Lígia Maria Rodrigues da Silva Ferreira

Programação por Restrições Distribuídas em Java

Dissertação apresentada para a obtenção do Grau de Doutor em Informática, pela Universidade de Évora.

Orientadador: Salvador Luís Bethencourt Pinto Abreu

2004

"Esta tese não inclui as críticas e sugestões feitas pelo júri"

Lígia Maria Rodrigues da Silva Ferreira

Programação por Restrições Distribuídas em Java

Dissertação apresentada para a obtenção do Grau de Doutor em Informática, pela Universidade de Évora.

Orientadador: Salvador Luís Bethencourt Pinto Abreu

2004

"Esta tese não inclui as críticas e sugestões feitas pelo júri"

Agradecimentos

Gostaria de agradecer ao meu orientador, o professor Salvador Abreu, pela atenção e disponibilidade dispensadas em todos os momentos necessários. Também lhe gostaria de agradecer pelo rigor que impôs à minha escrita.
À Irene pela motivação dada para que concluísse esta tese o mais rapidamente possível.
Ao Luís pela paciência demonstrada e por estar sempre ao meu lado mesmo que só em pensamento. Ao meu filho Zé Pedro, por sobretudo na fase final da escrita desta tese, ter crescido tão depressa e se ter tornado autónomo num conjunto de tarefas que os adultos desempenham facilmente mas as crianças não e por se entreter longas horas a jogar consola e a ver televisão.
À Milucha por ter sido das únicas pessoas da minha família que assegurava algumas das minhas tarefas de mãe e me libertava para que pudesse desempenhar este trabalho com algum desafogo e consciência tranquila.
À São e ao Jerónimo, por na falta da Milucha me ajudarem na medida do possível. À Mimicas, que sei que não ajudou mais por estar demasiado longe.
A todos os restantes membros da minha família pelo apoio moral.
A estes e a todos aqueles de que me possa ter esquecido, obrigado.

Resumo

Após uma breve introdução à problemática da resolução de sistemas de restrições de domínios finitos e da apresentação das mais conhecidas (hoje) implementações de domínios finitos na linguagem Java, são apresentadas nesta tese, duas abordagens possíveis e duas implementações em Java para resolvedores de restrições sobre domínios finitos. As principais contribuições originais desta tese são:

- Uma abordagem orientada a objectos à implementação dum resolvedor de restrições com iteração de variáveis, o *GC*. Este resolvedor é providenciado como uma biblioteca Java.

- Uma abordagem baseada em múltiplos estados independentes, o *AJACS*. Tal como a abordagem anterior, o *AJACS* é facultado numa biblioteca Java, que pode ser extendida. A resolução dum sistema de restrições com esta abordagem, passa por gerar novos estados em que a partilha com os estados anteriores, se a houver, é só em leitura.

- Vários modelos concorrentes para a exploração do espaço de estados anterior.

- Uma implementação paralela com threads Java, que implementa estes modelos.

- Uma implementação distribuída, assente num sistema DSM, do modelo de estados múltiplos.

- Uma aplicação da implementação distribuída a problemas de construção de horários.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução e Motivação

Neste capítulo vamos introduzir a Programação por Restrições, começando na Programação em Lógica com Restrições e acabando com a análise das livrarias de restrições para o Java. Pelo meio ficam as principais técnicas para abordar a problemática em mãos: os sistemas de resolução de restrições sobre domínios finitos.

1.1 Introdução

As Linguagens de Programação com Restrições tiveram a sua génese na Programação em Lógica com Restrições (CLP). Estas são uma generalização da Programação em Lógica(LP) resultante da substituição do mecanismo de unificação (operação básica das linguagens LP) pelo da resolução de restrições.

As linguagens de programação em lógica com restrições emergentes por uso desta técnica, combinam as vantagens da Programação em Lógica (não-determinismo, forma relacional e semântica declarativa) com a eficiência dos algoritmos para a resolução de restrições. A implementação da primeira geração deste tipo de linguagens, data do fim dos anos 80 e a ela pertencem linguagens como o CHIP [MDB88], CLP(R) [JJY] e Trilogy [Vod88]. Vários foram os sistemas de restrições na altura investigados para serem incluídos nas linguagens CLP: álgebras de Boole, Aritmética Linear de Racionais, Programação Linear Inteira, e os Domínios Finitos. Algumas linguagens incorporam mais do que um sistema de restrições, outros apenas um. Os domínios de aplicabilidade das CLPs são variados, indo desde os problemas combinatórios da IO (Investigação Operacional) até ao Design de Hardware passando pelos Sistemas de Apoio à Decisão.

A introdução das ideias de base da CLP[JL87] noutro paradigma da PL, o da programação lógica concorrente, fez surgir outro tipo de linguagens designadas Restrições Concorrentes (CC)"Concurrent Constraints"[Sar89]. O conceito de restrição implicada ("constraint entailment"), i.e. decidir quando uma restrição é implicada por uma conjunção de restrições existente, surgiu no contexto destas linguagens por necessidade de sincronizar os diferentes agentes de execução concorrentes.

As restrições implicadas catalogaram em duas famílias as linguagens de programação com restrições: Linguagens "Tell" e Linguagens "Ask and Tell". As primeiras têm uma operação primitiva, a resolução de restrições, as segundas têm duas, a resolução de restrições e a implicação de restrições. A implicação de restrições foi usada por [Sar89] para definir o operador implicação no contexto da programação em lógica concorrente mas foi também usado para definir o operador cardinalidade explicitamente para as linguagens CLP. O operador de

cardinalidade surgiu da tentativa de providenciar ao utilizador duma CLP uma forma de definir as suas próprias restrições em função dum conjunto básico de restrições providenciado pela linguagem. Como aplicação deste operador temos por exemplo as restrições *element* ou *atmost* .

São várias as implementações de linguagens de programação em lógica com restrições , para citar algumas temos por exemplo `clp(fd)` [CD96a], CHIP [MDB88], `cc(fd)` [PVHD95], `CLP(R)` [JJY], `AKL`, [BCH94] `OZ`[DGN98], [Hen97] ou `CIAO` [MHP99], mas a implementação de linguagens de programação por restrições fora do contexto da programação em lógica foi um assunto inovador, à época, abordado nas implementações do `ILOG solver`[PL95] ou o `2LP` [MT95].

As motivações para introduzir um resolvedor de restrições em Linguagens marginais à PL são providenciar uma técnica que está suficiente maturada e de aplicabilidade indiscutível, a outro tipo de utilizadores e de aplicações. O desenvolvimento de aplicações de "uso" industrial, faz-se recorrendo a outras linguagens. Sem dúvida a linguagem C(C++) era (é) uma das mais populares e eficientes sendo este um dos motivos que estiveram na base da escolha desta linguagem para a implementação duma biblioteca de restrições para a linguagem C++, o `ILOG solver`.

Nos dias que correm a linguagem Java assumiu grande protagonismo [1] ao qual não será certamente alheia a portabilidade oferecida pela linguagem e a popularidade crescente duma tecnologia que difundiu a própria linguagem, a Internet. Terá portanto cabimento a definição de bibliotecas de restrições para o Java tentando assim dotar esta linguagem, de tais capacidades. Na secção 1.4 serão abordadas estas propostas.

O resto do capítulo está organizado do seguinte modo: na secção 1.2 são apresentados os principais objectivos deste trabalho, na secção 1.3 são apresentados os domínios finitos e as técnicas usadas para resolver estes sistemas de restrições, na secção 1.4 são brevemente discutidos alguns sistemas de programação por restrições am Java e na secção 1.5 é apresentado um guia de leitura da tese.

[1] Não vamos aqui abordar o assunto da "popularidade" das linguagens, nem os motivos que as posicionam num ranking de popularidade, que são sem dúvida variados em número e em género

1.2 Objectivos

Existem várias propostas à introdução da programação por restrições na linguagem Java: JACK [SAS02], JCL [TWF97], JSolver [Chu99] e DJ [NFZY98], são algumas e são discutidas na secção 1.4. Esta tese apresenta duas bibilotecas de restrições para a linguagem Java, o GC [FA00b] e o AJACS [FA01]. Com estas abordagens pretendemos:

- Introduzir na linguagem Java o paradigma da programação por restrições, facultando uma biblioteca de restrições que possa ser integrada num qualquer programa Java de forma clara e natural. Deverá ser possível ao utilizador extender a biblioteca criando nova subclasses que providenciem a exploração das classes de base, na prespectiva Orientada-a-Objectos proporcionada pelo Java.

- Permitir que as aplicações criadas pelos utilizadores do AJACS possam ser executadas num ambiente paralelo e distribuído, também de forma transparente para o utilizador. A construção de aplicações com o AJACS não deverá exigir nenhum esforço adicional por parte do utilizador na elaboração dos seus programas, por esta potencialidade ser oferecida. É sabido que a construção de programas paralelos requer um esforço adicional de organização de código e tem uma depuração ("debugging") complexa e falível. O AJACS providenciará esta possibilidade minimizando esse esforço.

1.3 Domínios Finitos: uma introdução

Os sistemas de restrições devem providenciar um resolvedor de restrições para decidir da sua satisfação. Os mais conhecidos sistemas de restrições seja pela número aplicações industriais que suscitam, seja por os respectivos resolvedores estarem bastante bem documentados são as Álgebras de Bool, Aritmética linear de Racionais, e os Domínios finitos. Restringir-nos-emos aos Domínios Finitos, pelo que os resolvedores do GC e do AJACS incidirão exclusivamente sobre estes domínios. No decorrer desta secção sempre que falarmos de sistemas de restrições

estamos a referir-nos aos Domínios Finitos.

Um sistema de restrições é caracterizado por:

- Um conjunto de variáveis $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$.
- Um conjunto de restrições $\mathcal{C} = \{c_1, \ldots, c_k\}$. A natureza das restrições poderá ser:
  - de domínio $x_i \in D_i$, para D_i um conjunto discreto e finito, ou
  - aritméticas($ax \neq b$, $ax = b$ $ax = by + c$ $ax \geq by + c$, $ax \leq by + c$). Todas as variáveis que intervêm nas restrições aritméticas devem ter pelo menos uma restrição de domínio associada.

  Um sistema de restrições pode por este motivo ser definido por: $\mathcal{P} \equiv ((\mathcal{X}, \mathcal{D}); \mathcal{C})$ em que $\mathcal{D}$ é definido por $\{D_1, D_2, \ldots, D_n\}$, sendo cada D_i definido pela intersecção de todos os D_k tais que $x_i \in D_k$ é uma restrição de Domínio sobre x_i e $\mathcal{C}$ o conjunto de restrições aritméticas sobre as variáveis de $\mathcal{X}$.

$s = (s_1, s_2, \ldots, s_n)$ é solução de $\mathcal{P}$, sse cada s_i corresponde à atribuição de um valor de D_i para cada um dos x_i de $\mathcal{X}$ ($x_i = s_i \in D_i$) e todas as restrições de $\mathcal{C}$ são satisfeitas por s, i.e. $\forall c \in \mathrm{C}, c(s_i, s_j)$ é verdadeiro .

1.3.1 Resolvedores de Domínios Finitos

Existem grosso modo duas grandes categorias de processos para determinar as soluções dum sistema de restrições: Algoritmos de Pesquisa e Algoritmos de consistência.

Pesquisa

Um modo óbvio de encontrar soluções para um sistema de restrições é por tentativa e teste (*Generate and Test*). Este processo gera todas as combinações possíveis de variável/valor do domínio e verifica à posteriori, se as combinação geradas são consistentes com todas as restrições do problema. Trata-se dum processo que apesar da sua completude é certamente moroso e impraticável se a

combinatória do problema for elevada. Uma dos desvantagens desta abordagem, é que atendendo que as fases de geração e teste são separadas as instanciações conflituosas são repetidas tantas vezes, quantas o produto cartesiano das restantes variáveis o permitir.

Como exemplo se $x_1 > x_2$ for uma restrição e $\{1, 10\}$ o domínio de ambas as variáveis, todas as instanciações do valor 1 para a variável x_1, são conflituosas, visto não existir em todo o domínio de x_2 qualquer valor que satisfaça a restrição. No entanto esta instanciação é repetida, neste caso 10 vezes, que corresponde a todos os possíveis valores de x_2.

Backtracking (retrocesso)

O método de backtracking resulta de misturar as fases de geração e teste do método *GT*. As soluções são construídas instanciado uma a uma as varáveis do problema, construindo soluções parciais. Uma solução parcial, é um tuplo de variáveis umas já instanciadas, outras não. Mais ainda as restrições que referem somente as variáveis instanciadas não geram conflito, daí o nome de solução parcial. Quando se detectam conflitos, escolhe-se novo valor para a variável recentemente instanciada e que gerou o conflito.

Seja $s = (s_1, \ldots, s_{i-1}, x_i, \ldots, x_n)$ uma solução parcial. Se ao instanciar x_i é detectado um conflito, são agora poupadas as instanciações correspondentes às variáveis $x_{i+1}, \ldots x_n$. No entanto como foi ao instanciar x_i que se detectou o conflito, vão-se atribuindo a x_i outros possíveis valores do seu domínio na tentativa de solucionar o conflito. Apesar de serem agora mais cedo detectados os conflitos, existindo portanto cortes na árvore de pesquisa, trata-se de um método em que não é possível saber qual a origem do conflito, assumindo-se que é na variável responsável pela detecção do mesmo, o que pode até nem ser verdade.

