada, estacionamento e operação de carga ou descarga.

Conforme Silva (2001, p. 1), a tendência que se observa ultimamente é a de considerar trânsito numa definição abrangente, como o deslocamento em geral de pessoas e/ou veículos. Tráfego, por sua vez, embute a noção de via; refere-se ao deslocamento de pessoas, mercadorias ou veículos através de meios apropriados, com origens e destinos definidos, sujeito a algum tipo de ordenamento.

2.1.1 VIAS

Segundo Silva (2001, p. 10), uma via pode ser definida como um espaço para circulação. O conjunto estruturado de vias que servem a uma determinada região é conhecido como sistema viário e tem como funções básicas assegurar mobilidade e acessibilidade ao usuário.

As vias podem ser classificadas conforme sua circulação da seguinte forma:

a) circulação de sentido duplo: as vias com circulação natural, não requerendo sinalização estabelecendo sua circulação são denominadas Vias de Sentido Duplo, conhecidas também como vias de mão dupla. Estas vias têm como vantagens

básicas a melhor acessibilidade aos seus imóveis e melhor acesso aos serviços de ônibus, os quais podem circular em ambos os sentidos da mesma via, facilitando a utilização pelos usuários. As desvantagens são de riscos de colisões frontais, conflitos nas conversões à esquerda e maior dificuldade na travessia dos pedestres (SCHMITZ, 2002, p. 4);

b) circulação de sentido único: as Vias de Sentido Duplo podem ser transformadas em Vias de Sentido Único, ou conhecidas também como vias de mão única, sendo escolhido um sentido para sua operação e sinalizadas para esse fim. O motivo principal para essa transformação é a saturação da via como mão dupla, causando problemas de segurança e fluidez no tráfego. Tem como vantagens a eliminação de riscos de colisão frontal, maior segurança na travessia de pedestres, redução do número de movimentos conflitantes nos cruzamentos, aumento da capacidade viária proporcionando maior fluidez, e permite ótima progressão dos semáforos. As desvantagens principais são o aumento da velocidade, podendo aumentar os acidentes em determinados horários e menor acessibilidade (SCHMITZ, 2002, p. 4).

Outra classificação que pode-se atribuir as vias é com relação ao tipo de fluxo, que pode ser influenciado pelo tipo do cruzamento, em nível (interseções das vias) e em desnível (pontes e viadutos):

a) vias de fluxo ininterrupto: são aquelas concebidas para não haver restrições à passagem do fluxo de tráfego, havendo controle de acesso e com isso a quase inexistência de movimentos conflitantes. Os cruzamentos importantes são em desnível, e os cruzamentos em nível existentes geralmente não influenciam no escoamento do tráfego. São vias de fluxo ininterrupto as estradas e vias expressas (SCHMITZ, 2002, p. 5);

b) vias de fluxo interrompido: são as demais vias em que existem cruzamentos em nível e que há a necessidade de se controlar os fluxos que se cruzam através de dispositivos de controle, placas sinalizadoras para estabelecimento das vias

> preferenciais ou semáforos que alternam os momentos de direito de passagem das vias (SCHMITZ, 2002, p. 5).

É nas intersecções o conflito de interesses apresenta-se mais claramente. Por isso, é necessário utilizar instrumentos de controle para regulamentar o direito de passagem nessas interseções. Alguns desses instrumentos são: placa "Dê a preferência", placa "Pare" – parada obrigatória, mini- rotatória, canalização, rotatória, semáforo, intersecções em desnível.

2.2 SINALIZAÇÃO SEMÁFORICA

O semáforo é sinal de trânsito, que através de indicações luminosas altera o direito de passagem em interseções, separando os movimentos conflitantes. Pode-se dividir os semáforos em dois tipos:

a) semáforos veiculares: normalmente tem três focos, vermelho, amarelo e verde. Podendo-se substituir o comando do amarelo pelas luzes verdes e vermelhas acesas ao simultaneamente;
b) semáforos de pedestres: é constituído por três comandos, boneco verde fixo que indica possibilidade de travessia, boneco vermelho intermitente indicando que o tempo para travessia está acabando e boneco vermelho fixo indicando impossibilidade de travessia.

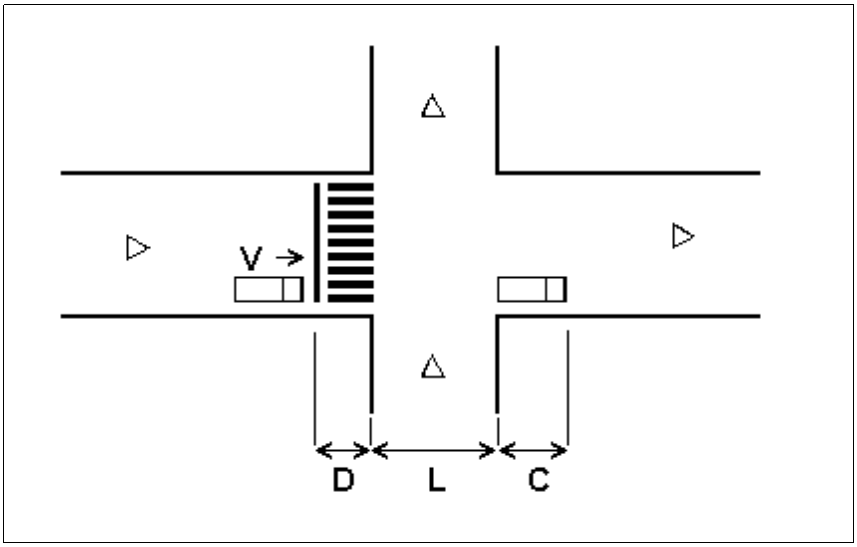
Para a sinalização veicular a seqüência de indicação de cores de um semáforo é verde, amarelo, vermelho e novamente verde. Esta seqüência aplicada a uma ou mais correntes de tráfego é denominado fase. O tempo total para a completa seqüência de sinalização, numa interseção, é denominado ciclo. Um dos vários períodos de tempo dentro do ciclo é denominado estágio (CARVALHO, 2000).

2.2.1 TEMPOS SEMAFÓRICOS

Segundo Espel (2000, p. 35) a programação dos tempos semafóricos consiste na obtenção dos valores dos tempos de ciclo para elaboração dos planos de tráfego necessários para a operação dos cruzamentos.

2.2.1.1 TEMPO DE VERDE MÍNIMO DE ESTÁGIO

É o tempo mínimo de verde que permite a saída de um veículo da retenção até a transposição do cruzamento com segurança (Figura 1) (SCHMITZ, 2002, p. 10).



Fonte: adaptado de Espel (2000, p. 51)

Figura 1 - Variáveis do tempo de verde mínimo de estágio

Sendo:

$$Tt = \sqrt{\frac{2 \cdot (D + L + C)}{a}} \quad \text{(Equação I)}$$

Onde:

Tt = tempo de travessia;

D = distância de retenção do cruzamento;

L = comprimento do cruzamento;

C = comprimento médio de um veículo;

a = aceleração do veículo.

Segundo pesquisado, para este tempo deve-se adicionar um tempo de percepção e reação do motorista (Tpr), sendo para este caso 1,5s.

2.2.1.2 TEMPO DE VERDE DE ESCOAMENTO

Durante a fase vermelha é esperado a formação de uma fila. Para que esta fila possa escoar deve ser calculado um tempo de escoamento, sendo este o tempo de verde de escoamento, este valor deve ser equacionado em função da quantidade de carros (SCHMITZ, 2002, p. 11).

Sendo:

$$T_{e1} = T_a + \left(N_1 - 1\right) \cdot t + \frac{C_v}{V_m} + \frac{(N_1 \cdot (C + 0{,}5)) + C_v}{V} \quad \text{(Equação II)}$$

Onde:

T_{e1} = tempo de escoamento procurado;

T_a = tempo de atraso do primeiro veículo imediatamente após a abertura do sinal;

N_1 = número de Veículos em fila;

t = intervalo de tempo entre o arranque de dois veículos sucessivos;

C_v = distância percorrida durante o período de embalagem;

V_m = velocidade média durante o período de embalagem;

C = comprimento médio de um veículo;

V = velocidade do regime.

Segundo Stern (1969), normalmente tem-se os valores de T_a = 3,0s, t = 1,5s, C_v = 60,00mt, C = 4,00mt e $V_m = \frac{V}{2}$.

2.2.1.3 TEMPO DE AMARELO

O tempo de amarelo é o tempo entre o final do verde e o início do vermelho, ele é necessário pois é impossível parar o veículo instantaneamente. Ao perceber a luz amarela o motorista deve parar o veículo, salvo se isto representar situação de perigo para o veículo que vem atrás.

O tempo de amarelo é calculado com base na taxa de percepção e reação, na velocidade da via e na aceleração do veículo.

Sendo:

$$Ta = Tpr + \frac{V}{2 \cdot a} \quad \text{(Equação III)}$$

Onde:

Ta = tempo de amarelo;

Tpr = tempo de percepção e reação;

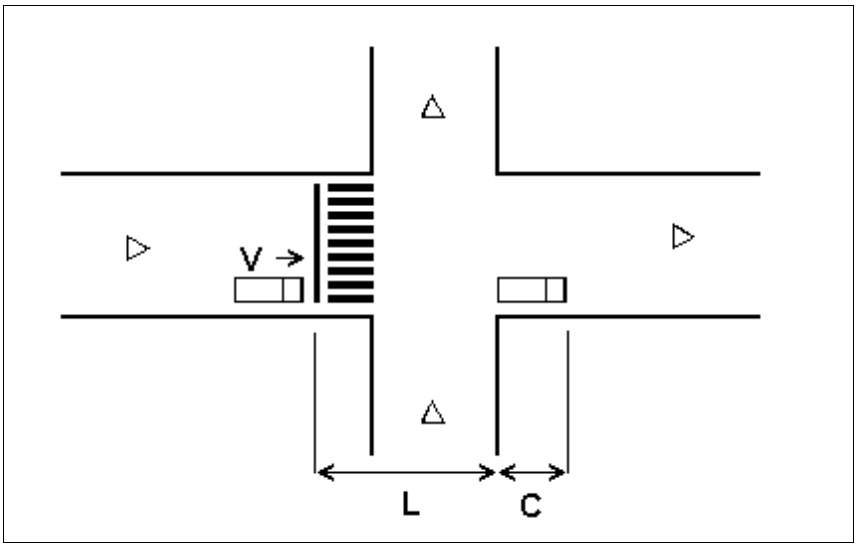
V = velocidade da via;

a = aceleração do veículo.

Conforme pesquisado, os valores encontrados para a desaceleração máxima (a) aceita pelos motoristas, variam entre 2,0m/s^2 e 4,2m/s^2, sendo recomendado a adoção do valor 2,8m/s^2. E para o tempo de percepção e reação os valores variam de 0,8 s a 1,2 s, sendo recomendado a adoção de 1,0 s. No caso do resultado da equação não ser um valor inteiro, deve-se sempre arredondá-lo para cima.

2.2.1.4 TEMPO DE VERMELHO DE SEGURANÇA

É o tempo necessário para que um veículo que iniciou a travessia no fim do tempo de amarelo consiga sair da área de conflito, conforme a Figura 2. Em geral é o tempo de vermelho de segurança é necessário em cruzamentos de grandes dimensões, onde é necessário um tempo maior para travessia.



Fonte: adaptado de Espel (2000, p. 50)

Figura 2 - Variáveis do tempo de vermelho de segurança

Sendo:

$$Tvs = \frac{(L+C)}{V} \quad \text{(Equação IV)}$$

Onde:

Tvs = Tempo de vermelho de Segurança;

L = Largura do cruzamento incluindo a faixa de pedestre anterior;

C = Comprimento do veículo;

V = Velocidade do veículo.

2.2.1.5 TEMPO DE DEFASAGEM

O tempo de defasagem é o intervalo de tempo entre o início e o final do tempo de verde de dois ou mais semáforos próximos. Segundo Espel (2000, p. 39), o tempo de defasagem visa que os veículos que partem do início do tempo de verde de um semáforo, após um determinado tempo de percurso até o próximo semáforo, consigam passar por este também no seu instante inicial de verde, havendo aproveitamento máximo dos tempos de verde de todo o sistema, objetivando minimizar ao máximo os inevitáveis atrasos e paradas que os semáforos ocasionam. Desta forma é criada a chamada "onda-verde".

2.3 CONSTRAINT SATISFACTION PROBLEM

Segundo Yokoo (2001, p. III), um problema de satisfação de restrições (do inglês *Constraint Satisfaction Problem* (CSP)) é um problema onde deseja-se encontrar um valor consistente para as variáveis envolvidas no problema. Um valor é consistente se não violar nenhuma restrição. Mesmo que a definição de CSP seja muito simples existe uma grande variedade de problemas de inteligência artificial que podem ser formalizados como CSPs.

Um CSP pode ser definido formalmente como (Tsang, 1993, p. 9):

a) um conjunto finito de variáveis $\{x_1, x_2, ..., x_n\}$, cujos valores $v_1, v_2, .., v_n$ pertencem a domínios $D_1, D_2, .., D_n$;
b) um conjunto finito (possivelmente vazio) de restrições $\{C_1, C_2, .., C_n\}$.

Um exemplo de CSP é o jogo das *N*-Rainhas. O objetivo deste jogo é colocar um número N ($N \geqslant 4$) de rainhas em um tabuleiro quadriculado NxN, de modo que só exista uma

rainha em cada linha e coluna; e que uma rainha não esteja na mesma diagonal que outra rainha (Figura 3). Em um exemplo com quatro rainhas tem-se o seguinte CSP:

a) o conjunto de variáveis $\{x_1, x_2, x_3, x_4\}$, cujo domínio são as colunas {1, 2, 3, 4};

b) o domínio de cada variável é o conjunto de posições que uma rainha pode ocupar na sua linha $D_1 = \{1,2,3,4\}$;

c) as restrições, por exemplo entre x_i e x_j, que podem ser representadas como $(x_i \neq x_j) \wedge (|\, i - j \,| \neq |\, x_i - x_j \,|)$ (YOKOO, 2001)[1];

d) um possível conjunto de solução é $\{x_1 = 2, x_2 = 4, x_3 = 1, x_4 = 3\}$.

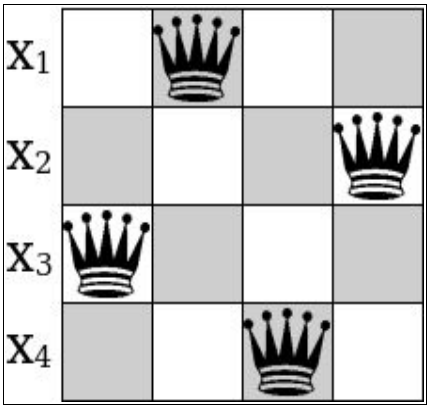


Figura 3 - Exemplo do jogo das n-rainhas

2.4 DISTRIBUTED CONSTRAINT SATISFACTION PROBLEM

Em um DCSP, as variáveis e restrições estão distribuídas entre agentes automatizados (YOKOO, 2001, p. 47). A diferença fundamental entre os algoritmos para resolução de CSP e DCSP é a manipulação do conhecimento. Enquanto em um algoritmo de CSP o conhecimento do problema (variáveis e restrições) é centralizado, em um algoritmo de DCSP o conhecimento está disperso entre diversos agentes, fazendo com que nenhum agente sozinho conheça todo o problema. O impacto direto é o aumento da dificuldade de ser ter uma visão global instantânea do problema, pois os agentes podem estar em máquinas diferentes, com tempos de respostas diferentes (TRALAMAZZA, 2004, p. 15).

[1] Neste predicado a primeira diferença da conjunção garante que duas rainhas não ocuparão a mesma coluna e a segunda diferença garante que duas rainhas não ocuparão a mesma diagonal.

A utilização de DCSP não está tão ligada ao ganho de desempenho na resolução dos problemas. Yokoo (2001, p. 49) cita exemplos de sistemas multi-agentes onde a centralização do conhecimento em um único agente seria inviável ou até mesmo inseguro. Segundo Tralamazza (2004, p. 15), este conhecimento distribuído proporciona novos desafios e áreas de estudo. Por exemplo, pode-se desejar que os algoritmos considerem tempos de resposta mínimos, quedas de agentes, problemas dinâmicos (restrições alteradas durante o processamento do algoritmo) e segurança ou privacidade dos dados trafegados.

2.5 ASYNCHRONOUS BACKTRACKING

Asynchonous *Backtracking* (AB) foi o primeiro algoritmo proposto por Yokoo e sua equipe capaz de resolver um DCSP. No algoritmo AB os agentes executam concorrentemente, de forma assíncrona e sem controle global (YOKOO, 2001, p.58). Segundo Tralamazza (2004, p. 15), cada agente possui as seguintes características:

a) um identificador único;
b) um conjunto de agentes diretamente conectados, chamados vizinhos;
c) para efeito de simplificação, exatamente uma variável x_i com um valor v_i pertencente a um domínio finito D_i;
d) um conjunto de restrições que afetam a variável x_i.

Para comunicação entre os agentes, o AB utiliza mensagens. As mensagens são enviadas a todos os agentes conectados por restrições. Estes agentes relacionados são chamados vizinhos. Os dois tipos de mensagens são:

a) *ok?* : enviada por um agente que escolheu um valor a um agente que avalia a restrição para saber se o valor escolhido é aceitável;
b) *nogood*: enviada por um agente que avaliou uma restrição para um agente que escolheu um valor, indicando que houve uma violação de restrição.

Segundo Tralamazza (2004, p. 16), cada agente possui um conjunto de agentes a qual está conectado, chamado de vizinhos, que é separado em dois outros conjuntos. O conjunto

Incoming contendo os agentes lhe enviarão mensagens *ok?* e *Outgoing* contendo os agentes que lhe enviam mensagens *nogood.* Esta separação é dada pela direção da aresta, como ilustra a Figura 4 e a Tabela 1.

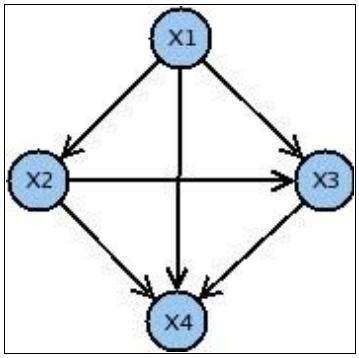


Figura 4 – Ordenação das variáveis do problema da 4 rainhas

Tabela 1 – Vizinho no problema da 4 rainhas

Agente	***Outgoing***	***Incoming***	**Domínio**
x_1	x_2, x_3, x_4	-	1, 2, 3, 4
x_2	x_3, x_4	x_1	1, 2, 3, 4
x_3	x_4	x_1, x_2	1, 2, 3, 4
x_4	-	x_1, x_2, x_3	1, 2, 3, 4

Os agente possuem ainda um mapeamento de variáveis para valores chamado *agent_view*, este mapeamento representa a visão que o agente possui dos valores dos seus vizinhos. O *agent_view* nada mais é do que uma tabela de pares variável/valor (TRALAMAZZA, 2004, p. 16).

O algoritmo inicia com os agentes escolhendo um valor para sua variável, enviando (*send_ok()*) este valor para os agentes que estão na sua lista *outgoing* e aguarda um retorno (Quadro 1). Caso a mensagem recebida seja um *ok* a rotina RECEIVED_OK, apresentada no quadro 1, será executada, adicionando o par variável-valor recebido ao seu *agent-view* e excecutando a rotina CHECK_AGENT_VIEW. Na rotina CHECK_AGENT_VIEW é verificada a consitência do valor do próprio agente com o seu *agent_view*, este valor será consistente se não violar nenhuma restrição do agente levando em conta os valores contidos no *agent_view* e o seu próprio valor, e se o seu *agent_view* mais o seu próprio valor não

estiver no conjunto de *nogoods* recebidos. Caso seu valor não seja consistente o agente irá procurar um valor que o torne consistente.

Quando o algoritmo executa um *backtrack* ele está sinalizando aos demais agentes que não conseguiu se tornar consistente. Para isto ele gera um *nogood* que no caso é o próprio *agent_view*. Este *nogood* gerado é então enviado ao agente de menor prioridade diretamente conectado. Cada *nogood* recebido é armazenado em uma lista chamada *nogoodlist* sendo utilizada na checagem de consistência do *agent_view*. Cada item desta lista (*nogoodlist*) pode ser visto como uma nova restrição, pois a rotina CHECK_AGENT_VIEW verificará se o *agent_view* é um *nogood* antigo conhecido. Armazenar tentativas passadas garante que elas não se repitam no futuro, porém, para certos domínios isto não pode ser aplicado, pois caso o domínio seja extenso/infinito, por exemplo o domínio dos número reais, o número de *nogoods* possíveis tornaria o processo inviável ou até impossível (TRALAMAZZA, 2004, p. 17).