Técnicas de Consistência

As técnicas de consistência foram introduzidas pela primeira vez por Codd [Cod70] e importadas da área da Inteligência Artificial através de Mackworth [Mac77]. Um sistema de restrições pode ser representado por um grafo $\mathcal{G}$, em que o con-

junto dos nós $\mathcal{N}$ é constituído pelas variáveis do problema e os arcos $\mathcal{A}$ são as restrições. Neste caso só poderão ser representadas restrições unárias(existe um arco dum nó para si próprio) e binárias. As restrições básicas, de domínio e aritméticas, dos domínios finitos são-no, mas no caso geral esta imposição à aridade das restrições não reduz a tratabilidade dos casos, uma vez que a equivalência entre um sistema de restrições qualquer e um sistema de restrições de aridade inferior ou igual a 2, está demonstrada [BvB98], [RDP90]. A ideia de base na redução da aridade das restrições é introduzir novas variáveis que encapsulem as variáveis da restrição que queremos eliminar, definindo como domínio da variáveis o produto cartesiano das variáveis encapsuladas.

Consistência de Nó

Um nó é consistente, se as restrições unárias sobre o nó, forem validadas por todos os valores do domínio da variável do nó:
$consistente(x) \iff \forall v \in D_x, \forall c_k \in \mathcal{C} : c_k$é uma restrição unária sobre x, se tem $c_k(v)$.
$\mathcal{G}$ apresenta consistência de nó sse todos os seus nós forem consistentes ($\forall x \in \mathcal{N}, consistente(x)$).

Tornar um nó inconsistente num nó consistente é directo eliminando todos os valores do domínio da variável que provocam inconsistências. A eliminação destes valores não elimina soluções uma vez que qualquer afectação da variável ao valor provocará sempre inconsistência. Se $\mathcal{G}$ é um grafo nó-consistente, as restrições unárias podem ser eliminadas pois a sua consistência não mais necessita ser ser verificada. A consistência de $\mathcal{G}$ prendem-se agora com as restrições binárias.

Consistência de Arco

A consistência das restrições binárias é verificada usando a consistência de arco. Um arco de x para y é consistente se para qualquer valor do domínio de x, existe pelo menos um valor de y, que valide a restrição associada ao arco:

$consistente((x,y)) \iff$
$\forall v \in D_x, \exists v \in D_y : c_k(v,u)$, para c_k a restrição associada ao arco (x,y)

Obtém-se a consistência de um arco (x, y) eliminando-se de D_x os valores para os quais não existe em D_y nenhum valor que valide a restrição. A consistência de arco é direccional, podendo (x, y) ser consistente e (y, x) não. Eliminar todas as inconsistências de arco de um grafo, requer rever as consistências previamente calculadas quando se reduzem domínios. Se D_x foi reduzido, então as consistências dos arcos que terminam em x têm de ser recalculadas.

Seja A o conjunto de todos os arcos de $\mathcal{G}$. Enquanto existirem arcos em A tome-se de A um arco, removendo-o. Seja (x, y) esse arco: Se o arco é inconsistente actualize-se D_x de modo a eliminar inconsistências. Seja D'_x o novo domínio de x, se $D_x \neq D'_x$, actualize-se A adicionando-lhe os arcos que terminem em x para serem novamente revistos $(A \leftarrow A \cup \{(w, x) : w \in N, (w, x) \in \mathcal{G}\})$. Quando A estiver vazio $\mathcal{G}$ é arco-consistente. Se durante o processo algum dos domínios ficar vazio, o problema não tem soluções. As soluções dum problema arco-consistente são encontradas fazendo pesquisa. Poderá até acontecer que um problema seja arco-consistente, nenhum dos domínios seja vazio, e mesmo assim não existam soluções.

O processo descrito é o algoritmo `AC-3` de Mackworth [Mac77] e é um dos processos mais usados para a determinação da arco-consistência dum grafo de restrições. A desvantagem do método é a repetição sistemática da análise da consistência dos valores dos domínios quando a consistência de um arco é revista. Outro processo muito usado (denominado `AC-4` [MH86]) na determinação da arco-consistência, usa estruturas de dados mais complexas sendo previamente calculados as afectações que suportam outras afectações. As revisões são analisadas usando contadores de suportes. Este método define inicialmente os conjuntos:

- $S_{x,u}$, para todo o $x \in \mathcal{N}, u \in D_x$ tal que o par $(y, v) \in S_{x,u}$ $iff x \leftarrow u, y \leftarrow v$ são afectações que validam a restrição associada ao arco (x, y).
- $counter[(x, y), u] = k$ se existem k valores no domínio de y, que validem a restrição associada ao arco (x, y), para $x \leftarrow u$. Se o contador de suportes for 0, retira-se u de D_x, e adiciona-se o par (x,u) ao conjunto Q.
- $Q = \{(x, u)\}$, se $\exists (x, w) \in \mathcal{A} \wedge \nexists v \in D_w$ cuja afectação $x \leftarrow u$ e $v \leftarrow w$ valide a restrição associada ao arco (x,w).

O algoritmo, após inicializar os conjuntos como o descrito, usa Q e S, para manter a consistência de arco. Enquanto Q não estiver vazio, retira de Q um elemento (x, u) e usa $S_{x,u}$. Para todo o par $(y, v) \in S_{x,u}$, decrementa o contador de suportes $counter[(y, x), v]$, visto existir menos um suporte correspondente a $x \leftarrow u$. Se o contador está a zero retira-se v de D_y, e introduz-se (y, v) em Q.

Outros algoritmos para efectuar a consistência de arco: `AC-5` [vHDT92] e `AC-6` em [Bes94]. Este último derivado de `AC-4`, mas menos "completo" nas estruturas de dados usadas já que não são guardados todos os valores de suporte a uma instanciação mas apenas um, quando tal valor desaparece por uma actualização de domínio é procurado outro valor para suporte . `AC-7` [CB99] derivado do `AC-6`, usa simetrias para os conjuntos de suporte (se $(y, v) \in S_{(x,u)}$ então $(x, u) \in S_{(y,v)}$.

k-consistência

A consistência de caminho("path-consistency") ou k-consistência é a extensão da consistência de arco. Um grafo é k-consistente se dados os valores de quaisquer $k-1$ variáveis que validem as restrições a elas associadas existir pelo menos um valor no domínio duma k-ésima variável que valide as restrições associadas às k variáveis. Um grafo é fortemente k-consistente se é j-consistente para todo o $j \leq k$.

A consistência de nó é uma 1-consistência forte. A consistência de arco é equivalente a uma 2-consistência forte. Embora existam algoritmos para tornar k-consistente ($k > 2$) um grafo de restrições, a sua utilização na prática é reduzida por questões de eficiência, [MH86], [HL88]. A importância da n-consistência forte dum grafo de restrições de n variáveis reside na possibilidade de eliminar a pesquisa, dado que com este tipo de consistência a determinação de soluções faz-se por afectação directa dos valores dos domínios às variáveis.

Propagação

A propagação de restrições resulta da mistura das técnicas de backtracking, secção 1.3.1 com as técnicas de consistência secção 1.3.1. Um modo evidente de implementar a abordagem é ir instanciando variáveis e verificar a consistência

dos arcos correspondente as varáveis já instanciadas. Como se instância variável a variável, as consistências analisadas correspondem aos arcos de/para a variável recentemente instanciada para/de uma das variáveis já instanciadas. Como agora os domínios são unitários a actualização de domínio, se existir, invalida a restrição e consequentemente a solução parcial, caso em que se faz "backtracking" instanciando a variável com novo valor. O processo em nada difere do apresentado na secção 1.3.1, mas pode ser melhorado usando técnicas de "look-ahead" ou "look-back".

As técnicas de look-back([Dec90], [KvB97]), são aplicadas quando o algoritmo de backtracking detecta uma inconsistência, e prendem-se com dois assuntos: Há necessidade de recuar, mas qual a medida desse recuo, i.e, recua-se até onde? Obviamente é a análise dos motivos da falha que permite decidir o salto de recuo(" backjumping"). Outro assunto abordado nas técnicas de look-back prende-se com a aprendizagem do erro, permitindo que as situações que conduziram à inconsistência não se repitam adicionando novas restrições ao problema que permitiram evitar as mesmas situações de conflito.

As técnicas de look-ahead visam prevenir, em vez de remediar, (caso do look-back) a ocorrência de conflitos evitando assim alguma da necessidade de fazer backtrack. O modo mais evidente de o conseguir é propagar para a frente e não para trás, isto é, quando se instancia uma variável com determinado valor quais as consequências dessa instanciação nos domínios das variáveis ainda não instanciadas?

"Forward checking", é o processo que instancia variáveis (uma a uma usando uma qualquer ordem entre as variáveis), e verifica a consistência (fraca) dos arcos de $\mathcal{G}$ aos quais pertencem a variável instanciada e uma que não o esteja (não há necessidade de verificar a consistência com instanciações de variáveis anteriores, visto que já foi determinada!). A revisão deste arcos permite reduzir (se for caso disso) os domínios das variáveis por instanciar. Se algum dos domínios das variáveis por instanciar ficar vazio, escolhe-se novo valor para a variável e repete-se o processo, até a obtenção dum solução total.

Pode-se ir ainda mais longe e verificar a consistência(forte) dos arcos correspondentes a domínios actualizados. Por exemplo no forward cheking quando

instanciamos uma qualquer variável v, vamos verificar a consistência dos arcos (x, v), para x uma qualquer variável ainda por instanciar, desde que, claro está, o arco pertença ao grafo. Se D_x foi actualizado podemos também rever os arcos (y, x), para y ainda não instanciado, diferente de x e de v. Este processo é referido por full look-ahead ou MAC("Mantaining arc consistency"). Trata-se claramente dum processo em que os cortes na árvore de pesquisa são maiores do que os do "Forward chaining", mas o custo duma afectação é bastante maior.

1.3.2 Reduzir a Pesquisa

A ordem pela qual são instanciadas as variáveis e a ordem dos valores considerados para cada variável poderá ter um grande impacto na eficiência da pesquisa([HJ80],). A ordem das variáveis a instanciar poderá ser definida dinâmica ou estaticamente. Num processo de backtracking normal, como não há redução dos domínios das variáveis a instanciar, qualquer que seja a ordem pré-definida por análise dos domínios será mantida em run-time. No processo de backtracking com forward checking a ordem pode ser calculada dinamicamente. Algumas heurísticas visam determinar essa ordem, no sentido de minimizar o espaço de pesquisa:

first-fail 1 heurística baseada no princípio homónimo, cuja semântica é "Atacar primeiro os casos mais complicados é uma estratégia que conduz ao sucesso." Na prática a aplicação da heurística selecciona para variável a iterar, a variável onde a probabilidade de falhar é maior. Claramente existem mais hipóteses de falhar tomando as variáveis com menor cardinalidade, pois o leque de valores à escolha é menor, e caso nenhum deles sirva a falha é detectada o mais precocemente possível. Adiar a instanciação destas variáveis não trás nenhum benefício, pois a instanciação tem de ser feita (não há soluções sem todas as variáveis estarem instanciadas), e quanto mais tarde ele for realizada(sem sucesso) mais trabalho é deitado fora;

first-fail 2 Quando os domínios têm a mesma cardinalidade, outros parâmetros que sejam susceptíveis de conduzir a falhas podem ser considerados, por

exemplo tomar as variáveis que participem em restrições cuja satisfação seja difícil, ou porque esta consideração pode ser difícil de quantificar, por exemplo variáveis que participem no maior número de restrições.

first-fail 3 Uma heurística cuja aplicação é também extensível à pesquisa usando um backtracking simples, é considerar como próxima variável a instanciar, a variável que participe no maior número de restrições com as variáveis já instanciadas.

Após a escolha da variável a instanciar, há que escolher um valor para a instaniação.Se o problema é determinar todas as soluções é irrelevante a ordem dos valores, se pretendemos apenas uma não! Ao pretender determinar uma solução quanto mais cedo lá chegarmos melhor. Tal significa que devemos tomar o valor correspondente a uma ramo que conduza a uma solução, não aqueles que conduzem a falhas e obrigam a backtracking. Agora, contrariamente ao caso da escolha de variáveis o princípio a aplicar **succed-first**. A heurística vai escolher agora os valores cujo leque de opções é mais vasto. Só em **AC-4** (secção 1.3.1) são contabilizadas as promessas de cada valor e considerar os valores que tenham maior sustentabilidade nas variáveis por instanciar é compatível com a heurística:

$$\max_{u} \left\{ \prod_{w} count[(x,w),u], \text{ para w por instanciar} \right\}$$

Nos casos mais gerais a escolha do valor poderá não ser compatível com os custos associados, ou ainda não ser possível. Casos concretos poderão induzir uma escolha de valores que seja de algum modo eficiente, de acordo com a heurística ou não. Por exemplo está demonstrado que no caso das *Queens* escolher os valores do meio dos domínios é melhor, visto existirem mais soluções com estes valores do que com os valores dos extremos.

1.3.3 Pesquisa Estocástica

Local Search

A metodologia de pesquisa local, tenta construir soluções a partir de determinadas configurações ou estados. Os estados diferem dos da pesquisa local, por não

existirem soluções parciais. É aleatoriamente gerado uma afectação para todas as variáveis, sendo avaliado o estado e tentando progredir para estados melhores. Existe uma forma de avaliar os estados (determinando o número de restrições violadas pelo estado, eventualmente ponderadas), que induz uma relação de ordem parcial entre estes. Um estado cuja avaliação tem o valor zero é uma solução. A passagem de um estado para outro é determinada avaliando os estados vizinhos e escolhendo a melhor transição. Considera-se como estado vizinho um qualquer estado em que uma das variáveis tem um valor diferente. O problema destas abordagens é ficar "preso" num mínimo-local, i.e. um estado que não é solução e cujos vizinhos têm uma avaliação igual ou superior à sua. Os mínimos-locais podem ainda ser estritos ou não-estritos. No primeiro caso os vizinhos têm todos uma avaliação estritamente superior ao estado, no segundo caso não.

Hill-Climbing A ideia é ir passando de um estado para um vizinho até encontrar um solução. Gera-se inicialmente um estado onde é determinado aleatoriamente um valor para cada uma das variáveis do estado. Se o estado é solução o algoritmo termina, senão avaliam-se os vizinhos. Se o estado é um mínimo local estrito gera-se aleatoriamente novo estado e começa-se de novo, senão transita-se para o estado vizinho com melhor avaliação.

Para evitar ficar também "preso" em mínimos locais não estritos o algoritmo pode ser parametrizado, assumindo que só se fazem n transições para estados de igual avaliação, após o que se recomeça noutro estado gerado aleatoriamente.

Claramente uma das desvantagens do algoritmo resulta da necessidade de avaliar todos os vizinhos (em número de $\sum_{i=1}^{n} \#Di - n$), para determinar a melhor escolha para efectuar as transições.

Heurística dos Mínimos-Conflitos [MJPL92]

Dados: Um conjunto de variáveis, um conjunto de restrições binárias, e uma afectação específica de valores para todas as variáveis. Duas variáveis estão em conflito se os seus valores violam uma restrição.

Procedimento: Seleccione-se uma variável em conflito, e atribua-se-lhe um valor que minimize o número de conflitos.

A heurística pode ser aplicada quer à estratégia *Hill-climbing*: caso em que, em vez da transição ser determinada pela avaliação de todos os vizinhos é escolhida uma variável em conflito, e escolhido um valor para essa variável que reduza o número de conflitos; quer usando o método de *backtracking*. Neste caso, usa-se o método normal de backtracking adicionando a heurística para determinar a variável e o valor a escolher. Por um lado trata-se dum método de pesquisa estocástica, já que todas as variáveis têm atribuído um valor, por outro existem variáveis por reparar, mantidas num conjunto `var-left` e variáveis já reparadas, mantidas em `var-done`. Inicialmente todas as variáveis estão em `var-left` e é construída uma afectação para todas as variáveis. À medida que as variáveis forem sendo reparadas vão sendo adicionadas a `var-done`. Quando todas as variáveis forem reparadas com sucesso, o algoritmo termina. Idealmente o algoritmo determina uma sequência de reparações de modo a que nenhuma variável seja reparada mais do que uma vez. Ao reparar uma variável, é retirada de `var-left` e adicionada a `var-done`, e construída uma lista dos valores possíveis para a variável ordenada por ordem crescente relativamente ao número de conflitos com as variáveis de `var-left`. Para todos os valores e até ser encontrada uma solução, se o valor não está em conflito com nenhuma das variáveis de `var-done`, faz-se a afectação e repete-se o processo usando com os novos `var-left` e `var-done`.

A utilização da heurística usando o método *Hill-climbing* pode tal como o método sem a heurística ficar preso numa mínimo local e não terminar. No segundo caso, i.e. usando backtracking, a solução se existe é encontrada, e caso não exista é reportada a impossibilidade de problema.

Tabu-Search [F.86], [GL95] Para evitar ficar aprisionado nos mínimos locais, existe uma lista de afectações tabu, que são mantidos durante um determinado período de tempo (simulação duma memória de curto prazo). A estratégia consistente em evoluir para configurações que não envolvam mo-

vimentos tabu (um movimento é representado pelo par variável valor), e que estejam mais perto da solução. O tamanho da lista tabu é mantida, retirando o movimento mais antigo e introduzindo na lista o movimento (X,v), quando se passa duma configuração s a s', substituindo a afectação (X,v) existente em s pela afectação (X,v') para obter a configuração s'. A eficiência do método está directamente relacionada com o tamanho da lista. Valores entre 10 e 35 parecem ser apropriados.

Outros algoritmos de pesquisa estocástica, como o Simulated Annealing, Adaptive Search [CD01], ou WSAT [SM92], podem ser consultados pelos leitores mais interessados.

1.4 Bibliotecas de restrições em Java

São apresentadas nesta secção as principais abordagens à problemática da resolução de restrições em Java. A ordem pela qual as abordagens figuram na descrição é puramente aleatória.

JACK "Java Constraint Kit" [SAS02], é uma biblioteca de restrições para o Java, composta por três módulos: o JCHR, o JASE e o VisualCHR. O módulo JCHR providencia uma implementação à la Java da linguagem de alto nível CHR(Constraint Hamdling Rules)[Frü98] especializada na escrita de resolvedores de restrições, quer para definir desde o "zero" um resolvedor, quer para alterar um resolvedor já definido. O CHR permite definir quer a propagação quer a definição de novas restrições (restrições definidas pelo utilizador). Programas escritos em JCHR são compilados para Java usando o compilador de JCHR. Dado que a definição do resolvedor de restrições não é regra geral suficiente para solucionar um sistema de restrições é facultado num módulo à parte, o JASE, responsável pela definição de estratégias de pesquisa. Este módulo permite implementar estratégias definidas pelo utilizador (daí o nome abstracto), providenciando também algumas estratégias clássicas como por exemplo o "deep-first". O JACK possui ainda um monitor visual dos resolvedores implementados com o JCHR, o VisualCHR, para

auxiliar o desenvolvimento dos resolvedores. A possibilidade de visualizar os efeitos da propagação facilita a definição das regras subjacentes à criação dos resolvedores.

JSolver "Java Solver" [Chu99] é uma biblioteca de restrições para utilização embebida nas aplicações Java. Admite domínios finitos e Booleanos. As operações usuais sobre um sistema de restrições são providenciadas pela classe `JSolver` através de métodos que permitem criar variáveis, vectores de variáveis ou restrições. Os vectores de variáveis permitem implementar as restrições sobre todas as variáveis do vector, tais como o "all-different" e restrições de soma ou cardinalidade. Estas últimas permitem definir relações de cardinalidade entre as variáveis do vector. Existem também variáveis "revertíveis" para ser possível recuperar o valor duma variável quando há necessidade de fazer backtracking. A pesquisa é realizada usando um array de variáveis: é escolhida uma variável para iterar e para essa variável é escolhido um dos seus possíveis valores que caso exista faz activar a propagação. Se há falha e não existe outro possível valor faz-se bactraking. A particularidade desta abordagem está nas variáveis revertíveis que no processo de bactracking permitem recuperar os valores das variáveis correspondentes aos pontos de escolha para onde se recua. A escolha de variáveis e de valores é controlda pelo uso de heurísticas que podem ser definidas pelo utilizador definindo os métodos apropriados nas respectivas classes.

JCL "Java Constraint Library" [TWF97] Trata-se duma livraria para o Java, que pode ser usada tanto em aplicações "stand-alone" como em "applets". Esta livraria tem implementado um vasto conjunto de algoritmos de pesquisa, tais como o BT (bactracking), Backjumping, Forward Chainning, ou ainda combinações destes algoritmos. É providenciada uma shell de restrições no topo da livraria, cujo intuito é a interação facilitada com a camada da livraria. Esta shell providencia uma forma amigável para criar, edita e salvaguardar os sistemas de restrições.

Os sistemas de restrições são pré-processados aplicando-lhes as técnicas de consistência de nó e arco de modo a reduzir a pesquisa. A pesquisa é

activada através da shell onde o utilizador seleciona qual dos algoritmos de pesquisa que pretende usar e o tipo de soluções pretendidas, se todas, uma ou um número específico.

DJ "Declarative Java" [NFZY98] Não se trata duma biblioteca de restrições para o Java mas duma extensão da sintaxe do Java, que mescla o Java com a programação por restrições. Foi desenvolvida para simplificar a construção GUI's e applets. O utilizador especifica as componenetes da interface gráfica e as relações entre elas, usando restrições. A disposição do componentes é automaticamente determinada pelo sistema de restrições. A definição O compilador de código DJ foi construído em B-Prolog. O compilador extrai do código DJ o sistema de restrições, resolve-o, activando o resolvedor e gera o programa Java e um ficheiro HTML correspondentes ao código DJ original e a solução obtida. A resolução do sistema de restrições está portanto a cargo do B-Prolog uma linguagem de Programação em Lógica com Restrições.

Flow Java[DSHB03] Pese embora não se tratar uma biblioteca de restrições para Java, o Flow Java é também uma extensão do Java que incorpora o mecanismo da programação declarativa no Java, através da introdução de varáveis de afectação única ("single assignment"). Estas variáveis são usadas para a sincronização e comunicação entre processos.

1.5 Plano

Os restantes capítulos desta tese seguem a ordem que seguidamente se especifica sendo apresentado um guia de leitura:

- Nos capítulos 2 e 3 são apresentadas as implementações, respectivamente, do GC e do AJACS sequencial.
- No capítulo 4 é apresentado o modelo formal do AJACS distribuído.
- No capítulo 5 são introduzidos alguns conceitos básicos sobre DSM's, modelos de consistência e o sistema Hyperion.

- No capítulo 6 é feita uma uma descrição da implementação em Java de dois modelos distribuídos do AJACS e a experimentação dos modelos num ambiente de memória partilhada e distribuída através da utilização do Hyperion.
- No capítulo 7 são descritas outras implementações distribuídas do AJACS, com base no conceito de estratégia.
- No capítulo 8 é apresentada uma ilustração do AJACS na resolução dum problema real, a construção de horários.
- No capítulo 9 são apresentadas as conclusões.

Capítulo 2

Resolução sequencial de CSPs num contexto OO

Introduzir o paradigma da programação por restrições na linguagem Java, extendendo a hierarquia de classes do Java de modo a contemplar os objectos que caracterizam um sistema de restrições, bem como as classes e objectos que implementam um resolvedor de domínios finitos, é o objectivo do Generic Constraint, uma biblioteca de restrições para Java.

2.1 Introdução

A programação por restrições (PR) [JL87] é um paradigma de programação geralmente usado para solucionar problemas de optimização combinatória. A formulação e métodos de resolução deste tipo de problemas, tem sido objecto de estudo em áreas da Matemática como a Investigação Operacional. São regra geral, problemas de resolução difícil, principalmente quando o número de variáveis do problema é elevado, sendo os algoritmos propostos pelas áreas referidas de aplicabilidade reduzida, salvo excepções e casos particulares que são tratados à margem do problema geral.

Muitos problemas reais (escalonamento, planeamento, etc), são facilmente formulados como problemas de optimização combinatória , e encontraram na PR em geral, e na programação em lógica (PL) em particular, métodos de formulação e resolução apropriados. É reconhecida à PL expressividade e adequação, para solucionar problemas combinatórios complexos. O reverso da medalha, é a difusão e a aceitação que o Prolog tem, fora dos meios académicos. Este aspecto, limita obviamente o uso destas técnicas onde elas são mais necessárias, i.e., em ambientes não especializados.

O desenvolvimento de aplicações em plataformas apelativas, parece ser o caminho a tomar para fomentar o uso de técnicas de aplicabilidade indiscutível. No campo teórico, a linguagem Java é proclamada como possuidora de uma independência de arquitectura, da máquina onde será executado o código, sendo portanto uma opção a considerar. Na prática as implementações existentes são um entrave, ainda que ligeiro, à tão proclamada independência. Esta desvantagem tenderá a ser minimizada pela emergência de melhores sistemas de apoio em tempo de execução ("run-time support systems"). Compiladores "just-in-time", como o TYA e as recentes versões do JDK, são disso um exemplo.

A ineficiência das implementações da máquina virtual(JVM), poderá parecer um impedimento ao desenvolvimento de outros paradigmas de programação, especialmente aqueles cuja eficiência é determinante. Contudo, no caso das Linguagens de Programação por Restrições, o incentivo de possuirmos uma plataforma, altamente difundida (afinal todos os browsers da Web, possuem incorporados

um sistema de tempo de execução para o Java) e binário-compatível, remete os entraves acima referidos para um plano secundário.

A situação aqui exposta foi já abordada, tendo neste caso a linguagem de programação por restrições Ilog:Solver [PL95], optado pelo C++. O nosso objectivo será construir um sistema análogo, mas melhorado, em Java. A eficiência do sistema, ou a falta dela, será um aspecto, que não consideraremos crucial, na fase inicial deste projecto, sendo o desenvolvimento dum enquadramento claro e transparente, para uma linguagem de programação por restrições, integrada com Java, o objectivo primordial desta abordagem.

Outra característica importante, e que abona em favor da opção tomada, são as ferramentas gráficas que o Java tem incorporadas, e que partilham da mesma portabilidade. Estas ferramentas gráficas, em especial as "Java Foundation Classes"ou o "Swing", permitir-nos-ão construir ambientes de programação sofisticados, e extender o nosso trabalho para outras áreas, como por exemplo a das Linguagens de Programação Visuais.

Neste capítulo a secção 2.2, descreve O GC, sendo que as secções seguintes descrevem com detalhe, algumas das características mais importantes do sistema.

2.2 O GC

Numa linguagem "Object Oriented" existem objectos que comunicam entre si através de mensagens. Os objectos são instâncias de uma única classe (no caso da linguagem não admitir polimorfismo), sendo definidas na classe as características dos objectos, vulgarmente designadas por atributos, ou no caso do Java variáveis de instância. A criação dum objecto criará uma instância dessa classe, sendo-lhe especificados os atributos que o particularizam. As classes estão realcionadas através duma hierarquia sendo que uma classe só poderá ter uma superclasse (se não existir polimorfismo), mas poderá ter mais do que uma subclasse. Uma subclasse herdará as características da sua superclasse e especificará um conjunto de particularidades.