Yokoo (2001, p. 64) garante que o algoritmo ABT é completo, pois sempre será achada uma solução, caso ela exista. Entretando o algoritmo não detecta a parada por sucesso, pois após acharem uma solução os agentes continuam esperando uma mensagem para continuarem a computação. No caso de não existir solução para o problema o algoritmo irá parar. Portanto, para que seja detectada a parada em caso de sucesso deve-se implementar um mecanismo para detecção de parada.

```
RECEIVED_OK(xj, dj)
       add(xj, dj) to agent_view;
       check_agent_view();

RECEIVED_NOGOOD(xj, nogood)
       add(nogood) to nogoodlist;
       if xk is not connected is contained in nogood then
              REQUEST_ADDLINK(xk, xj);
              add(xk, dk) to agent_view;
       old_value = current_value;
       CHECK_AGENT_VIEW();
       if old_value = current_value then
              SEND_OK(xj, current_value, xj);

CHECK_AGENT_VIEW()
       if agent_view is not consistent with current_value then
              if no value in Di is consistent with agent_view then
                     BACKTRACK();
              else
                     current_value = d;
                     SEND_OK(xi, d) to outgoing links;

BACKTRACK()
       nogoods = {V|V = inconsistent subset of agent_view};
       if nogood = {} then
              BROADCAST_NOSOLUTION();
              return;
       for each V in nogood do
              select(xj, dj) where xj has the lowers priority;
              SEND_NOGOOD(nogood, xi, V) to xj;
              remove(xj, dj) from agent_view;
       CHECK_AGENT_VIEW();
```

Quadro 1 – Algoritmo ABT adaptado de Yokoo (2001)

2.6 TRABALHOS CORRELATOS

Como trabalhos correlatos na área de trânsito pode-se citar: Sistema de Informação e Controle para Mobilidade Urbana, chamado SincMobil (SINCMOBIL, 2003) e Sistema de Controle de Tráfego Urbano utilizando Sistemas Multi-Agentes (SCHIMITZ, 2002). Na área de DCSP verificou-se a existência do seguinte trabalho: Desenvolvimento de um Algoritmo para Problema de Satisfacão de Restrição Distribuída (TRALAMAZZA, 2004).

2.6.1 SINCMOBIL

SincMobil é um projeto da Universidade Federal de Santa Catarina, que visa disponibilizar informações em tempo real sobre o tráfego urbano (Sistema de Informação) e o controle em tempo real dos semáforos para garantir desempenho ótimo da malha viária (Sistema de Controle).

Ao buscar mobilidade urbana, os cidadãos brasileiros dispõem de pouca informação: usuários do transporte coletivo tem dificuldade em acessar quadros de horários e itinerários, inexistindo terminais de consulta com estimativa de chegada dos ônibus; operadores de fretes não podem aperfeiçoar suas logísticas por falta de informações sobre atrasos em rotas; motoristas não podem escolher trajetos por não contarem com informação em tempo real.

Pelo lado do controle do tráfego urbano, tem-se frequentemente semáforos mal ajustados, operando com planos desatualizados e sem sincronismo. Os avanços da última década foram alcançados com sistemas importados (São Paulo, Fortaleza), cujo alto custo impede a adoção em larga escala dos benefícios da otimização em tempo real do tráfego com base em monitoração das vias. Além disso, predomina no setor a inexistência de padrões de comunicação entre equipamentos, criando um domínio territorial dos fornecedores.

Objetiva-se ajudar a mudar esta realidade, através da concepção e implementação de dois sistemas que, interagindo entre si, agilizem a mobilidade urbana:

a) um Sistema de Informação capaz de interligar subsistemas distintos como centrais de controle de tráfego, sistemas de consulta via Internet, e centrais de gerência de estacionamentos, provendo informação em tempo real aos usuários;
b) um Sistema de Controle com controle ótimo dos semáforos que reduziria o consumo de combustível e os atrasos de viagens em vias urbanas .

Ambos baseados em arquiteturas de software padronizadas. Para dar suporte à interação entre tais sistemas, prevê-se a constituição do sistema de comunicação usando protocolos abertos (SINCMOBIL, 2003).

2.6.2 SISTEMA DE CONTROLE DE TRÁFEGO URBANO UTILIZANDO SISTEMAS MULTI-AGENTES

O trabalho desenvolvido por Schmitz (2002) utiliza a técnica de Sistemas Multi-Agentes para controle de interseções semafóricas. O sistema consiste em um conjunto de agentes-semáforos que negociam entre si para conseguir o direito de passagem dos veículos

nos cruzamentos. O sistema não se limita apenas a sincronismo de semáforos, mas sim em análise e decisão em tempo real sobre as ações do mundo. Neste aspecto, procura-se diminuir o tempo em que os motoristas ficam esperando em cruzamentos e a ocupação (saturação) da capacidade das vias (SCHIMITZ, 2002, p. 58).

2.6.3 DESENVOLVIMENTO DE UM ALGORITMO PARA PROBLEMA DE SATISFACÃO DE RESTRIÇÃO DISTRIBUÍDA

Neste trabalho Tralamazza (2004) especificou, implementou e analisou empiricamente os principais algoritmos para resolução de DCSP propostos por Makoto. Também foram propostas alterações no algoritmo original AWC para melhorar o seu desempenho:

a) adaptação da heurística do valor menos utilizado;
b) ordenação das restrições
c) não armazenamento de *nogoods* recebidos pelos agentes.

Tralamazza (2004, p. 36) destaca que o protótipo alcançou bons resultados e evidenciou que a busca pela completude do algoritmo AWC. Guardando os *nogoods* recebidos para isto, causa grande perda de desempenho. Foi notado que a escolha de valores possui papel importante na execução, podendo diminuir o número de tentativas, com isto aumentar o desempenho e diminuir o grau de exposição do problema.

3 DESENVOLVIMENTO DO SISTEMA

Neste capítulo será descrito o desenvolvimento do trabalho proposto, onde as seções abaixo descrevem os requisitos do sistema, sua especificação, implementação, os resultados e discuções envolvidas no desenvolvimento deste trabalho.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Abaixo são descritos os requisitos funcionais (RF) e os requisitos não funcionais (RNF) propostos para este trabalho:

a) implementação de otimização de DCSP (RF);
b) implementação de restrições dinâmicas (RF);
c) implementação de variáveis locais (RF);
d) modelagem de um DSCP para resolução do problema de sincronismo de semáforos viários (RF);
e) utilização da plataforma Java para construção do simulador (RNF);
f) permitir a criação de malhas viárias que serão utilizadas no simulador (RF);
g) utilização de XML para entrada de dados no simulador (RNF);
h) obtenção de resultados rápidos para estados dos semáforos (RF);

3.2 VISÃO GERAL DA IMPLEMENTAÇÃO

O protótipo implementado neste trabalho foi desenvolvido utilizando o *framework* DynDCSP (DYNDCSP, 2003), desenvolvido pelo grupo de pesquisa em Inteligência Artificial da Fundação Universidade Regional de Blumenau (FURB). O *framework* foi desenvolvido utilizando a linguagem Java, por este motivo utilizou-se a mesma linguagem para desenvolvimento do protótipo. Para execução do protótipo é necessário que a ferramenta Apache Ant, versão 1.6.0 ou superior, esteja instalada.

O *framework* DynDCSP implementa três algoritmos para resolução de DCSP: *Asynchronous Backtracking*, *Asynchronous Weak-Commitment* (AWC) e *Distributed Breakout* (DB), que estão no pacote *dcsp.alg*, conforme apresentado na Figura 5.

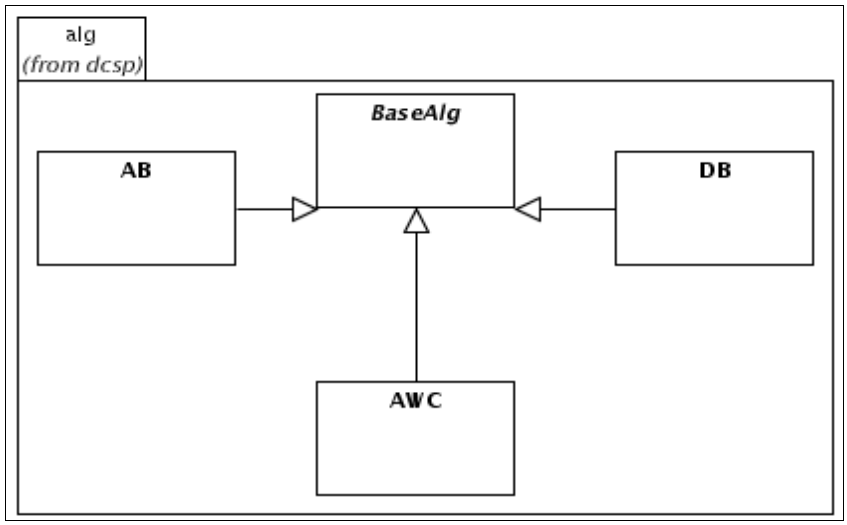


Figura 5 – Pacote *dcsp.alg* original

Para desenvolver o protótipo foi usado o algoritmo *AB*. Como as implementações para inclusão de Variáveis Locais e Retrições Dinâmicas necessitam de uma especialização da classe BaseAlg, criou-se uma subclasse de *BaseAlg* chamada *MBaseAlg* e passou-se a derivar a classe *AB* de *MBaseAlg*. A figura 6 apresenta as alterações no pacote *dcsp.alg*.

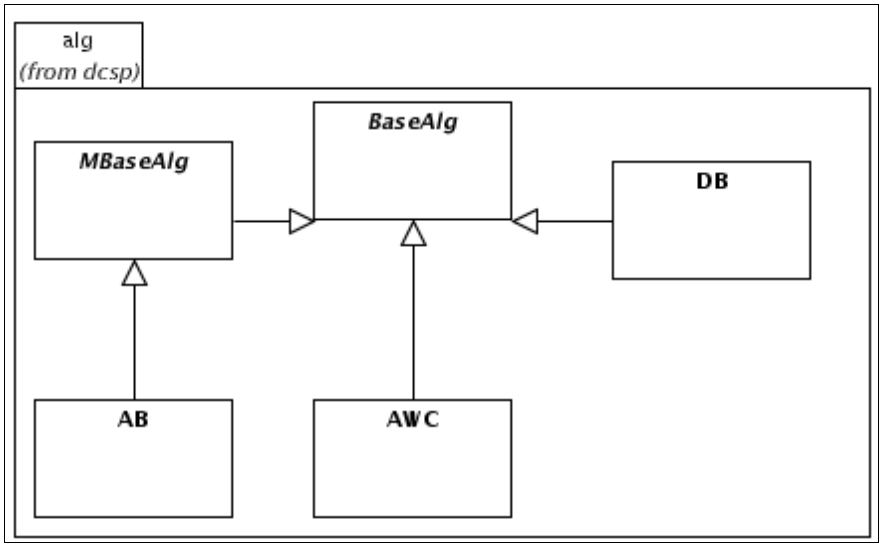


Figura 6 - Modificação no pacote *dcsp.alg* modificado

Nas próximas seções serão detalhadas as implementações no *framework* para a inclusão das novas funcionalidades necessárias para o desenvolvimento do protótipo.

3.3 VARIÁVEIS LOCAIS

As variáveis locais são variáveis que ficam restritas ao agente, pois não são do interesse do CSP global. No caso dos semáforos o número de carros esperando em um semáforo pode ser visto como uma variável local. Essas variáveis também não possuem domínio, em geral seus valores são alterados por um agente externo, no caso deste protótipo este agente é o simulador.

A título de ilustração, a Figura 7 representa um CSP onde existem cinco variáveis que são globais para o CSP (X1, X2, X3, X4, X5), neste caso cada agente fica responsável por uma variável global. No exemplo o Agente 1 tem a variável local Y1, que não é acessível a nenhum outro agente.