De modo a obter um dialecto para uma linguagem orientada a objectos, que permita uma metodologia de programação por restrições, existem duas hierar-

quias de classe independentes, mas paralelas: A classe *Variable* e a classe *Constraint*. A figura 2.1 apresenta a hierarquia de classes do GC. A comunicação e sincronização destas duas classes, obtêm-se através de relações de referência. As restrições contêm variáveis (acessíveis pelo atributo de classe "environment", que especifica o ambiente da restrição) e as variáveis referem o conjunto de restrições nas quais ocorrem (pelo atributo de classe "dependencies", que especifica as dependências da variável). Por exemplo, se existirem as variáveis X, Y e Z e as restrições $C_1 : X + Y = Z$ e $C_2 : X \neq Y$, o ambiente de C_1 é definido pelas variáveis X, Y e Z, sendo o de C_2 caracterizado por X e Y. Por outro lado a variável X, tem o conjunto de dependências, $\{(C_1, Y), (C_1, Z), (C2, Y)\}$, Y, tem o conjunto , $\{(C_1, X), (C_1, Z), (C2, X)\}$ e Z as dependências $\{(C1, X), (C_1, Y)\}$.

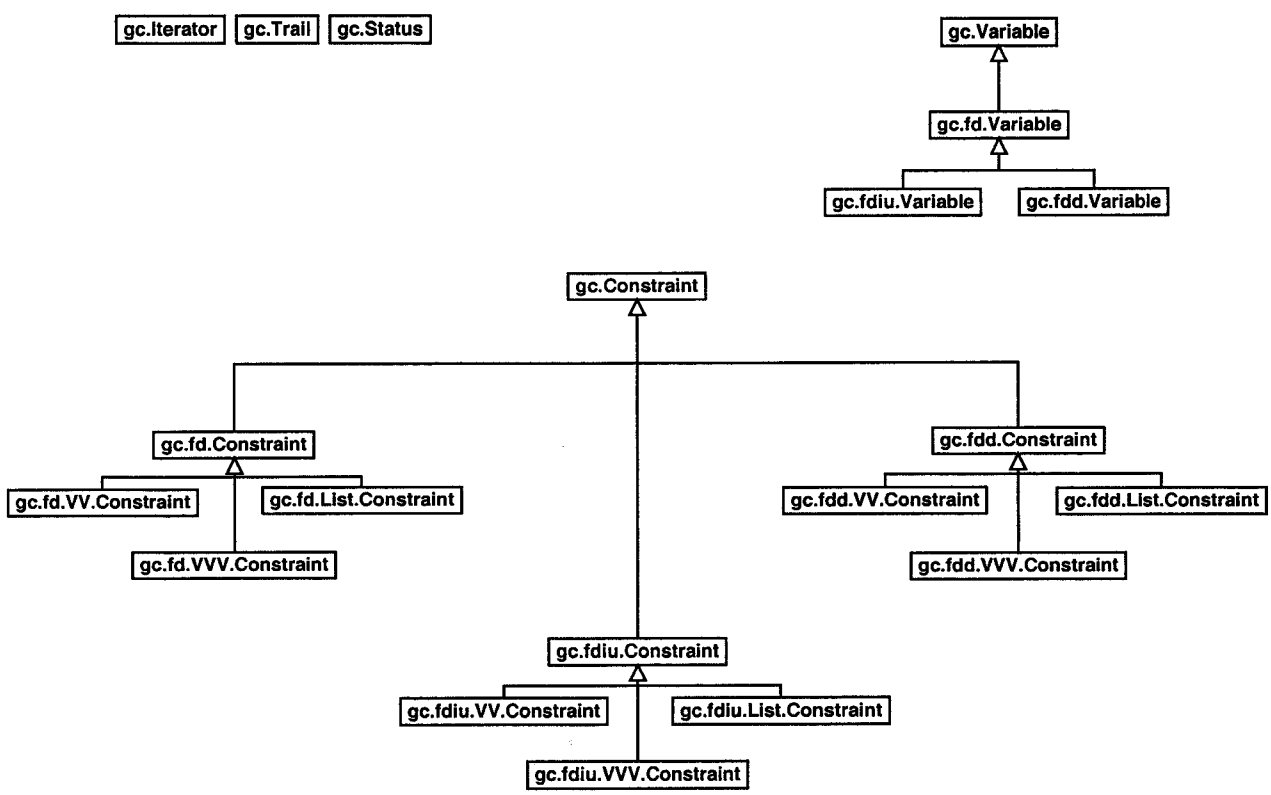


Figura 2.1: Hierarquia de classes do GC

As aplicações conterão instâncias de subclasses de Variable, Constraint e Ite-

rator (para forçar soluções de valor único). A arquitectura do GC, compreende três níveis. O nível do Núcleo, o nível das Restrições e o nível das Aplicações. Ao nível do núcleo estão implementadas todas as definições de base, necessárias ao funcionamento das restrições e das variáveis. No nível das restrições estão implementados domínios específicos (i.e. Domínios Finitos(FD), Booelanos, etc), e as restrições básicas sobre estes domínios ($<$,$\leq$, $=$,$\neq$, etc). Finalmente no nível das Aplicações, são utilizados os dois anteriores e podem ser especificadas novas classes de restrições, que sejam necessárias para solucionar o problema em mãos.

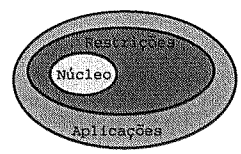


Figura 2.2: Níveis da arquitectura do GC

2.3 Variáveis (de domínio)

Um domínio é conjunto de valores. O domínio de uma variável é o conjunto de valores que esta pode assumir. Estão implementadas três classes distintas de variáveis, modelos da representação de variáveis sobre domínios finitos. As variáveis FD, têm associado um domínio que é representável por um único intervalo de valores inteiros positivos. Esta representação baseia-se na manipulação dos valores máximo e mínimo do domínio. As variáveis FDD, são variáveis FD, às quais é adicionado um mapa de bits, para representar os valores do domínio, e as variáveis FDIU, também variáveis FD, são variáveis cujos valores do domínio são representados por uma união disjunta de intervalos. Na hierarquia de classes das variáveis, ver figura 2.1, o modo de funcionamento genérico das variáveis é definido pela classe *gc.variable*. Esta classe implementa todo o processo respeitante às dependências, ficando o trabalho respeitante aos domínios para ser implementado nas subclasses correspondentes. Estas devem responder a mensa-

gens do tipo: se um domínio é unitário ou se está vazio, efectuar cópias ou clones das variáveis, obter o primeiro valor do domínio, saber qual o valor que se segue a outro pré–determinado, e construir domínios singulares, com determinado valor.

2.3.1 Variáveis FD

Estas variáveis são implementadas pela classe *gc.fd.Variable*, que é subclasse de *gc.Variable*. Desde que se assuma que os valores possíveis para a variável são representáveis por um domínio compacto, as variáveis FD resultam bem. Trabalhar com este tipo de variáveis, resume-se a manipular os extremos do domínio. As restrições sobre este tipo de variáveis avaliam somente os valores máximo e mínimo do domínio, e actualizam-nos convenientemente por forma a validar as restrições existentes sobre a variável.

2.3.2 Variáveis FDD

Nem sempre e possível representar o domínio de uma variável por um intervalo de valores, e por isso uma representação FDD é nestes casos necessária. Trata-se duma subclasse de *gc.Variable* à qual é acrescentado um mapa de bits, para representar os valores do domínio. A manipulação destes domínios é realizada recorrendo aos métodos *clear* e *put*, que eliminam e acrescentam, respectivamente, valores ao domínio. As restrições sobre este tipo de variáveis actualizam os valores do domínio, usando repetidamente estes métodos.

2.3.3 Variáveis FDIU

Do exposto até ao momento, é imediato que a representação duma variável cujo domínio tenha uma dimensão considerável, i.e., como muitos valores, mas que seja compacto, é realizada pelas variáveis FD, e que variáveis com domínios de dimensões mais reduzidas mas esparsos, são representados por variáveis FDD. A questão agora coloca-se em como representar domínios de dimensões consideráveis, que não sejam contíguos? Usar um mapa de bits para representar domínios extensos não é de todo apropriado. É sabido que um modo conden-

sado de representar uma série de valores contíguos, é através dum intervalo, uma vez que é suficiente conhecer-lhe os extremos. Uma união de intervalos surge assim como uma representação natural, e condensada, de domínios extensos e esparsos. As variáveis FDIU, usam esta representação no seu domínio de valores. A "package" *gd.fdiu.domain* implementa esta definição de domínio, e exporta métodos para adicionar e remover intervalos de um domínio nestas condições. Operações básicas sobre intervalos (e domínios), como a disjunção, intersecção, soma, diferença, etc, são também disponibilizadas pela "package".

2.4 Restrições

Sem dúvida, as restrições e o seu sistema de propagação, são o âmago do GC. Uma restrição é uma relação entre variáveis e eventualmente entidades externas ao sistema de propagação das restrições (i.e. constantes). A inclusão duma nova restrição num sistema de restrições irá criar novas dependências nas variáveis que intervêm nessa restrição. A imposição duma restrição ao sistema, induzirá, regra geral, uma diminuição dos valores do domínio das suas variáveis. Quando tal acontece, todas os domínios das variáveis dependentes da variável que foi reduzida, terão de ser analisados, e actualizados, caso seja necessário. As classes de restrições seguem uma hierarquia de heranças de classes. Novas restrições podem ser facilmente adicionadas ao sistema, acompanhadas necessariamente do apropriado método *localupdate(n)*. Este método é o responsável pela actualização do domínio da n-ésima variável, para a restrição em causa. É um método que é invocado por outro, o *update* (ver 4.3), e portanto a este nível não existe propagação, não sendo portanto necessário ter em conta outro tipo de actualizações para os domínios. A definição duma nova restrição é assim simplificada, bastando para isso enquadrá-la na hierarquia de restrições já existente e definir a forma de actualizar os domínios das variáveis da restrição, escrevendo o respectivo método.

2.5 Propagação

A progragação é o mecanismo que permite validar as restrições de domínio sofridas pelas variáveis do sistema (ver secções 1.3.1 e 4.3). Estas reduções ocorrerão ou pela iteração de variáveis ou pela determinação da consistência dos arcos associados às restrições. Juntamente com o uso de iteradores (ver secção 2.6) são o modo de construir as soluções dum sistema de restrições de domínios finitos. Na raiz da hierarquia de classes das restrições, está definido o método *update(n)*, responsável pelo mecanismo de propagação, que é iniciado no momento da imposição da restrição. A função deste método é actualizar a n-ésima variável da restrição. Se alguma alteração no correspondente domínio se verificar, todas as restrições sobre aquela variável serão reavaliadas, até ser alcançado um ponto fixo, ou seja, não existam mais alterações. A imposição duma restrição é realizada pelo método *tell* . Este método actualiza todas as variáveis da restrição, invocando o método *update*. Caso o *update* de alguma das variáveis modifique um domínio, o método *updateDeps* será invocado para actualizar as dependências da variável modificada. Este método será recursivamente invocado sempre que as actualizações efectuadas produzirem alterações.

2.5.1 Exemplo da actuação dos métodos da propagação

De modo a clarificar qual o papel de cada um dos métodos responsáveis por realizar a actualização dos valores das variáveis, e responsáveis pelo mecanismo de propagação considere-se o seguinte exemplo:

```
Variable X= new Variable(2,7);
Variable Y= new Variable(4,8);
Constraint c=new XplusYeqZ(theVariable[0],theVariable[1],theVariable[2]);
c1.tell();
```

A imposição da restrição, desencadeará a execução do método *update* para todas as variáveis da restrição. Ver figura 2.3. Após o *update* da variável X, e por que o domínio desta variável se alterou, são actualizadas as dependências desta variável, pelo método *updateDeps*. X depende de Y e 8, sendo chamados

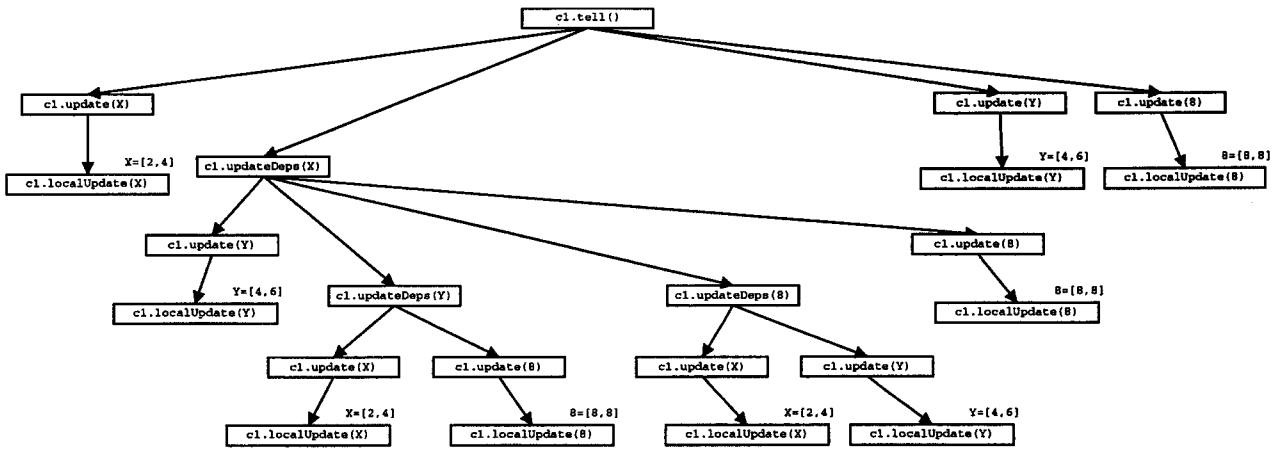


Figura 2.3: Imposição da restrição X+Y=8

os métodos *update* para cada uma destas variáveis. Como Y vê o seu domínio modificado, novas chamadas de *updateDeps* são efectuadas, *updateDeps(Y)* e *updateDeps(8)* são invocados pois são as variáveis que constituem o ambiente da restrição, e não são a variável cujas dependências estão a ser actualizadas, i.e. X. Nenhum destes métodos *updateDeps* produz novas alterações nos domínios das variáveis, e o processo termina com a execução dos métodos pedentes.

2.6 Iteradores

A propagação por si só não permite construir as soluções dum sistema de restrições, existindo necessidade de forçar instanciações singulares às variáveis do sistema e usar a propagação para validar essas instanciações. Uma a uma e após todas as variáveis serem iteradas com sucesso é possível obter as soluções do sistema de restrições. Caso a instanciação gere conflito, e a propagação resulte em falha há instanciar a variável com outro possível valor do seu domínio e tentar novamente. A geração de todas as soluções do problema é obtida forçando todas as possíveis instanciações para todas as variáveis. O objecto no *GC* responsável pela iteração das variáveis é o iterador.

O objectivo do uso de iteradores, é restringir um determinado conjunto de variáveis, forçando-as a assumir domínios singulares. As variáveis iteradas, serão

aquelas que forem adicionadas ao iterador. É necessária uma inicialização do iterador, de modo a prepará-lo para a obtenção da primeira solução. As soluções seguintes, são obtidas sequencialmente, por iteração de todos os valores possíveis, com a consequente activação do mecanismo de propagação. Tal é conseguido enviando uma mensagem à instância do iterador. As classes dos domínios, devem providenciar métodos que permitam seleccionar o primeiro valor dum domínio, e o valor seguinte a outro. Com o auxílio destes métodos, é possível realizar a iteração dum domínio.

A estratégia usada pelo iterador, por analogia com o Prolog, é o "Depth-first", "left-to-right", embora outro tipo de estratégias ("breath-first", "first-fail", etc) possam ser implementadas como subclasses do iterador. Uma estrutura de dados para pista ("trail") é usada, uma vez que o iterador recua("backtrack") em caso de falha. Esta estrutura, é constituída por uma cópia das variáveis, livres de dependências, que armazenam os domínios.

2.6.1 Exemplo

Defina-se $X \in [2..4] \cup [6..20]$ e $Y \in [1..7]$. Imponha-se a restrição $X < Y$. Os domínios admissíveis para as variáveis são dados agora por $X \in [2..4] \cup \{6\}, Y \in [3..7]$. A tabela da figura 2.4 mostra os resultados obtidos, quando usamos o iterador, sobre uma, ou ambas as variáveis.

Iteraçao	*# Soluções*	m*Soluções (X, Y)*
X	4	m$\{(2,[3..7])\ ;\ (3,[4..7])\ ;\ (4,[5..7])\ ;\ (6,\{7\})\}$
Y	5	m$\{(\{2\},3)\ ;\ ([2..3],4])\ ;\ ([2..4],5)\ ;\ ([2..4],6)\ ;\ ([2..4] \cup \{6\},7)\}$
X, Y	13	m$\{(2,3)\ ;\ (2,4)\ ;\ (2,5)\ ;\ (2,6)\ ;\ (2,7)\ ;\ (3,4)\ ;\ (3,5)\ ;\ (3,6)\ ;$ m $(3,7)\ ;\ (4,5)\ ;\ (4,6)\ ;\ (4,7)\ ;\ (6,7)\}$

Figura 2.4: Tabela exemplificativa do uso de iteradores

2.7 Exemplos

Apresentamos nesta secção três exemplos de utilização do GC. Os exemplos escolhidos são por demais conhecidos no âmbito da literatura da PR. As soluções implementadas seguem a abordagem sugerida em [Dia95].

- "The Five Houses Puzzle"

  - Problema:
    Existem 5 pessoas que moraram em casas vizinhas, na mesma rua. As suas nacionalidades, profissões, bebidas favoritas, e animais de estimação são todos distintos. Com base nas restrições do problema, determinar quem é quem.

  - As Variáveis:
    Existem 25 variáveis, a saber: os 5 nomes, as 5 nacionalidades, as 5 profissões, as 5 bebidas e os 5 animais. Definam-se os vectores de variáveis

$$\begin{aligned}
\texttt{N} &= [\ N_0,\ N_1,\ N_2,\ N_3,\ N_4\ ] \\
\texttt{C} &= [\ C_0,\ C_1,\ C_2,\ C_3,\ C_4\ ] \\
\texttt{P} &= [\ P_0,\ P_1,\ P_2,\ P_3,\ P_4\ ] \\
\texttt{A} &= [\ A_0,\ A_1,\ A_2,\ A_3,\ A_4\ ] \\
\texttt{D} &= [\ D_0,\ D_1,\ D_2,\ D_3,\ D_4\ ]
\end{aligned}$$

    sendo `N`, `C`, `P`, `A`, `D`, respectivamente, o vector dos nomes, nacionalidades, profissões, animais e bebidas. Todas as variáveis são inteiros do intervalo $[1, 5]$, devendo ser criadas usando o construtor de tipo, *Variable*, da forma:
    `Variable X=new Variable(1,5)`

  - As Restrições:
    As restrições do problema apresentam-se na tabela em abaixo, aparecendo agrupadas por tipo.

2 Variáveis	$N_5 = 1$; $D_5 = 3$; $N_1 = C_2$; $N_2 = A_1$ $D_5 = 3$; $N_3 = P_1$; $N_4 = D_3$; $P_3 = D_1$ $C_1 = D_4$; $P_5 = A_4$; $P_2 = C_3$
3 Variáveis	$C_1 = C_5 + 1$
Listas	allDifferent(N);allDifferent(C) ;allDifferent(P); allDifferent(A);allDifferent(D)
Disjunctivas	$A_3 = P_4 + 1 \vee A_3 = P_4 - 1$ $A_5 = P_2 + 1 \vee A_5 = P_2 - 1$ $A_5 = P_2 + 1 \vee A_5 = P_2 - 1$

As restrições da primeira linha, correspondentes a restrições sobre duas variáveis são instaladas por

`new EQ(N5,new Variable(1)).tell()` para a primeira restrição e

`new EQ(P2,C3).tell()` para a última.

A restrição da segunda linha é obtida fazendo

`new XplusYeqZ(C5,new Variable(1),C1).tell()`

As restrições correspondentes à terceira linha, podem ser instaladas por um dos seguintes processos:

1. Estabelecer que as n variáveis duma lista são todas distintas umas das outras, é equivalente a estabelecer todas as desigualdades, entre quaisquer duas variáveis dessa lista. Como `NE` ($\neq$), é uma restrição básica, o problema pode ser solucionado fazendo:

```
...
for (int i=0;i<5;i++)
   for (int j=i+1;j<5;j++)
       new NE(X[i],X[j]).tell();
...
```

2. Usando a restrição para listas, `allDifferent`. Esta restrição aceita como argumento um vector de variáveis, e instala todas as restrições que lhe estão associadas, de modo idêntico ao realizado no ponto 1, mas automaticamente. Neste caso, a instalação

das restrições é obtida por `new allDifferent(X).tell()`, para `X` um qualquer dos vectores de variáveis.

```
public class allDifferent extends gc.fdiu.List.Constraint{
public allDifferent (Variable List[]) {
    super(List);
}
public void localUpdate (int n) {
    int l=env.length;
    for (int i=0; i<l;i++){
        if (i!=n && env[i].ground()){
            env[n].clear (env[i].min);
            env[n].updateMinMax();
        }
    }
}
}
```

As restrições da quarta linha da tabela, são as disjuntivas. São restrições não básicas, e portanto é necessário implementá-las explicitamente. Tal como foi referido anteriormente, a implementação duma nova restrição, requer que seja definida uma nova classe, no local apropriado da hierarquia de classes das restrições, e definir o método, `localUpdate`, para a restrição em causa. Este método definirá o modo conveniente (em função da restrição em mãos) de actualizar qualquer das variáveis da restrição, por alteração do valor das restantes .

Por exemplo, no caso da restrição `EQ` e para variáveis `FDIU`, se ambos os domínios são forçados a partilhar os mesmos valores, basta calcular a intersecção dos domínios, para actualizar as variáveis. A actualização dos valores máximo e mínimo dos domínios é também necessário, visto estas variáveis serem uma subclasse (especialização) de `FD`.

Para implementar a restrição disjuntiva `plusOrMinus`, é necessário

definir uma subclasse de gc.fdiu.VV.constraint.

Esta subclasse define uma variável de instância `value`, usada no `localUpadte` para criar um domínio singular. `localUpadte` actualiza as variáveis fazendo uso da relação:

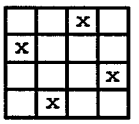
$domain(X) = domain(X) \cap [[domain(Y) - \{value\}] \cup [domain(Y) + \{value\}]$

```
public class plusOrMinus extends gc.fdiu.VV.Constraint{
    private int value;
    public plusOrMinus (Variable VX, Variable VY, int V) {
        super(VX, VY);
        value=V;
    }
    public void localUpdate (int n) {
        int m=1-n;     // env[n]=env[m]-i  V env[n]=env[m]+i
        Domain Dn=env[n].domain;
        Domain Dm=env[m].domain;
        Domain Di=new Domain(value);

        Domain Ddiff=Dm.diff(Di);
        Domain Dsum=Dm.sum(Di);
        env[n].domain=Dn.intercept(Ddiff.union(Dsum));
        env[n].updateMinMax();

    }
}
```

- "N Queens"
  - Problema:
    Dispor `N` rainhas num tabuleiro de xadrez, de modo a que nenhuma rainha ameace qualquer das outras.
  - Variáveis:



Temos `N` variáveis que assumem os valores de `1...N`. Por exemplo para $N = 4$, a solução dada por `[3,1,4,2]`, significa que a rainha da primeira linha deverá estar na terceira coluna, a da segunda linha na primeira coluna, etc (ver figura).

– As restrições:

De modo a garantir que na disposição eleita, as rainhas não se ataquem, temos de implementar a restrição correspondente a esse ataque. Se C_i e C_j, são rainhas colocadas nas colunas i e j respectivamente, temos que garantir que:

$$\forall j > i \begin{cases} C_j \neq C_i \\ C_j \neq C_{i+(j-i)} \\ C_j \neq C_{i-(j-i)} \end{cases}$$

A restrição de duas variáveis `NoAttack`, implementa estas limitações.

```
public class NoAttack extends gc.fdiu.VV.Constraint{
private int c;
public NoAttack(Variable VX, Variable VY, int V) {
    super(VX,VY);
    c=V;
};
public void localUpdate(int n) {
    int m=1-n;
    if (env[m].ground ()) {
        env[n].clear (env[m].min);
        env[n].clear (env[m].min + c);
        env[n].clear (env[m].min - c);
    }
    env[n].updateMinMax ();
 }
}
```

Dado que `NoAttack`, é uma restrição sobre duas variáveis, é necessário um método para aplicar as restrições implícitas a todas as rainhas de uma lista:

```
static void safe (Variable queen[], int n) {
    for (int i=0; i<n; ++i)
        for (int j=i+1; j<n; ++j)
            new NoAttack(queen[i], queen[j], j-i).tell();
}
```

- "Send More Money"
  - Problema:
    Determinar uma correspondência injectiva dígito-letra, de modo que a operação da figura seja válida.

```
  S E N D
+ M O R E
---------
M O N E Y
```

  - Variáveis:
    Temos 8 variáveis, `S`, `E`, `N`, `D`, `M`, `O`, `R` e `Y`, todas distintas. Todas excepto `M`, tomam valores de 0 to 9. `M` assume valores de 1 a 9. Na construção da solução serão necessárias outras variáveis, dependendo o seu número da abordagem efectuada.

- Restrições:
  Duas abordagens são possíveis, para determinar a solução do problema.

  1. A solução pode ser encontrada resolvendo a equação:
     $1000S+100E+10N+D+1000M+100O+10R+E = 10000M+1000O+100N+10E+Y$
     Para tal, utilizaremos as restrições básicas `XtimesYeqZ` ($X \times Y = Z$) , `XplusYeqZ` ($X + Y = Z$) e algumas variáveis auxiliares. São necessárias muitas variáveis, muitas restrições (do tipo`XtimesYeqZ` e `XplusYeqZ`) e muito tempo também. A determinação da solução por este processo é morosa e ineficiente, mas possível.
  2. É preferível implementar uma restrição, `FullAdder`, que some três dígitos, e gere o dígito resultante e o correspondente transporte.

É uma restrição sobre 5 variáveis, sendo os argumentos da restrição, respectivamente, os três dígitos a somar, o resultado e o transporte. Com esta restrição, o seguinte sistema de restrições é suficiente para solucionar o problema:

```
new AllDifferent(V);
new FullAdder(new Variable(0),D,E,Y,C1);
new FullAdder(C1,N,R,E,C2);
new FullAdder(C2,E,O,N,C3);
new FullAdder(C3,S,M,O,C4);
new Eq(C4,M);
```

2.8 Análise Experimental

Na tabela seguinte, apresentamos os tempos obtidos para os exemplos descritos no capítulo anterior.

Program	GC (FDIU)	GC (FDD)	GC (FD)
Five	0m0.66s	0m0.81s	0m0.71s
Queens 4	0m0.38s	0m0.37s	0m0.35s
Queens 8	0m0.73s	0m0.67s	0m0.55s
Queens 16	1m18.07s	1m1.41s	1m2.68s
Send	0m1.16s	0m1.22s	0m2.65s

No programa "Send", a segunda abordagem foi a usada. Os tempos para as "Queens", são medidos no momento em que é encontrada a primeira solução. Os programas "Five"e "Send", usam a restrição *allDifferent* . Os tempos foram obtidos num AMD k6/300 em ambiente Linux, com o JDK 1.1.7 standard, i.e., não foi usado nenhum compilador "just-in-time". Uma opção do Java que inibe o "garbage collection"assincrono, foi usada.