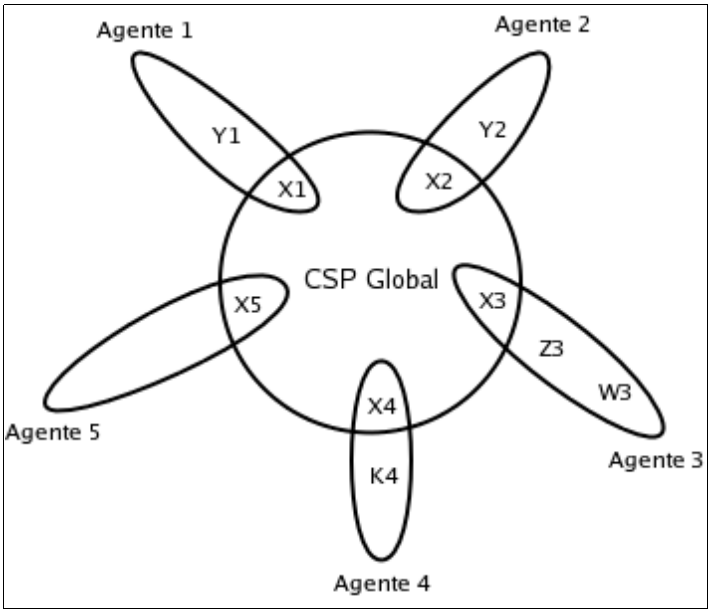


Figura 7 – Representação das variáveis locais e globais do CSP

As variáveis locais não são declaradas no arquivo de definição do CSP como acontece com as variáveis do CSP. Para incluí-las, excluí-las e alterar seu valor são definidas mensagens na classe *MBaseAlg*. A Figura 8 apresenta o detalhamento da classe *MBaseAlg*.

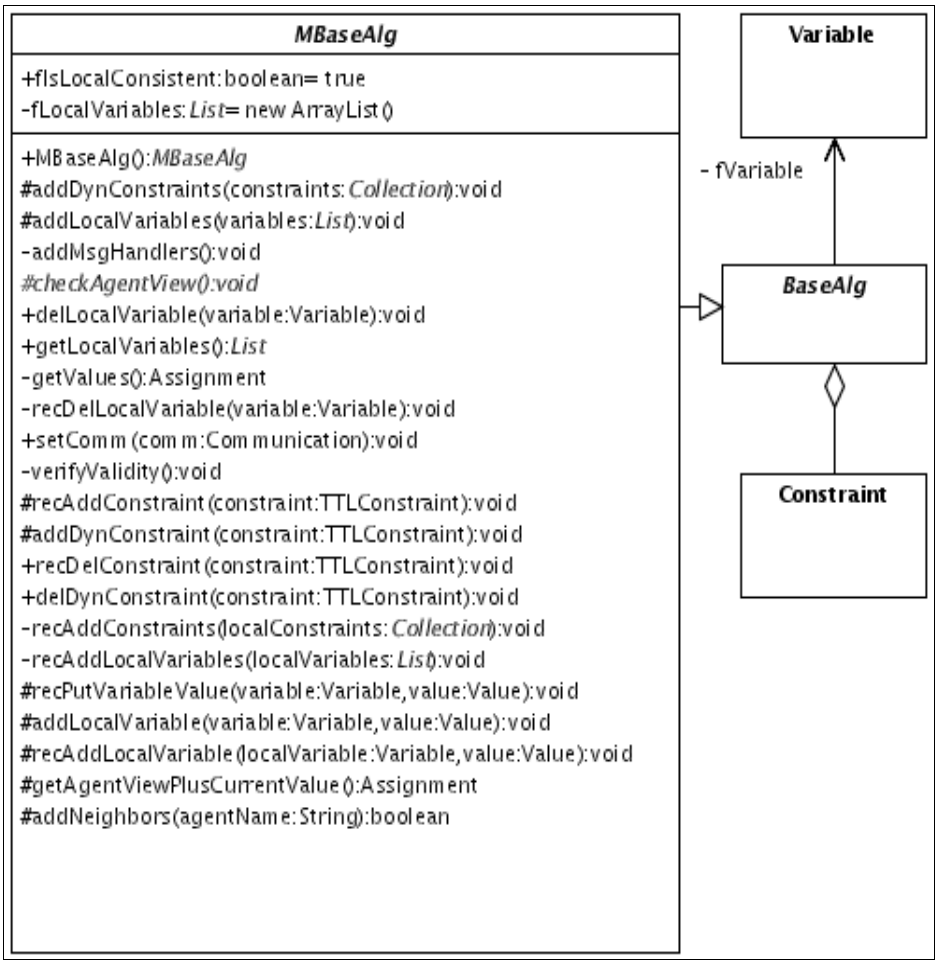


Figura 8 - Classe *MBaseAlg*

As mensagens para manipulação das variáveis locais são as seguintes:

a) *addLocalVariables*: mensagem usada para incluir uma lista de variáveis locais. Deve conter uma lista com as variáveis locais chamada *localVariables;*

b) *addLocalVariable*: mensagem usada para incluir uma variável local. Deve conter uma variável chamada *localVariable*;

c) *putVariableValue*: mensagem que altera valor de uma variável. Deve conter a variável, chamada *localVariable,* a qual o valor será adicionado e o valor da variável, chamado *value*;

d) *delLocalVariable*: mensagem que exclui uma variável local. Deve conter a variável, chamada *localVariable*, que será excluída.

3.4 RESTRIÇÕES DINÂMICAS

Restrições dinâmicas são restrições que não estão declaradas no arquivo de definição do CSP e são adicionadas ao CSP durante o funcionamento do sistema. As restrições dinâmicas são do tipo *TTLConstraint*, esta classe é derivada da classe *Constraint*, implementada pelo *framework,* conforme mostra a Figura 9. A diferença entre os dois tipos de restrições é que *TTLConstraint* pode ter um tempo de vida, isto é, a restrição será válida por determinado número de segundos. Após este tempo a restrição não é mais avaliada.

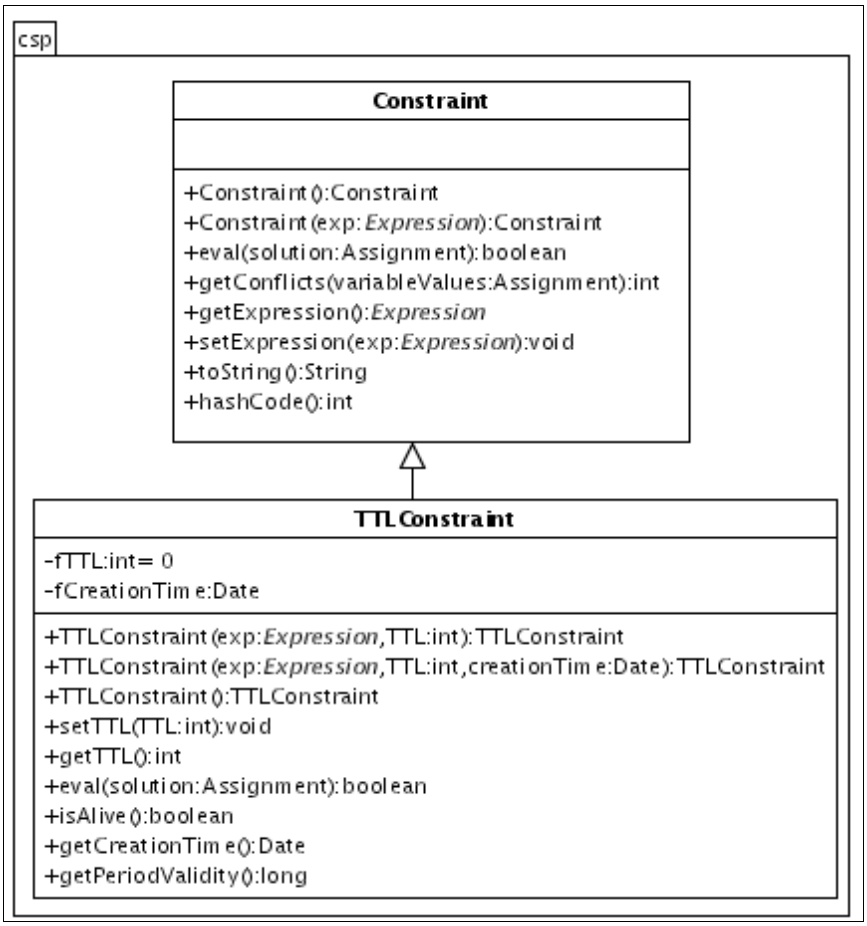


Figura 9 – Classe *TTLConstraint*

A classe *TTLConstraint* tem uma propriedade privada chamada *fTTL*, que guarda o número de segundos que esta restrição tem validade. Este tempo começa a ser contado a partir da instanciação do objeto que representa a restrição. O método *getPeriodValidity* retorna o número de segundos que a restrição ainda tem validade, caso este número seja negativo a restrição já espirou e deve ser removida. O método *isAlive* retorna verdadeiro caso a restrição

ainda seja válida e falso caso contrário. No método *eval* é verificada a validade da restrição, se ela for válida a análise prossegue no método *eval* da classe *Constraint,* caso contrário esta restrição não sofre análise e é avaliada como consistente, retornando *true*. A implementação do método *eval* da classe *TTLConstraint* pode ser vista no Quadro 2.

```
public boolean eval(Assignment solution) throws VariableNotFoundException  {
	// Restrição tem validade então precisa ser avaliada
	if (isAlive()) {
		return super.eval(solution);
	} else {
		return true;
	}
}
```

Quadro 2 – Método *eval* da classe *TTL Constraint*

A avaliação das restrições será feita no método *isConsistent* da classe *BaseAlg*, conforme Quadro 3. Este método é responsável por definir se a solução encontrada pelo agente é consistente com suas restrições, para isso são avaliadas todas as restrições com variáveis de maior prioridade que a variável do próprio agente. Caso uma das restrições esteja inconsistente o agente é dito inconsistente.

```
public boolean isConsistent(Assignment solution) {
	// Retorna um subconjunto contendo as variáveis de maior prioridade
	Map vars = solution.getHigherPriorityVariables(getVariable());
	if (vars.isEmpty())
		return true;
	// Percorre todas as restrições do agente
	Iterator ic = getConstraints().iterator();
	boolean found;
	Iterator icv;
	while (ic.hasNext()) {
		Constraint c = (Constraint) ic.next();
		// Percorre todas as variáveis da restrição até encontrar uma
		// contida em vars
		icv = c.getExpression().getVariables().iterator();
		found = false;
		// Deve encontrar uma variável contida em vars para avaliar
		// a restrição
		while (icv.hasNext() && (!found))
			found = vars.containsKey(icv.next());
		try {
			if (found && (!c.eval(solution)))
				return false;
		} catch (VariableNotFoundException e) {
		}
	}
	return isSolutionIncompatible(solution);
}
```

Quadro 3 – Método *isConsistent* da classe *BaseAlg*

Como comentado anteriormente, as restrições dinâmicas não são declaradas no arquivo de definição do CSP, portanto é necessário definir mensagens para que essas restrições possam ser manipuladas. A classe que implementa o tratamento das mensagens com pedidos de inclusão e exclusão das restrições dinâmicas é a classe *MBaseAlg.* Essas mensagens são:

a) *addDynConstraint*: mensagem que adiciona uma nova restrição ao CSP, deve conter a restrição, chamada *dynConstraint*;
b) *addDynConstraints*: mensagem que adiciona uma lista de restrições ao CSP, deve conter uma lista, chamada *dynConstraints;*
c) *delDynConstraint*: mensagem que exclui um restrição do CSP, deve conter a restrição, chamada *dynConstraint*.