Os tempos obtidos são bastante similares, em todas as implementações. A restrição `NoAttack` é a única cuja implementação é igual para todas as variáveis.

Por inerência da restrição pretendemos remover um valor do domínio, e portanto nenhum método particular de manipulação de domínios FDIU foi usado. Note-se que, nenhum dos exemplos apresentados evidencia as motivações que nos levaram a implementar os domínios FDIU, visto em todos os casos a extensão dos domínios ser reduzida. A definição de restrições sobre variáveis FD, só é possível(na maioria dos casos), quando temos domínios singulares. Tal significa que os domínios não vão sendo reduzidos gradualmente, ficando adiada a validação dos valores dos domínios com as restrições do problema até ao momento em que, por utilização do iterador, os valores sejam singulares. Quando a dimensão do problema aumenta os tempos de execução são claramente piores. As variáveis FD deverão ser usadas quando o leque de valores a representar é vasto e quando é necessária a realização de operações sobre estes conjuntos de valores (interceptar, unir, etc), sendo que quando a cardinalidade das variáveis é pequena, varáveis do tipo FDD providenciam um melhor desempenho. A utilização de variáveis FD tem utililidade para a sua criação, dado terem um representação compacta.

2.9 Conclusões

A utilização do *GC* como solver de restrições apresenta uma total integração com os programas Java por o seu uso ser realizado recorrendo à biblioteca que o implementa. A utilização de bibliotecas nos programas Java é um método vulgarizado para os seus utilizadores, não introduzindo o *GC* qualquer particularidade adicional neste caso. Claramente a eficiência do solver tem de ser contextualizada com o ambiente em que o mesmo é integrado. Atendendo a que a linguagem Java é interpertada, e apesar de alguma optimização de código poder ser feita usando técnicas de compilação e geração de código especiais (por exemplo pelo uso compiladores *just-in-time* mais eficientes) o desempenho do Java não é comparável à de linguagens como por exemplo o *C*. O próprio Java tende a relativizar a sua falta de performance, salientado que se trata duma linguagem para aplicações distribuídas sobre a internetA utilização de Threads que o Java incorpora permite a execução de processos leves (*light weigth processes*) e realizar pequenas tarefas em simultâneo. Será possível paralelizar a pesquisa de soluções dum sis-

tema de restrições usando os threads do Java? Se sim teremos certamente o problema da eficiencia dum solver de srestrições em Java ultrapassado ou pelo menos minimizado.

Capítulo 3

Proposta para uma organização OO com múltiplos estados independentes

Uma implementação diferente para uma biblioteca de restrições, que se caracteriza por uma estrutura de múltiplos estados independentes: o Another JAva Constraint System, é apresentado neste capítulo. A opção de iniciar uma nova aplicação e não reescrever a antiga, o GC, permitiu além introduzir os melhoramentos pretendidos abrir outras "portas" e redireccionar este trabalho para outros ângulos.

3.1 Introdução

O processo de pesquisa de soluções herdado da programação lógica e usado pelo GC, requer que os diferentes valores, que todas as variáveis do problema vão assumindo ao longo das sucessivas tentativas para encontrar uma solução, sejam armazenados, para, em caso de falha, ser possível recuar e restabelecer um contexto, que permita prosseguir, expandindo noutras direcções a árvore de pesquisa. O processo de "bactracking" inerente às linguagens de programação lógica, não existe no JAVA, sendo portanto necessário um esforço adicional, por parte do sistema, para simular esse comportamento. O GC apresenta alguns problemas que pretendemos solucionar, a saber:

- A manutenção dum rasto ("trail") de valores de variáveis, agravada pela ineficiência dos compiladores de Java, bem como pelas implementações em tempo de execução, não são de todo alheios aos fracos resultados apresentados pelo sistema.

- As estruturas de dados usadas pelo GC, não o tornam apropriado para evoluir para uma implementação paralela. Este facto limita as nossas pretensões de construir uma linguagem concorrente e distribuída.

- Estão implementadas no GC várias representações de domínios de variáveis, mas não é providenciado um modo de transição automático entre as diferentes representações, sendo necessário o utilizador escolher à priori o tipo de variáveis pretendendido, que será mantido durante a execução dos programas.

Modificar a estrutura do *GC* para contemplar as alterações pretendidas é certamente um mau príncipio, motivo pelo qual decidimos criar uma nova livraria por Restrições Concorrente, num enquadramento Java. Os nossos objectivos são agora:

- Automatizar a transição entre as diferentes representações de domínios, de modo a que a representação eleita seja sempre a mais apropriada. Esta automatização liberta o utilizador da tarefa de escolher a representação

inicial e permite que em função das operações realizadas pelo sistema um domínio mude de representação.

- Providenciar uma plataforma de programação facilitada, para uma cenário de execuções paralela e concorrente.
- Eliminar completamente o "trailling". Este objectivo poderá ser contemplado pela alínea anterior, já que o intuito do "trailing" é recuperar o valor antigo duma variável, tratando-se dum processo indesejável num cenário de paralelismo, visto implicar comunicações extra e reduzir a independência dos processos paralelos que queremos o mais autónomos possíveis.
- Providenciar um enquadramento, onde, diferentes estratégias de pesquisa possam ser facilmente especificadas pelo utilizador, ao mesmo tempo que o sistema fornece implementações credíveis para as estratégias mais comummente usadas.

3.2 Conceitos

Nesta secção iremos caracterizar as entidades que constituem o *Ajacs* e o modo como os mesmo se integram na caracterização dum CSP. Convenções notacionais: Um só objecto é designado por uma letra minúscula, um conjunto de objectos é representado pela respectiva letra maiúscula.

3.2.1 Valores

Um valor representa um subconjunto do domínio duma variável, i.e, o conjunto de elementos que são designados por valores singulares. Um valor diz-se básico ("ground") se contem exactamente um valor singular. Denotaremos pela u os valores, e pela letra U uma colecção de valores.

3.2.2 Variáveis

Não existe explicitamente o conceito de variável no Ajacs. Podemos concebê-las abstractamente como o conjunto de valores localizados no mesmo índice numa

sequência de estados ("stores").

3.2.3 Estados

Um estado é uma colecção indexada de valores. O objectivo é que durante a resolução dum problema de restrições, sejam criados vários estados similares (no que diz respeito ao número de valores) em que os valores associados ao mesmo índice nos diferentes estados , representem uma variável.

Um estado obtém-se de outro por propagação, contendo por isso uma referência ao estado anterior. Para o caso particular do estado inicial, o estado anterior é indefinido. A cada linha[1] de um estado está associada uma variável específica.

Além do mais, e por que um estado obtém-se do anterior, por contrição do valor de determinada variável, cada estado refere qual o índice da *linha* que foi restringida, para obtenção do próximo estado.

A definição de um estado s é dada por:

$$s \equiv (U, i, s_p)$$

Onde:

- $U = (u_1, u_2, \ldots, u_n)$, sendo u_i o valor actual da i-ésima linha do estado.
- i é o índice de U correspondente à variável que será restringida nos estados seguintes ao estado actual s, (i.e. todos os estados s' tais que $s' \equiv (U', i', s'_p) \wedge s'_p \equiv s$.
- s_p é o estado anterior de s.

3.2.4 Restrições

As restrições são relações entre as variáveis que ocorrem no problema. O conceito de restrição no Ajacs, espera que estas sejam o mecanismo responsável para

[1]Usaremos a expressão "linha" como sinónimo de índice do estado.

propagar às restantes variáveis do estado, as modificações efectuadas numa das variáveis. Uma restrição é definida por:

$$c \equiv (f, (i_1, i_2 \cdots i_n))$$

Onde $f = \lambda x_1 x_2 \ldots x_n \cdot e$ é uma função lógica sobre as variáveis do estado. Esta função transforma $(i_1, i_2 \cdots i_n)$ em *Verdadeiro* sempre que a restrição é válida para os valores correspondentes às linhas $i_1, i_2 \cdots i_n$ e *falso* sempre que o estado resultante é inconsistente.

O tuplo $(i_1, i_2 \cdots i_n)$ é designado por *ambiente* da restrição.

3.2.5 Problema

A modelação dum CSP, é realizada através do conceito de problema. Este, define um conjunto de variáveis, às quais está associado um domínio inicial (i.e. um estado), conjuntamente com um conjunto de restrições sobre essas mesmas variáveis.

O objectivo da formulação dum problema é determinar-lhe as soluções, i.e. o conjunto de valores básicos para todas as variáveis, que seja consistente com as restrições impostas.

No Ajacs, as soluções são obtidas de um estado, e portanto estes pertencem à nossa formulação do problema. Um problema p é definido como:

$$p \equiv (s_{init}, C, C_v)$$

onde:

- $s_{init} = (U, -, -)$ é o *estado inicial.*
- $C = \{c_1, c_2 \cdots c_k\}$ é o conjunto de restrições do problema que pretendemos solucionar. Estas, dizem respeito às linhas do estado s_{init}.
- $C_v = \{(i, C'_i) \ / \ \forall i \leq \#U,\ C'_i = \{(c_j, k) \ / \ c_j = (f, (i_{l_1}, \ldots, i_{l_n}) \in C \wedge i_{l_k} = i\}\}$. Informalmente, C_v é o conjunto de pares formados por uma linha (que representa uma variável) e o conjunto de restrições nas quais a variável

entrevem. O último conjunto é formado por uma restrição específica, e a ordem da variável no conjunto das variáveis da restrição. A intenção desta componente é indicar o conjunto de restrições nas quais uma variável ocorre.

O estado inicial do problema é obtido dos valores iniciais das variáveis. Todos os estados dum problema, têm a mesma estrutura.

3.2.6 Pesquisa e Estratégias de Pesquisa

O conceito de pesquisa incorpora o procedimento que permite calcular as soluções do problema, definindo o espaço de soluções e o modo como o mesmo é percorrido. A ordem pela qual as variáveis são instanciadas define uma configuração diferente do espaço de soluções se a mesma for alterada. Sendo o espaço de soluções uma árvore em que as folhas são as soluções do problema, árvores diferentes corresponderão eventualmente a alturas diferentes sendo consequentemente diferente o tempo para calcular uma solução (ver 1.3.2).

Um procedimento de pesquisa pode ser visto como a aplicação repetitiva do *passo de pesquisa*, até que seja encontrada uma solução ou concluir que a mesma não existe por exaustão do espaço de soluções. De modo a fazer cumprir os objectivos a que nos propusemos e sermos tão flexíveis quanto possível, os procedimentos podem ser sequenciais ou paralelos, independentemente do *passo de pesquisa.*

O *passo da pesquisa* pode ser definido como a acção concreta estipulada pela estratégia de pesquisa (ou simplesmente estratégia). A estratégia aplica-se a um estado computacional e especifica o próximo estado deste processo. Determinar o próximo estado implica decidir:

- Qual das variáveis não-básicas [2] será seleccionada, e:
- Para a variável seleccionada, a forma como será efectuada a redução de domínio. Regra geral, tal significará determinar o valor singular (dentre os possíveis) que será o escolhido para afectar a variável.

[2]Variáveis básicas possuem domínios singulares logo não são susceptíveis de ver reduzido o seu domínio.

De modo a satisfazer estes objectivos uma estratégia e é definida pelo par:

$$e \equiv (f_v, f_u)$$

Sendo f_v a função que selecciona a variável de s que será modificada e f_u a função que selecciona o valor particular que tal variável irá assumir, no novo estado.

Estas funções são específicas a cada estratégia e devem ser definidas do seguinte modo:

$$f_v: \quad s \mapsto i$$
$$f_u: (s, i, j) \mapsto u$$

Onde i é uma variável (índice) do estado s, u um valor básico e j um número de ordem cuja interpretação é dada pelas instâncias particulares da estratégia (i.e. f_v e f_u).

Por exemplo a estratégia $e_0 \equiv (f_v, f_u)$ para utilizar em pesquisas "'depth-dirst" e "breadth-first", que selecciona uma qualquer variável não-básica do estado e itera sobre todos os valores singulares pode ser dada por:

$$f_v(s) = i \;/\; U = \text{vars}(s) \wedge \#U_i > 1$$
$$f_u(s, i, j) = u \;/\; u = \{x\} \wedge x = \text{nth}(U_i, j)$$

Onde, *vars* é a função que extrai as variáveis do estado e *nth(u,j)* retorna o j-ésimo valor singular do domínio especificado por u. A título de exemplo se $u = \{4, 5, 6, 7\}, \text{nth}(u, 3) = 6$.

Um algoritmo para um procedimento de pesquisa sequencial que retorna a primeira solução encontrada, é dado por $sequencial\_search\_first(s, C)$, onde s é um estado e C um sistema de restrições. O algoritmo está ilustrado na figura 3.1. A versão paralela deste procedimento, pode ser facilmente obtida, substituindo o ciclo *foreach* por um ciclo paralelizado, efectuando cada um dos agentes que efectuam a pesquisa em paralelo as instruções do corpo do ciclo.

O resultado final dum procedimento de pesquisa dum problema, é o conjunto de soluções desse problema. Uma solução é simplesmente um estado, com a mesma estrutura do inicial, com a particularidade de serem básicas todas as suas variáveis.

Uma estratégia define o método de construção (dinâmica) do espaço de soluções definindo a pesquisa o modo como se precorre esse espaço em busca das soluções.

```
sequential_search_first(s, C)
IF < s is ground >
THEN RETURN s
ELSE let U = vars(s);
   i = f_v(s);
   FOREACH j in U_i DO
      s' = f_u(s, i, j);
      propagate(s', C);
      IF < s' is consistent >
      THEN RETURN sequential_search_first(s', C)
```

Figura 3.1: Algoritmo para um procedimento de pesquisa sequencial

A quantidade de estratégias que podem ser implementadas é vasta, por exemplo uma estratégia que implemente a heurística *first-fail* pode ser dada pelas estratégias cuja função de selecção da variável seja $f_v(s) = i \ / \ U = \text{vars}(s) \wedge \#U_i > 1 \wedge \forall j \ \#U_i \leq \#U_j$, e a função de selecção de valor uma qualquer das funções deste tipo.

3.3 Classes Java

A implementação do Ajacs num enquadramento Java, é obtida pela hierarquia de classes que seguidamente apresentamos:

3.3.1 Classe *Value*

A interpretação de um *valor* é o conjunto de inteiros que uma variável pode assumir. Internamente existem três representações distintas de *valor*, uma de conjuntos compactos, as outras de conjuntos esparsos. A classe *Value* implementa intervalos de valores compactos, a classe *FddValue* implementa intervalos não-compactos como vectores de bits e a classe *FdiuValue* implementa intervalos não-compactos como a união de intervalos compactos disjuntos.

A transição entre as diferentes representações do tipo valor é efectuada pelo

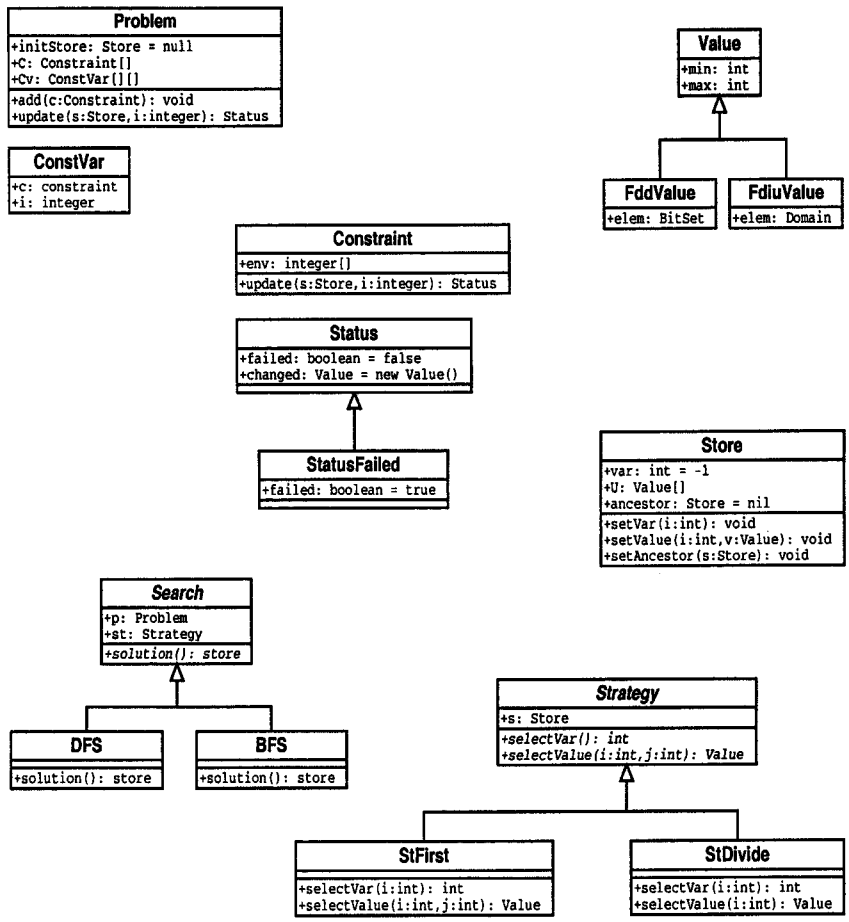


Figura 3.2: Sistema de Classes do Ajacs

sistema com base nas características apresentadas pelas representações. As restrições afectam os valores das variáveis realizando operações sobre estes. Remover um inteiro dum *valor*, interceptar, unir, ou somar valores, são algumas das acções passíveis de ser realizadas pelas restrições para gerar novos valores. A representação apropriada para estes novos valores é então decidida pelo sistema. Tal significa que uma operação realizada sobre objectos de uma das classes de *valor* pode retornar um objecto de outra das classes de *valor*. Um objecto da classe *value* pode sempre ser representado por um objecto de *FddValue* ou *FdiuValue*. O inverso nem sempre é possível.

A Representação Apropriada de um Valor

Sempre que tal seja possível as representações compactas são sempre preferidas. A figura 3.3 expõe todas as transições possíveis entre as diferentes representações. As transições são realizadas da seguinte forma: Um objecto da classe *Value* per-

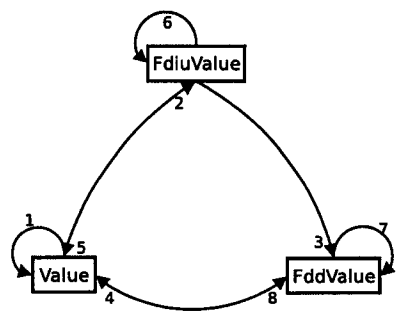


Figura 3.3: Transições nas classes de *Value*

manece nesta classe se a operação manipula somente os extremos do intervalo(1), caso contrário será convertido num objecto de *FdiuValue*(2), ou *FddValue*(8) dependendendo da cardinalidade dos valores. Um *FdiuValue* será convertido num objecto de *Value* se a união de intervalos é constituída por um só intervalo(5) e será convertido num *Fddvalue* se os intervalos da união são constituídos por um só inteiro(3). Nos restantes casos o objecto da classe *FdiuValue* permanece na sua classe(6). Um *Fddvalue* será transformado num *Value* se os elementos do conjunto são contíguos(4), permanecendo um *Fddvalue* nos restantes casos(7). A tabela em baixo exemplifica algumas das transições de representações de valores:

Representação	Operação	Representação	Transição
[1-10]	clear(10)	[1-9]	1
[1-9]	clear(5)	{1,2,3,4,6,7,8,9}	8
[1,20]	clear(10)	{[1-9] U [11-20]}	2
{[9-9] U [11-25]}	clear(9)	[11-25]	5
{1,2,3,5,6,7,...,21}	set(4)	[1-21]	4
{[1-1] U [3-3] U ...U [38-39]}	clear(38)	{1,3,5,..,39}	3

3.3.2 Classe *Store*

A implementação Java da classe "store", segue a sua discrição formal. Um construtor é usado para criar um novo estado, com base numa colecção de valores. O método *setVar* permite afectar a variável de instância *var*, correspondente à variável que será modificada pela pesquisa. O método *setValue* permite afectar o valor duma variável e *setAncestor* cria uma referência para o estado anterior, fazendo a ligação entre os dois estados.

3.3.3 Classe *Constraint*

A classe "Constraint", que implementa as restrições, possui uma variável de instância *env*, que armazena o ambiente da restrição. Este armazenamento é feito pelo método construtor da classe. *update(s,i)* irá actualizar todas as variáveis de $env[k], para\ k \neq i$. O objecto retornado pelo método é do tipo *Status*, podendo ser falha ou um valor. Neste último caso o valor retornado corresponde aos índices das variáveis que foram actualizadas, i.e. cujo valor foi efectivamente modificado. Sempre que a um valor i sofre alteração, seja por ser forçada uma instanciação seja por propagação, o método update(s,i) é chamado.

3.3.4 Classe *Problem*

A classe "Problem" é implementada com três variáveis de instância: *initStore*, correspondente ao estado inicial, C a lista das restrições e Cv uma lista de listas de objectos do tipo *ConstVar*. O construtor *new*, é usado para criar novos problemas, com base num tuplo de valores, que constituirão o estado inicial. As restrições são adicionadas ao problema através do método *add*, que actualiza as listas C e Cv. O método *update* será usado para propagar os efeitos de afectar um novo valor à i-ésima variável do estado. A propagação é obtida usando os pares (c, n) de C_{v_i}, e invocando *c.update(s,n)*.

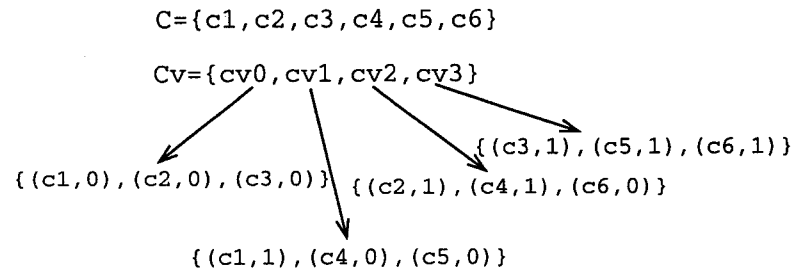


Figura 3.4: Listas, das restrições do problema: C e das restrições das variáveis: C_v.

3.3.5 Classes *Search* e *Strategy*

A classe "Search" é abstracta, uma vez que o método *solution* o é. Todas as subclasses de "Search", como as exemplificadas na figura 3.2 por *DFS* e *BFS* devem redefinir o método, cuja responsabilidade é gerar uma sequência de estados que conduza à solução ou a uma falha. A estratégia usada é a definida em *st*.

A classe "Strategy" é também abstracta, devendo todas as estratégias implementadas ser subclasses desta classe. Na figura 3.2 temos como exemplo de estratégias *StFirst* e *StDivide*. *Stfirst* é a estratégia dada como exemplo em 3.2.6 e *StDivide* a estratégia que resulta da função de selecção de valor, genérica, dada na equação 3.1, que particiona um domínio em N subdomínios, mas particularizando para $N = 2$. Na equação, U_i é o valor no índice i do estado s, sendo u_k é o seu k-ésimo valor.

$$f_u(s,i,j) = \begin{cases} \{u_{(j-1)a+1}, \ldots, u_{ja}\} & j = 1, \ldots, N-1, a = \lfloor \#U_i/N \rfloor \\ \{u_{a(j-1)a}, \ldots u_{\#U_i}\} & j = N \end{cases} \quad (3.1)$$

3.3.6 Exemplo

Um exemplo de como podemos usar este sistema de classes para resolver um problema de restrições é apresentado nesta secção. O clássico N-Queens é o exemplo escolhido, tomando para N o valor 4 por uma questão de simplicidade.

Temos 4 *valores*: `u0`, `u1`, `u2` e `u3`, definidos `u0 = u1 = u2 = u3 = new Value(1,4)`. É necessário implementar a restrição, *NoAttack*. Qualquer restrição implementada pelo utilizador, deverá ser definida como uma subclasse de *Constraint*. *No-*

Attack, à semelhança do exemplo da secção 2.7 do capítulo 2, será a restrição que assegura que as rainhas dispostas no tabuleiro não se ataquem umas às outras. O problema é definido dando o vector de valores que constituem o estado inicial, `p = new Problem([u0,u1,u2,u3])`. De modo a actualizar as listas C e Cv, as restrições são adicionadas ao problema, fazendo:

```
for (i=0; i<=2; ++i)
   for (j=i+1; j<=3; ++j)
        p.add (new NoAttack (i,j,j-i))
```

Considerem-se as seguintes denominações para as restrições:

$c1 \leftarrow NoAttack(0,1,1)$

$c2 \leftarrow NoAttack(0,2,2)$

$c3 \leftarrow NoAttack(0,3,3)$

... ...

Os ambientes das restrições estão definidos por:

$env(c1) = \{0,1\}$

$env(c2) = \{0,2\}$

$env(c3) = \{0,3\}$

... ...

Adicionar as restrições a `p`, retorna para C e Cv os valores da figura 3.4. Agora, estamos em condições de aplicar uma pesquisa ao problema com a instrução `s=new Search(p,new StFirst())`. A solução é obtida desencadeando o processo de pesquisa com `s.solution()`.

A figura 3.5 mostra uma sequência de estados gerada pelo método *solution* até ser obtida a primeira solução.

3.4 Pesquisa

Criar um "solver"de restrições, que pudesse ser tão versátil quanto possível em matéria de pesquisa, era um dos nossos principais objectivos na construção do Ajacs. Esta versatilidade foi alcançada de duas formas:

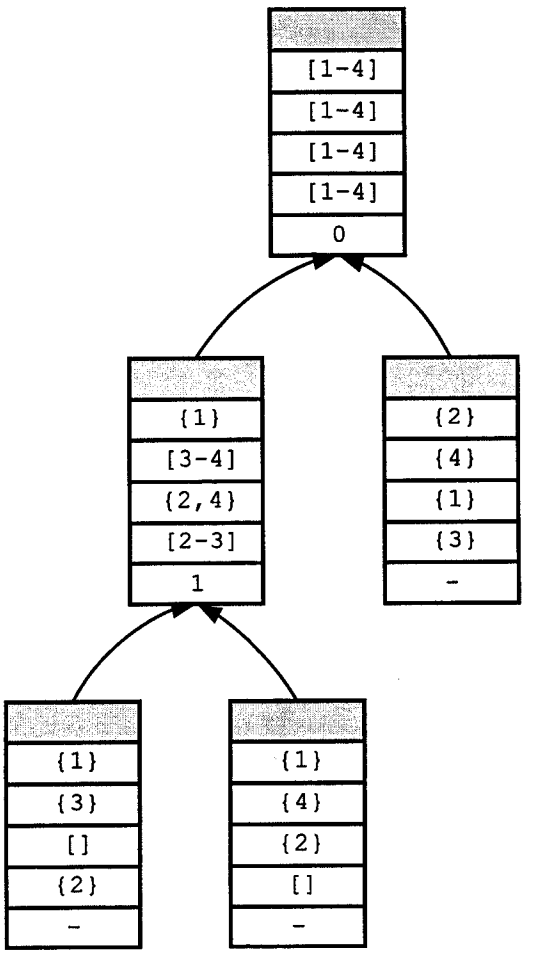


Figura 3.5: Algoritmo de pesquisa DFS aplicado ao problema 4-Queens

- Em qualquer fase da pesquisa, é possível saber qual o estado das variáveis, essa informação está presente no "estado".