Ao receber a mensagem *addDynConstraint* o método *recAddDynConstraint* é executado. Como a nova restrição pode tornar o agente inconsistente o método *recAddDynConstraint* além de adicionar a restrição ao CSP também define que o agente não pode mais ser considerado consistente e chama o método *checkAgentView.*

O método *recDelDynConstraint* tem o mesmo comportamento de *recAddDynConstraint* após a restrição ser excluída o estado do agente é definido como inconsistente e o método *checkAgentView* é executado.

As restrições dinâmicas que foram adicionadas ao DCSP serão removidas pelo método *verifyValidity* definida na classe *MBaseAlg,* onde é verificada a validade de cada restrição do tipo *TTLConstraint* através do método *isAlive*, como pode ser observado no quadro 4. O método *verifyValidity* é executado a cada 20 segundos por uma *thread* criada no método *setComm*, como pode ser visto no quadro 5.

```
private void verifyValidity() {
	List constraints = getConstraints();
	Iterator i = constraints.iterator();
	while (i.hasNext()) {
		try {
			TTLConstraint c = ( TTLConstraint ) i.next();
			if (c instanceof TTLConstraint) {
				if (!c.isAlive()) {
					delDynConstraint(c);
				}
			}
		} catch (ClassCastException e) {}
	}
}
```

Quadro 4 – Implementação do método *verifyValidity*

```
public void setComm(Communication comm) {
	super.setComm(comm);
	addMsgHandlers();
	// Cria thread que verifica se pode excluir restrição
	// que não tem mais validade
	Thread t = new Thread() {
		public void run() {
			while (true) {
				verifyValidity();
				try {
					sleep(20000);
				 } catch (InterruptedException e) {}
			}
		}
	};
	t.start();
}
```

Quadro 5 – *Thread* criada para limpar restrições dinâmicas que não tem mais validade

3.5 OTIMIZAÇÃO

Como descrito na seção 2.5, o algoritmo ABT não detecta o fim da execução caso um solução seja encontrada. Por isso o *framework* implementa um algoritmo para detecção de parada chamado *circulating token.* Este algoritmo estabelece uma lista circular entre os agentes que compõem o problema, cada agente possue uma *flag* que indica seu estado de consistência. Ao iniciar a execução é enviado um *token* ao primeiro agente da lista contendo informações sobre a parada, quando o *token* completar uma volta na lista com todos os agentes consistentes a execução é encerrada. Sendo assim, a execução é encerrada quando a primeira solução for encontrada.

Para implementar o protótipo foi necessário encontrar todas as soluções possíveis para o problema. Pois isso foram necessárias alterações no *framework* para que a execução não fosse encerrada ao encontrar a primeira solução e que as soluções encontradas não fossem reencontradas.

A classe do *framework* responsável por coordenar a execução do DCSP é *SaciManager*, esta classe é derivada da classe abstrata *BaseManager* (figura 10). Com essa classe é criado um agente e este dá inicio a criação de todos os outros agentes do DCSP. Quando uma possível solução é encontrada, o agente *SaciManager* recebe a mensagem *partial-end*, esta mensagem contem uma possível solução para o DCSP, esta solução é verificada e caso a solução seja válida a resolução do DCSP é encerrada e os agentes são finalizados pelo método *finished.*

Para se obter todas as soluções foi criada uma classe abstrata chamada *MBaseManager.* Esta classe é derivada de *BaseManager* e sua principal diferença está no método *verifyEnd,* que ao contrário do método da classe pai não encerra a resolução do DCSP quando uma solução é encontrada e ainda armazena as soluções em uma lista chamada *fAllSolution*. A Figura 10 apresenta o diagrama de classe para *MBaseManager.*

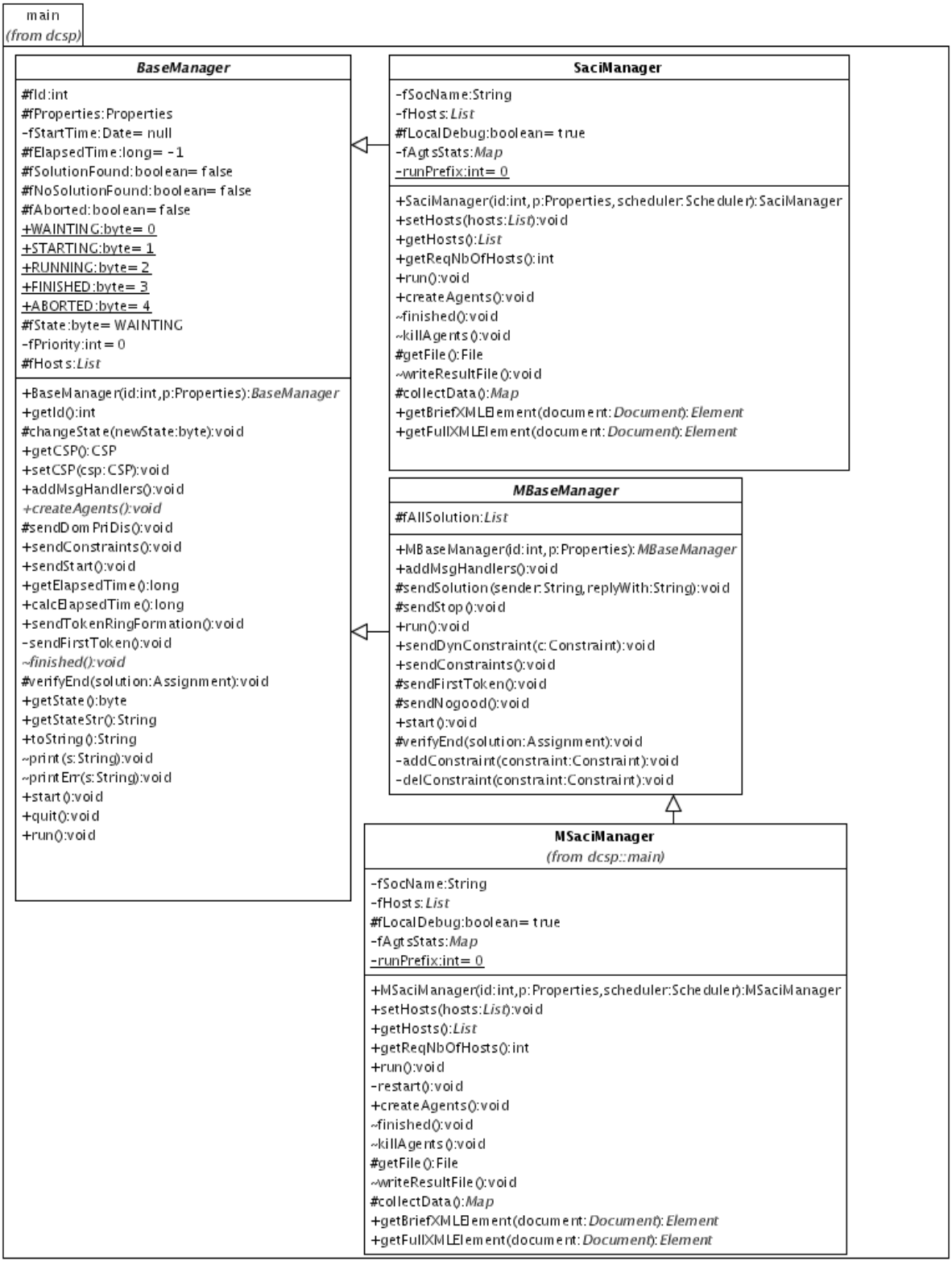


Figura 10 – Classes *MBaseManager*, *MBaseManager* e *MSaciManager*

A classe que cria o agente gerenciador do DCSP foi substituída pela classe *MSaciManager* (figura 10), que é derivada da classe *MBaseManager*, pois o agente da classe *SaciManager* executa somente enquanto não for encontrada uma solução. No caso do agente

da classe *MSaciManager* o agente executa até que todas as soluções sejam encontradas. O quadro 6 apresentando método *run* da classe *MSaciManager*.

```
public void run() {
	fComm = new SaciCommunication();
	print("Entering in society " + fSocName);
	if (((SaciCommunication) fComm).enterSoc("manager", fSocName)) {
		while (!fNoSolutionFound) {
			// Verifica se ? a primeira vez que est? executando
			switch (fState) {
				case STARTING:  // É a primeira vez
					super.run();
					break;
				case RUNNING:   // Já executou a primeira vez
					restart();
					break;
			}
			// Espera a computação terminar, verificando se achou
			// solução ou se acho que não existe solução.
			while (fSolutionFound || fNoSolutionFound) {
				try { Thread.sleep(100); }
				catch (InterruptedException e) {e.printStackTrace
();}
			}
			fSolutionFound = false;
		}
		changeState(FINISHED);
		finished();
	} else {
		// enter this agent in the DCSP society
		printErr("Could not enter into a saci society named " +
fSocName);
	}
```

Quadro 6 – Método *run* da classe *MSaciManager*

A primeira vez que o agente executa o método *run* é feita uma chamada ao método *run* da classe pai, com isso os agentes do DCSP são criados, seus domínios e suas restrições são enviados e o algoritmo de detecção de parada é inicializado, e a computação será inicializada. O agente *MSaciManager* aguarda a finalização da computação. Se foi encontrada uma solução o agente continuará executando o método *run.* Como o DCSP já foi inicializado o método executado será o *restart*. O método *restart* (Quadro 7) envia a solução encontrada anteriormente para os agentes na forma de *nogoods*, isto garante que uma mesma solução não seja encontrada mais de uma vez, levando a encontra todas as soluções para o CSP. Após o envio dos *nogoods* o algoritmo de detecção de parada e a computação são reinicializados.

```
private void restart() {
	sendNogood();
	sendStart();
	sendFirstToken();
}
```

Quadro 7 – Método *restart* da classe *MSaciManager*

3.6 DEFINIÇÃO DO CSP PARA O SEMÁFOROS

Para que um problema seja executado no *framework* devem ser criadas duas classes. A primeira classe implementa a interface *CSPFactory*, ela é responsável por instanciar o objeto do CSP propriamente dito. A segunda classe é derivada de CSP, nesta classe são definidas as variáveis do CSP, seus domínios e suas restrições.

O protótipo implementa a interface *CSPFactory* na classe *CruzamentoCSPFactory* e *CruzamentoCSP*, que é a classe derivada de CSP como pode ser observado no diagrama da figura 11.

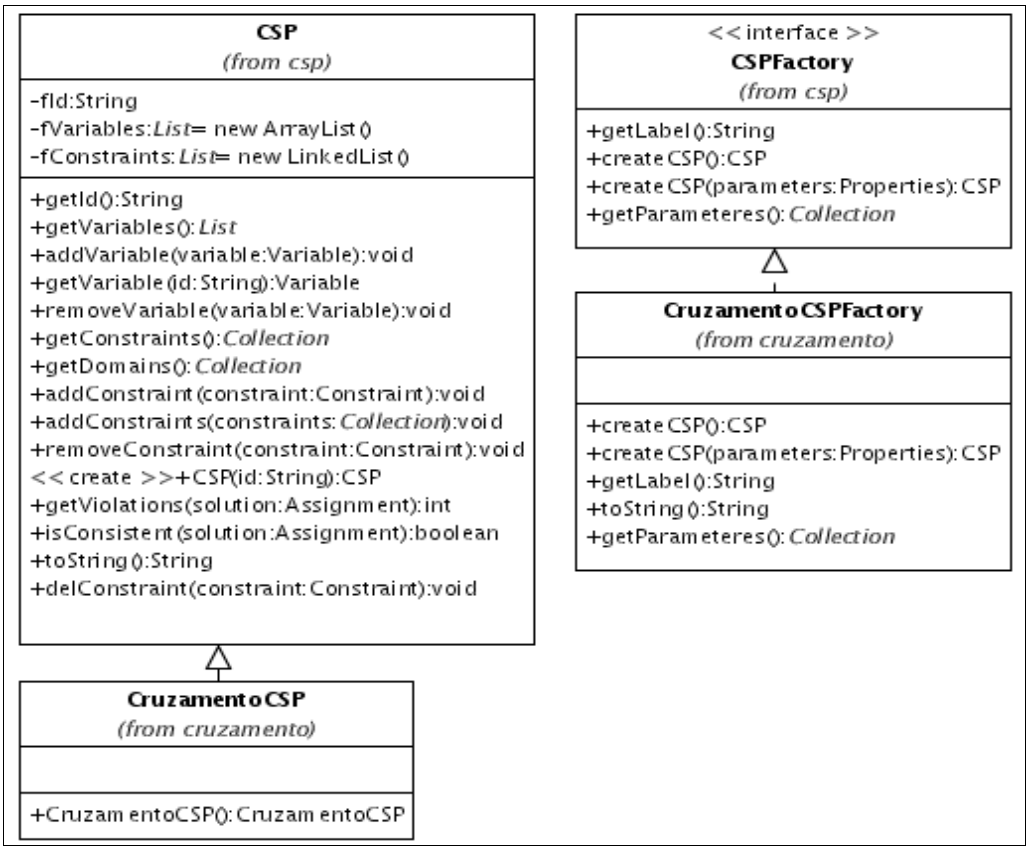


Figura 11 – Pacote cruzamento

O *CruzamentoCSP* cria 16 variáveis representado os semáforos (4 cruzamentos com 4 semáforos em cada cruzamento), a cada variável é atribuído um domínio inteiro de 1 até 2 (1 representa verde e 2 representa vermelho). Posteriormente são criadas 4 listas (uma para cada cruzamento), cada uma com as 4 variáveis que representam os semáforos do cruzamento. Então são criados duas restrições para cada cruzamento, uma especificando que pelo menos

uma variável do cruzamento **deve** ser igual a verde e a outra especificando que somente uma variável do cruzamento **pode** ser igual a verde. O quadro 8 apresenta a uma parte da implementação do construtor da classe *CruzamentoCSP* onde são declaradas as variáveis, seus domínios e as restrições para o cruzamento 1, representado graficamente na figura 12. Assim, o CSP dos cruzamentos pode ser definido como:

a) o conjunto de variáveis {c1s1, c1s2, c1s3, c1s4, c2s5, c2s6, c2s7, c2s8, c3s9, c3s10, c3s11, c3s12, c4s13, c4s14, c4s15, c4s16};

b) o domínio, que são as cores verde (representado pelo número 1) e vermelho (representado pelo número 2) {1, 2};

c) o conjunto das restrições {atleast([c1s1, c1s2, c1s3, c1s4], 1, verde), atmost[c1s1, c1s2, c1s3, c1s4], 1, verde), atleast([c2s5, c2s6, c2s7, c2s8], 1, verde), atmost[c2s5, c2s6, c2s7, c2s8], 1, verde), atleast([c3s9, c3s10, c3s11, c3s12], 1, verde), atmost [c3s9, c3s10, c3s11, c3s12], 1, verde), atleast([c4s13, c4s14, c4s15, c4s16], 1, verde), atmost[c4s13, c4s14, c4s15, c4s16], 1, verde),}[2].

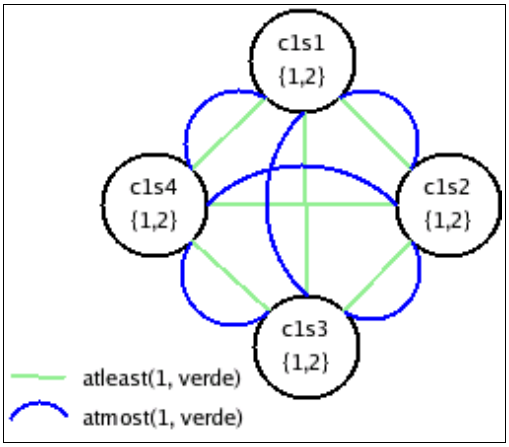


Figura 12 – Representação gráfica do CSP dos cruzamentos

[2] A função *atleast* garante que pelo menos um elemento da lista passada no primeiro parâmetro receberá o valor especificado no segundo parâmetro. Já a função *atmost* garante que somente um elemento da lista passada no primeiro parâmetro receberá o valor especificado no segundo parâmetro.

```
      Integer verde    = new Integer(1);
      Integer vermelho = new Integer(2);

      // creating domains
      Domain semaphoresDomain = new IntegerDomain("semaphoresDomain",
                                verde.intValue(), vermelho.intValue());

      // Cria 4 cruzamentos.
      for (int i = 0; i < 4; i++) {
            for (int j = 0; j < 4; j++) {
                  Variable av = new Variable("c" + (i + 1) + "s" + ((i * 4) +
                                             j + 1));
                  av.setDomain(semaphoresDomain);
                  addVariable(av);
            }
      }

      // Cria restrições para todos os semáforos do cruzamento 1
      ArrayList valuesC1 = new ArrayList(0);
      for (int i = 0; i < 4; i++) {
            valuesC1.add(new VariableExpression((Variable) getVariable("c1s"
+
                                                                  (i + 1))));
      }

      Expression atlC1 = new AtLeastExpression(valuesC1, 1, verde);
      Expression amC1 = new  AtMostExpression(valuesC1, 1, verde);

      addConstraint(new Constraint(atlC1));
      addConstraint(new Constraint(amC1));
```

Quadro 8 – Parte do construtor da classe *CruzamentoCSP*

A Figura 13 apresenta graficamente o ambiente com os cruzamentos:

Figura 13 – Representação gráfica do ambiente

3.7 SIMULADOR

O simulador foi utilizado para testar a implementação do CSP e das funcionalidades implementadas no *framework*. O simulador comunica-se com o *framework* através de mensagens, pelas quais são recebidos os resultados das execuções do DCSP e enviadas restrições para os agentes do DCSP.

O simulador é constituído das classes *MSimulator* e *MSemaphore*, armazenadas no pacote *sim*, como demonstra a Figura 14.

Figura 14 – Classes *MSimulator* e *MSemaphore*

A classe *MSimulator* é derivada da classe *SaciCommunication*, pois o simulador será um agente e trocará mensagens como *framework*. *MSimulator* contem uma lista com os semáforos que fazem parte da simulação.

A classe *MSemaphore* representa o semáforo real. Ela encapsula propriedades referentes ao semáforo como seu identificador, seus vizinhos, o próximo semáforo na onda verde, o estado atual do semáforo e características físicas.

A comunicação entre o *framework* e o simulador acontece por meio de mensagens. As mensagens utilizadas são:

a) *getSolution*: enviada do simulador para o gerenciador do DCSP pedindo a lista com as soluções encontradas. A lista é enviado no parâmetro *solutionlist*;
b) *sendStop*: enviada do simulador para o gerenciador do DCSP informando que deve finalizar a execução;
c) *sendStart*: enviada do simulador para o gerenciador do DCSP informando que deve iniciar a execução;
d) *addDynContraint*: enviada do simulador para o gerenciador do DCSP informando para incluir uma restrição dinâmica.

3.7.1 OPERACIONALIDADE DO SIMULADOR

A operacionalidade do simulador é dividida em duas parte. A primeira parte é a especificação do CSP, conforme descrito na seção 3.6. Esta especificação será utilizada pelo *framework* para executar o DCSP.

A outra parte é a especificação do ambiente de simulação através de um arquivo Extensible Markup Language (XML), chamado *Semaforos.xml*. Este arquivo é baseado na estrutura descrita por Schmitz (2002, p. 43). O arquivo possui os seguintes atributos:

a) *id_local*: nome do semáforo a ser criado;
b) *id_superior*: nome do semáforo superior;

c) *id_esquerdo*: nome do semáforo a esquerda;

d) *id_direito*: nome do semáforo a direita;

e) *id_inferior*: nome do semáforo inferior;

f) *largura*: largura em metros do cruzamento;

g) *velocidade*: velocidade da via;

h) *comprimento*: comprimento da via;

i) *verde*: semáforo do cruzamento que inicia com verde;

j) *prox_onda*: próximo semáforo para formação da onda verde;

k) *taxa_entrada*: taxa de entrada dos carros na via, expressa em segundos.

A onda verde é definida utilizando-se o atributo *prox_onda*. Neste atributo deve-se especificar qual será o próximo semáforo na onda verde, para que o simulador tenha condições de montar as restrições que serão enviadas ao *framework*.

Para análise da simulação, o simulador apresenta uma tela, onde são exibidos os semáforos presentes na simulação, seu estado (verde ou vermelho) e a quantidade de carros esperando o direito de passagem. Ao final da simulação é gerado um arquivo chamado *LogSimulacao.txt* no diretório *dcsp/bin*, contendo os estado que o semáforo passou, juntamente com a data e hora que este estado foi alterado.

3.7.2 ESTUDO DE CASO

Para demonstração do protótipo foi criado um estudo de caso contendo 4 cruzamentos, com 4 semáforos em cada cruzamento. A velocidade de todas as vias é a mesma, 40 km/h (11 m/s). A largura de todos os semáforos também é a mesma, 10 metros. A Figura 13, apresentada na seção 3.6, ilustra o ambiente que será simulado.

A onda verde que se deseja gerar é representada pela linha pontilhada na cor verde. O Quadro 9 apresenta o arquivo XML de entrada para o simulador com a definição do ambiente a ser simulado.

```
<Cidade>
  <cruzamento id_cruzamento="c1" verde="c1s1">
       <semaforo id_local="c1s1"  id_superior="c3s9" id_esquerdo="c2s8"
                  id_direito="c0s0" id_inferior="c0s0" largura="10" velocidade="11"
                  comprimento="280" verde="s1" prox_onda="" taxa_entrada="1"/>
       <semaforo id_local="c1s2"  id_superior="c0s0" id_esquerdo="c3s9"
                 id_direito="c0s0"  id_inferior="c2s6" largura="10" velocidade="11"
                 comprimento="300" verde="s1" prox_onda="" taxa_entrada="2"/>
       <semaforo id_local="c1s3"  id_superior="c0s0" id_esquerdo="c0s0"
                 id_direito="c2s8" id_inferior="c3s11" largura="10" velocidade="11"
                 comprimento="250" verde="s1" prox_onda="c2s8" taxa_entrada="1"/>
       <semaforo id_local="c1s4"  id_superior="c2s8" id_esquerdo="c0s0"
                 id_direito="c3s9" id_inferior="c0s0"  largura="10" velocidade="11"
                 comprimento="400" verde="s1" prox_onda="" taxa_entrada="1"/>
  </cruzamento>
  <cruzamento id_cruzamento="c2" verde="c2s5">
       <semaforo id_local="c2s5"  id_superior="c0s0" id_esquerdo="c0s0"
                 id_direito="c1s2" id_inferior="c0s0"  largura="10" velocidade="11"
                 comprimento="280" verde="c2s2" prox_onda="" taxa_entrada="2"/>
       <semaforo id_local="c2s6"  id_superior="c1s2" id_esquerdo="c0s0"
                 id_direito="c0s0" id_inferior="c0s0"  largura="10" velocidade="11"
                 comprimento="400" verde="c2s2" prox_onda="" taxa_entrada="3"/>
       <semaforo id_local="c2s7"  id_superior="c0s0" id_esquerdo="c1s2"
                 id_direito="c0s0" id_inferior="c0s0"  largura="10" velocidade="11"
                 comprimento="280" verde="c2s2" prox_onda="" taxa_entrada="1"/>
       <semaforo id_local="c2s8"  id_superior="c0s0" id_esquerdo="c0s0"
                 id_direito="c0s0" id_inferior="c2s4" largura="10" velocidade="11"
                 comprimento="300" verde="c2s2" prox_onda="" taxa_entrada="2"/>
  </cruzamento>
  <cruzamento id_cruzamento="c3" verde="c3s9">
       <semaforo id_local="c3s9"  id_superior="c4s13" id_esquerdo="c0s0"
                 id_direito="c0s0" id_inferior="c1s1"  largura="10" velocidade="11"
                 comprimento="250" verde="c3s9" prox_onda="" taxa_entrada="2"/>
       <semaforo id_local="c3s10" id_superior="c0s0" id_esquerdo="c4s13"
                 id_direito="c1s3" id_inferior="c0s0" largura="10" velocidade="11"
                 comprimento="400" verde="c3s9" prox_onda="" taxa_entrada="1"/>
       <semaforo id_local="c3s11" id_superior="c1s3"  id_esquerdo="c0s0"
                 id_direito="c0s0" id_inferior="c4s15" largura="10" velocidade="11"
                comprimento="400" verde="c3s9" prox_onda="c1s3" taxa_entrada="1"/>
       <semaforo id_local="c3s12" id_superior="c0s0" id_esquerdo="c1s3"
                 id_direito="c4s13" id_inferior="c0s0" largura="10" velocidade="11"
                 comprimento="200" verde="c3s9" prox_onda="" taxa_entrada="2"/>
  </cruzamento>
  <cruzamento id_cruzamento="c4" verde="c4s13">
       <semaforo id_local="c4s13" id_superior="c0s0" id_esquerdo="c0s0"
                  id_direito="c0s0" id_inferior="c3s9" largura="10" velocidade="11"
                  comprimento="400" verde="c4s13" prox_onda="" taxa_entrada="2"/>
       <semaforo id_local="c4s14" id_superior="c0s0" id_esquerdo="c0s0"
                 id_direito="c3s11" id_inferior="c0s0" largura="10" velocidade="11"
                 comprimento="300" verde="c4s13" prox_onda="" taxa_entrada="3"/>
       <semaforo id_local="c4s15" id_superior="c3s11" id_esquerdo="c0s0"
                 id_direito="c0s0" id_inferior="c0s0" largura="10" velocidade="11"
             comprimento="250" verde="c4s13" prox_onda="c3s11" taxa_entrada="1"/>
       <semaforo id_local="c4s16" id_superior="c0s0" id_esquerdo="c3s11"
                 id_direito="c0s0" id_inferior="c0s0" largura="10" velocidade="11"
                 comprimento="300" verde="c4s13" prox_onda="" taxa_entrada="1"/>
  </cruzamento>
</Cidade>
```

Quadro 9 – Arquivo *Semaforos.xml*

O Quadro 10 apresenta a classe com a definição do CSP para o estudo de caso. Nesta classe são criadas as 16 variáveis representando os semáforos, o nome da variável deve ser o mesmo do *id_local* usado no arquivo *Semaforos.xml*.

A classe *CruzamentoCSP* deve ser salva no diretório *examples/cruzamento,* que é um subdiretório de *framework.* O arquivo *Semaforos.xml*, deve estar no diretório *dcsp/bin* do *framework.* Para iniciar a execução do simulador deve-se inicializar o SACI nas máquinas que farão parte da simulação e em seguida inicializar o *framework* DynDCSP. Ao executar o *framework* será apresentada uma tela para seleção dos parâmetros referentes a execução do DCSP, como pode ser visto na Figura 15.

```
public CruzamentoCSP() {
       super("CruzamentoCSP");

       Integer verde    = new Integer(1);
       Integer vermelho = new Integer(2);

       // creating domains
       Domain semaphoresDomain = new IntegerDomain("semaphoresDomain",
                                    verde.intValue(), vermelho.intValue());

       // Cria 4 cruzamentos.
       for (int i = 0; i < 4; i++) {
              for (int j = 0; j < 4; j++) {
                     Variable av = new Variable("c" + (i + 1) + "s" + ((i * 4) +
                                                                      j + 1));
                     av.setDomain(semaphoresDomain);
                     addVariable(av);
              }
       }
       // Cria restrições para todos os semáforos do cruzamento 1
       ArrayList valuesC1 = new ArrayList(0);
       for (int i = 0; i < 4; i++) {
           valuesC1.add(new VariableExpression((Variable) getVariable("c1s" +
                                                                (i + 1))));
       }
       Expression atlC1 = new AtLeastExpression(valuesC1, 1, verde);
       Expression amC1 = new  AtMostExpression(valuesC1, 1, verde);
       addConstraint(new Constraint(atlC1));
       addConstraint(new Constraint(amC1));

       // Cria restricoes para todos os semaforos do cruzamento 2
       ArrayList valuesC2 = new ArrayList(0);
       for (int i = 4; i < 8; i++) {
           valuesC2.add(new VariableExpression((Variable) getVariable("c2s" +
                                                                 (i + 1))));
       }
       Expression atlC2 = new AtLeastExpression(valuesC2, 1, verde);
       Expression amC2 = new  AtMostExpression(valuesC2, 1, verde);
       addConstraint(new Constraint(atlC2));
       addConstraint(new Constraint(amC2));

       // Cria restricoes para todos os semaforos do cruzamento 1
       ArrayList valuesC3 = new ArrayList(0);
       for (int i = 8; i < 12; i++) {
           valuesC3.add(new VariableExpression((Variable) getVariable("c3s" +
                                                                (i + 1))));
       }
       Expression atlC3 = new AtLeastExpression(valuesC3, 1, verde);
       Expression amC3 = new  AtMostExpression(valuesC3, 1, verde);
       addConstraint(new Constraint(atlC3));
       addConstraint(new Constraint(amC3));

       // Cria restricoes para todos os semaforos do cruzamento 4
       ArrayList valuesC4 = new ArrayList(0);
       for (int i = 12; i < 16; i++) {
           valuesC4.add(new VariableExpression((Variable) getVariable("c4s" +
                                                               (i + 1))));
       }
       Expression atlC4 = new AtLeastExpression(valuesC4, 1, verde);
       Expression amC4 = new  AtMostExpression(valuesC4, 1, verde);
       addConstraint(new Constraint(atlC4));
       addConstraint(new Constraint(amC4));
}
```

Quadro 10 – Classe *CruzamentoCSP*

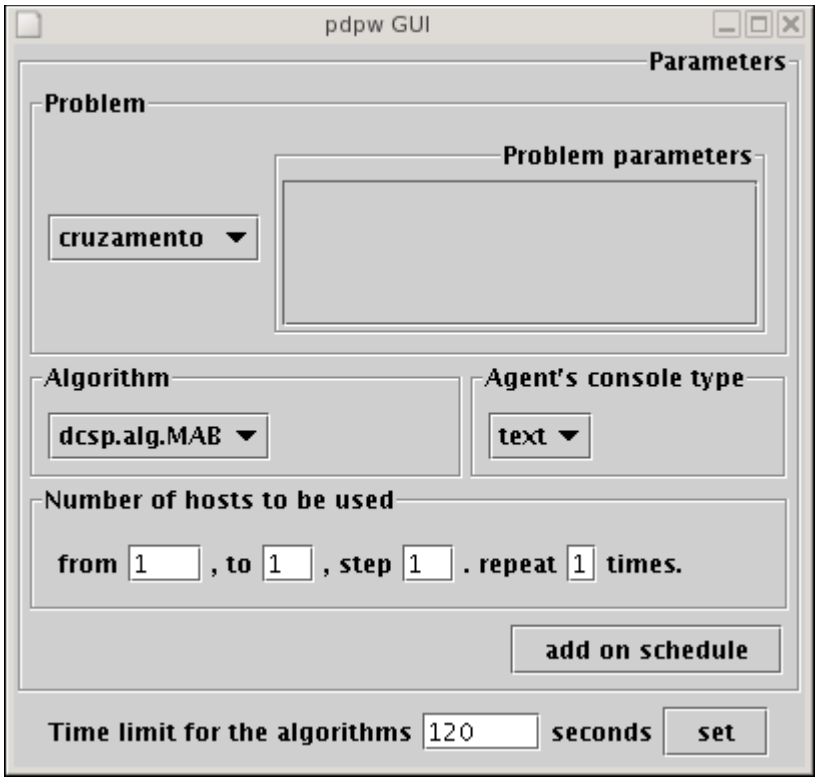


Figura 15 - Tela do *framework* com parâmetros da execução do DCSP

Deve-se selecionar o problema a ser executado, que neste caso chama-se cruzamento, e pressionar o botão *add to schedule*. A execução é inicializada, o agente simulador apresenta uma tela mostrando os semáforos, o estado de cada semáforo e quantos carros estão esperando no semáforo. Esta tela é apresentada na figura 16. A simulação prossegue até que o botão *Fechar* seja pressionado. Ao final da simulação o arquivo de *log LogSimulacao.txt* é gerado, contendo informações sobre a hora que cada semáforo mudou de estado, um exemplo que um arquivo de *log* é apresentado no quadro 11.

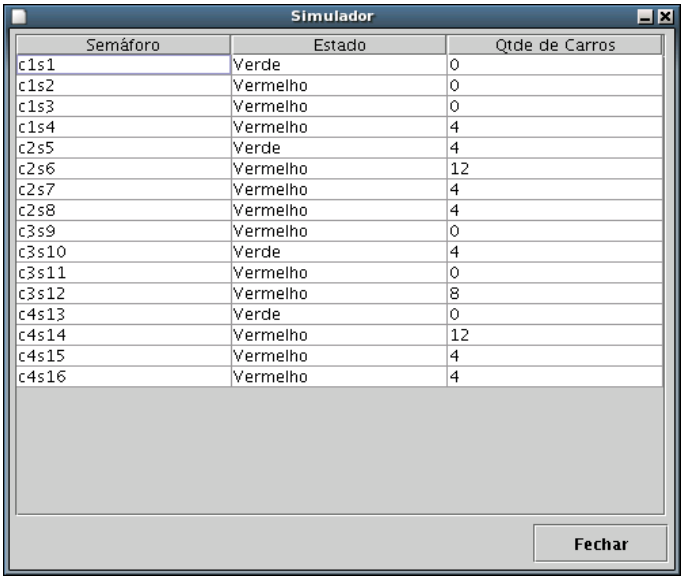


Figura 16 - Tela com as informações do simulador

```
Em Thu Jan 27 13:30:29 BRST 2005 c3s9 iniciou fase Vermelho
Em Thu Jan 27 13:30:30 BRST 2005 c3s10 iniciou fase Verde
Em Thu Jan 27 13:30:34 BRST 2005 c1s1 iniciou fase Vermelho
Em Thu Jan 27 13:30:34 BRST 2005 c1s4 iniciou fase Verde
Em Thu Jan 27 13:30:34 BRST 2005 c2s5 iniciou fase Vermelho
Em Thu Jan 27 13:30:34 BRST 2005 c2s8 iniciou fase Verde
Em Thu Jan 27 13:30:34 BRST 2005 c3s10 iniciou fase Vermelho
Em Thu Jan 27 13:30:34 BRST 2005 c3s12 iniciou fase Verde
Em Thu Jan 27 13:30:34 BRST 2005 c4s13 iniciou fase Vermelho
Em Thu Jan 27 13:30:34 BRST 2005 c4s16 iniciou fase Verde
Em Thu Jan 27 13:30:37 BRST 2005 c1s3 iniciou fase Verde
Em Thu Jan 27 13:30:37 BRST 2005 c1s4 iniciou fase Vermelho
Em Thu Jan 27 13:30:37 BRST 2005 c2s7 iniciou fase Verde
Em Thu Jan 27 13:30:37 BRST 2005 c2s8 iniciou fase Vermelho
Em Thu Jan 27 13:30:37 BRST 2005 c3s11 iniciou fase Verde
```

Quadro 11 – Arquivo de *log LogSimulacao.txt*

4 RESULTADOS

Com o objetivo de tentar resolver o problema de sincronismo de semáforos viários este trabalho propôs-se a utilizar a técnica de satisfação de restrições distribuídas. Para tanto utilizou-se o *framework* DynDCSP que implementa algoritmos para resolução de tais problemas.

Ao iniciar o desenvolvimento deste trabalho constatou-se que o problema em questão necessitaria de funcionalidades que não existiam no *framework* utilizado. Tais funcionalidades são: obtenção de todas as possíveis soluções de um CSP, utilização de variáveis locais e utilização de restrições dinâmicas.

A utilização da técnica de DCSP mostrou-se interessante pois pode ser utilizada em vários tipos de problemas sem a necessidade de alteração nos algoritmos, bastando apenas redefinir as variáveis e suas restrições.

Como resultado conseguiu-se o sincronismo de uma seqüência de semáforos, conforme estabelecida no arquivo de configuração do ambiente que foi simulado. O simulador não obteve um rendimento máximo, gerando congestionamentos nas vias que não faziam parte da seqüência definida para sincronismo. Para resolver este problema seria necessário a utilização de restrições com prioridade, o que resolveria o fato do CSP se tornar insolúvel no caso de dois semáforos no mesmo cruzamento necessitarem da cor verde devido a restrição do sincronismo e a restrição do número máximo de carros esperando o direito de passagem.

Os resultados obtidos através da resolução do DCSP foram selecionados de forma aleatória. Isto pode gerar a seleção de um resultado não ótimo em alguns casos. Para resolver tal problema seria necessária a implementação de um algoritmo para resolução de DCSP que retornasse somente a solução ótima. Um proposta para tal algoritmo é apresentada por Modi (2003).

No caso da simulação de tráfego viário nem sempre conseguiu-se obter a solução para o DCSP no tempo necessário para se decidir qual o estado de cada semáforo. Para solucionar este problema utilizou-se uma função que avalia o tempo que os veículos estão esperando em cada semáforo e decide qual semáforo deve receber a cor verde naquele instante.

O *framework* DynDCSP foi de grande utilidade para obtenção dos resultados propostos por implementar os algoritmos para resolução de DCSP e abstrair a parte de comunicação distribuída entre agentes. Por se tratar de um projeto em desenvolvimento nem todas as características necessárias para resolução do problema proposto estavam disponíveis, sendo estas implementadas neste trabalho.

A utilização do ambiente de programação Eclipse ajudou no desenvolvimento deste trabalho fornecendo um ambiente de programação integrado com a linguagem de programação Java e um depurador de código que permitiu encontrar erros de forma mais rápida.

5 CONCLUSÕES

Este trabalho apresentou uma possível modelagem para o problema de sincronismo de semáforos viários utilizando a técnica de satisfação de restrições distribuídas. Definiu-se cada semáforo como uma variável, sendo que, cada variável poderia assumir os valores verde e vermelho. Sobre os semáforo presentes em um mesmo cruzamento foram aplicadas duas restrições que não são alteradas o passar da execução. Essas restrições garantem que, somente um semáforo no cruzamento receberá o valor verde e que pelo menos um dos semáforos receberá a cor verde. Para que o sincronismo aconteça são usadas restrições dinâmicas, que são incluídas no problema conforme a necessidade.

O *framework* DynDCSP, utilizado para desenvolver o protótipo facilitou a implementação do mesmo, pois implementa os algoritmos mais utilizados para solucionar dos problemas de satisfação de restrições. Foram implementadas funcionalidades extras para permitir que fossem encontradas todas as soluções para os problemas, para que fosse possível definir variáveis locais as variáveis do DCSP e para permitir a utilização de restrições dinâmicas.

A utilização da técnica de satisfação de restrições distribuídas apresentou-se de maneira interessante, pois não necessita de alterações nos algoritmo para executar problemas diferentes. Bastando apenas redefinir o problema, isto é, suas variáveis, domínios e restrições.

5.1 EXTENSÕES

Como extensão a este trabalho, poderia-se enriquecer a simulação adicionando controle tráfego de pedestres e priorização de veículo especiais, como ambulâncias e ônibus. Uma maneira de fazer isto seria através da adição de novas restrições.

Outra extensão poderia se desenvolvida para permitir que o protótipo alterasse o sentido da onda verde conforme a necessidade do ambiente no momento, garantindo assim

que a prioridade de passagem fosse sempre para o sentido com maior tráfego. Para isto poderia ser utilizada a técnica descrita por Oliveira et al (2004)

Para uma melhor análise da situação do ambiente que está sendo simulado poderia-se desenvolver um *interface* gráfica para o simulador, a qual representaria o estado dos semáforos de forma mais rápida, possibilitando uma melhor compreensão dos resultados.

Com relação ao *framework* DynDCSP, sugere-se implementação de algoritmos para otimização de DCSP. Desta forma o resultado encontrado seria o melhor resultado possível. Um algoritmo para otimização de DCSP é apresentado por Modi (2003).

Outra funcionalidade que poderia se implementada no *framework* seria a inclusão de variáveis locais com domínio, possibilitando que o agente escolhesse os valores para essas variáveis.

REFERÊNCIAS BIBLIOGRÁFICAS

CARVALHO, Rêmulo Dias de. **Simulação de um sistema reativo e centralizado de coordenação de sinais de tráfego utilizando a programação em lógica com restrições.** Belo Horizonte, [2000]. Disponível em: <http://www.dcc.ufmg.br/pos/html/spg2000/anais/remulo/remulo.html >. Acesso em: 17 out. 2004.

DYNDCSP. **DynDCSP**, Blumenau, [2003]. Disponíveis em: <http://www.inf.furb.br/gia/dynDCSP/>. Acesso em 10 jan. 2004.

ESPEL, Marcelo. **O controle eficaz dos semáforos para melhoria do tráfego urbano**. 2000, 53 f. Monografia (Especialista em Gestão Integrada de Trânsito) - Universidade Católica de Santos, Santos.

MODI, Pragnesh J. et al. **An asynchronous complete method for distributed constraint optimization**. In: International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'2003), 2., 2003, Melbourne, Australia. Anais**...** [S.l.]: ACM Press, 2003.

OLIVEIRA, D. et al. **A Swarm-based Approach for Selection of Signal Plans in Urban Scenarios**. In: IV International Workshop on Ant Colony Optimization and Swarm Intelligence (ANTS 2004), 2004, Bruxelas.

SCHMITZ, Marcelo; HÜBNER, Jomi Fred. **Uso de SMA para avaliar estratégias de decisão no controle de tráfego urbano**. In: Seminário de Computação, 11., 2002, Blumenau. **Anais...** Blumenau: FURB, 2002. v. 1, p. 243-254.

SCHMITZ, Marcelo. **Sistema de controle de tráfego urbano utilizando sistemas multi-agentes**. 2002. 62 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SILVA, Paulo C. M. **Elementos dos sistemas de tráfego**, Brasília, 2001. Disponível em: <http://www.unb.br/ft/enc/pagdisc/engtraf/apostilas/APOSTILA1.pdf>. Acesso em: 5 set. 2004.

SINCMOBIL. **Sistema de informação e controle para mobilidade urbana**, Florianópolis, [2003]. Disponível em: <http://www.das.ufsc.br/sincmobil>. Acesso em: 20 nov. 2004.

STERN, Yvone et al. **Um estudo sobre tráfego:** sincronização de sinais. Rio de Janeiro: Instituto de Pesquisas Rodoviárias, 1969.

TRALAMAZZA, Daniel. D**esenvolvimento 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.

TSANG, Edward. **Foundations of constraint satisfaction**. London: Academic Press, 1993. ISBN 0-12-701610-4.

YOKOO, Makoto. **Distributed constraint satisfaction**. Berlin: Springer, 2001.