- Existe uma total liberdade para definir (e usar) qualquer tipo de pesquisa, implementado a subclasse de "Search" apropriada. Esta liberdade é igualmente extensível à estratégia. A combinação entre uma estratégia apropriada e a geração de próximo estado, permite implementar uma variedade de procedimentos de pesquisa. "Depth-Dirst" "Breadth-First" "Best-first" são alguns exemplos. A melhor estratégia (i.e a mais apropriada) depende do problema em mãos. É inclusive possível que em determinada pesquisa a solução não seja um conjunto de variáveis básicas.

3.5 Conclusões

O Ajacs é a continuação do trabalho que iniciamos com o Gc, e enquadra-se obviamente no mesmo contexto. No entanto, relativamente ao Gc, distingue-se por:

- O "trail" de variáveis foi totalmente erradicado, o que era indubitavelmente uma operação de custos excessivos no âmbito de variáveis associadas a restrições.
- O sistema está organizado de tal forma que torna concebível pensar, sem alterações de fundo, em execuções paralelas.

A implementação de solvers de restrições como bibliotecas de linguagens de difusão alargada é uma ideia que apesar de não ser totalmente inovadora (o sistema *ILOG* [] é disso um paradigma), tem no Ajacs a particularide da linguagem de base ser o Java e não o C ou C++. Uma das características inovadoras no Ajacs, é o seu modelo computacional baseado em múltiplos estados. Este modelo cujo detalhe será o objecto do capítulo 4, permite uma representação da informação respeitante ao processo de pesquisa particularmente adaptável a modelos de execução paralela.

O paralelismo num sistema de restrições poderá ser obtido, ou paralelizando os algoritmos referentes à consistência, ou paralelizando a pesquisa. Algoritmos paralelos para a determinação da consistência de arco derivados de AC4, que tal como foi demonstrado por Mohr e Henderson em [MH86] é o mais eficiente algoritmo para o caso uni-processador, estão tratados em [CS94], no entanto uma estrutura que permita execuções sequenciais e simultâneamente adaptáveis a execuções paralelas pode ser encontrada no Ajacs.

Algumas das abordagens do Ajacs a assuntos como por exemplo a pesquisa ou mesmo o paralelismo podem encontrar num ou noutro sistema existente alguma similaridade (por exemplo no *OZ* ou no *OPL*), no entanto em nehuma destas implementações existe uma plataforma em que utlizadores menos dedicados à área da programação por restrições possam facilmente explorar as potenciallidades ofericidas. Isto advém do facto do Ajacs ser uma package do Java.

Capítulo 4

O modelo computacional distribuido do AJACS

Apresentar o modelo computacional dum solver de restrições caracterizando-o e especificando as entidades que o constituem; isolar o modelo da implementação sequencial dada no capítulo 3 e formular a hipótese duma implementação distribuída, introduzindo um possível cenário do paralelismo no modelo são os objectivos deste capítulo.

4.1 Introdução

A formulação teórica do solver de restrições cuja implementação em Java foi caracterizada no capítulo 3 permitirá, além da óbvia caracterização formal, isolar o modelo de qualquer implementação concreta, em Java ou numa qualquer outra linguagem. Permitirá também demonstrar a apropriação do modelo ao seu objectivo: um solver de restrições baseado e múltiplos estados adaptado a execuções sequenciais e paralelas. No decorrer deste capítulo a caracterização do modelo, fazer-se-á por: na secção 4.2, serão apresentadas as definições básicas do modelo, caracterizando-se as entidades habituais dos sistemas de restrições sobre domínios finitos: variáveis, domínios e problemas e o conceito de base do Ajacs: o estado. Na secção 4.3 é definida a problemática da propagação, sendo na secção 4.4 caracterizado o espaço de soluções e demonstrada a sua estrutura de árvore, o que permitirá abordar a problemática da resolução dum problema com restrições como um problema de pesquisa numa árvore de estados. Na secção 4.5 será finalmente abordada a pesquisa relacionando-a com as travessias efectuadas na árvore de estados.

Sempre que os termos usados correspondem a uma tradução para Português de um termo usado na literatura anglo-saxónica, o termo original aparece entre aspas.

4.2 Variáveis, Domínios e Problemas

Sendo um problema de restrições sobre domínios finitos definido através de variáveis domínios e restrições, o modelo dum solver de restrições sobre estes domínios terá inevitavelmente de formalizar a caracterização estas entidades:

Definição 1 (Domínio) *É um conjunto enumerável de valores quaisquer. Seja U o universo de todos os valores possíveis, v é um domínio se:*

$$v \equiv \{x_i : i \in \mathbb{N}, x_i \in U\} \tag{4.1}$$

Casos Particulares:

- *v é um domínio vazio se*

$$v \equiv \emptyset \tag{4.2}$$

- *v é um domínio singular se*

$$v \equiv \{x\} \tag{4.3}$$

Definição 2 (Funções sobre Domínios) *As funções sobre domínios permitem efectuar algumas das operações básicas sobre estes conjuntos. Dado um domínio, podemos determinar o seu primeiro (ou último) elemento usando a função α (ou ω), respectivamente. Podemos determinar o elemento de ordem i, usando θ, ou calcular o elemento que se segue a outro, usando η. Estas funções são caracterizadas por:*

$$\begin{array}{rccl} \alpha: & \mathcal{P}(U) & \longrightarrow & U \\ & \{x_1, ..., x_n\} & \longmapsto & x_1 \end{array} \tag{4.4}$$

$$\begin{array}{rccl} \eta: & U \times \mathcal{P}(U) & \longrightarrow & U \\ & (x, \{x_1, ..., x_n\}) & \longmapsto & x_i, \ se \ x \equiv x_{i-1} \end{array} \tag{4.5}$$

$$\begin{array}{rccl} \theta: & \mathbb{N} \times \mathcal{P}(U) & \longrightarrow & \mathcal{P}(U) \\ & (i, \{x_1, ..., x_n\}) & \longmapsto & \{x_i\} \end{array} \tag{4.6}$$

$$\begin{array}{rccl} \omega: & \mathcal{P}(U) & \longrightarrow & U \\ & \{x_1, ..., x_n\} & \longmapsto & x_n \end{array} \tag{4.7}$$

Sem perda de generalidade consideraremos no decorrer do capítulo uma reescrita da função 4.6 fazendo $\theta_i(v) \equiv \theta(i, v)$.

Definição 3 (Variável) *Uma variável é um índice dos possíveis duma colecção indexada de domínios. Representa-se por $V \equiv (v_1, v_2, \ldots, v_m)$ uma colecção indexada de domínios, e v_i, o domínio da variável* i.

Da definição apresentada ressalta que uma variável corresponde tão somente a uma posição relativa dum domínio particular no contexto duma colecção de valores.

Definição 4 (Estado) *É uma colecção indexada de domínios, que representa uma configuração para as variáveis da colecção. Representa-se-se por:*

$$s \equiv (V, i) \tag{4.8}$$

Sendo $V \equiv (v_1, v_2, \ldots, v_n)$, *a colecção indexada de domínios que caracteriza o estado e* i *uma variável distinguida da colecção.*

A variável distinguida no estado, indicará qual dos domínios da colecção foi reduzido na sua obtenção. Quando for irrelevante esta indicação o valor apresentado será zero. Representa-se por $s|_V$ o tuplo dos domínios, e $s|_{v_j}$ o domínio da variável j, no estado s.

Definição 5 (Estado Inconsistente) *Um estado diz-se inconsistente, se algum dos domínios da colecção é vazio, ou seja*

$$\exists j \ : s|_{v_j} = \emptyset \tag{4.9}$$

Definição 6 (Estado Consistente) *É consistente qualquer estado que não seja inconsistente, i.e.*

$$\forall j \ \#s|_{v_j} \geq 1 \tag{4.10}$$

Definição 7 (Estado Solução) *Um estado é uma solução se é um estado consistente, em que os domínios de todas as variáveis são singulares:*

$$\forall j \ \#s|_{v_j} = 1 \tag{4.11}$$

Definição 8 (Estados Iguais) *Dois estados* s_1 *e* s_2 *são iguais, se têm a mesma dimensão, e os respectivos domínios de ambas as colecções são idênticos:*

$$\forall j \ s_1|_{v_j} = s_2|_{v_j} \tag{4.12}$$

Definição 9 (Funções de selecção) *Uma função de selecção é uma função que dada uma colecção elege uma variável particular dessa colecção, dentre aquelas cujo domínio é não singular O objectivo da selecção, é definir qual a variável que será reduzida para construir novos estados. Considere-se $\mathcal{S}$, o conjunto de todas as funções de selecção, e $\mathcal{D}$ o conjunto de todos os domínios. $\mathcal{S}$ está definido por:*

$$\mathcal{S} = \left\{ \begin{array}{lccl} \delta: & \mathcal{D}^n & \longrightarrow & \mathbb{N} \\ & (v_1, ..., v_n) & \longmapsto & i : \#v_i > 1 \end{array} \right\} \tag{4.13}$$

Um elemento de $\mathcal{S}$, será uma função de domínio e contradomínio especificados, e que defina inequivocamente qual a variável seleccionada. A título de exemplo, considerem-se as funções δ_0 e δ_1 (equações 4.14 e 4.15 respectivamente) pertencentes a esta classe. A variável seleccionada, no caso de δ_0 corresponde à primeira variável da colecção cujo domínio é não singular e, no caso de δ_1 à de maior cardinalidade dentre aquelas que possam sofrer reduções.

$$\begin{array}{lccl} \delta_0: & \mathcal{D}^n & \longrightarrow & \mathbb{N} \\ & (v_1, ..., v_n) & \longmapsto & i : \forall j < i, \#v_j = 1 \end{array} \tag{4.14}$$

$$\begin{array}{lccl} \delta_1: & \mathcal{D}^n & \longrightarrow & \mathbb{N} \\ & (v_1, ..., v_n) & \longmapsto & i : \forall j < i : \#v_j > 1, \text{ se tem } \#v_i > \#v_j \end{array} \tag{4.15}$$

Definição 10 (Funções de redução) *Uma função de redução é uma função que dada uma colecção e uma variável dessa colecção, gera uma outra colecção, obtida por redução de domínio de uma das suas variáveis e mantendo os restantes domínios inalterados. Seja $\mathcal{R}$ o conjunto de todas as funções de redução, $\mathcal{R}$ é definido por:*

$$\mathcal{R} = \left\{ \begin{array}{lccl} \varphi: & \mathbb{N} \times \mathcal{D}^n & \longrightarrow & \mathcal{D}^n \\ & (i, (v_1, ..., v_n)), & \longmapsto & (v_1', ..., v_n') = \left\{ \begin{array}{ll} v_k' \equiv v_k & k \neq i \\ v_i' \subset v_i & \end{array} \right. \end{array} \right\} \tag{4.16}$$

Um exemplo duma função de $\mathcal{R}$, é a função φ_j (equação 4.17), em que a redução é feita escolhendo o j-ésimo elemento do domínio, usando a função θ definida em 4.6.

$$\begin{array}{rcl} \varphi_j : \mathbb{N} \times \mathcal{D}^n & \longrightarrow & \mathcal{D}^n \\ (i,(v_1,...,v_n)), & \longmapsto & (v_1',...,v_n') = \begin{cases} v_k' \equiv v_k & k \neq i \\ v_i' \equiv \theta_j(v_i) \end{cases} \end{array} \tag{4.17}$$

Definição 11 (Restrição('constraint')) *c, é uma restrição sobre as variáveis* $i_1,\ldots,i_k$, *escrevendo-se* $c|_{i_1,\ldots,i_k}$, *se c é uma relação sobre os domínios de* $i_1,\ldots,i_k$, *i.e. c é um subconjunto de* $v_{i_1} \times \ldots \times v_{i_k}$.

Definição 12 (Ambiente('environment') duma restrição) *Designa-se por ambiente duma restrição o conjunto de variáveis que nela intervêm. Seja C o conjunto de todas as restrições:*

$$\begin{array}{rcl} env : \mathcal{C} & \longrightarrow & \mathcal{P}(\mathbb{N}) \\ c|_{i_1,\ldots,i_k} & \longmapsto & \{i_1,...,i_k\} \end{array} \tag{4.18}$$

Definição 13 (Ordem duma variável) *Entende-se por ordem duma variável numa restrição, a ordem de ocorrência da variável, no ambiente da restrição:*

$$\begin{array}{rcl} ord : \mathbb{N} \times \mathcal{C} & \longrightarrow & \mathbb{N} \\ (i,c) & \longmapsto & m, \ se \ env(c) = \{i_1,\ldots,i_{m-1},i,i_{m+1},\ldots\} \end{array} \tag{4.19}$$

Definição 14 (Dependências duma variável) *As dependências duma variável são dadas pelo conjunto de associações var/restrição , desde que ambas as variáveis pertençam ao ambiente da restrição:*

$$dep(x) = \{v/c : x, v \in env(c), v \neq x\} \tag{4.20}$$

■

Por exemplo se num sistema estiverem definidas somente duas restrições c_1 e c_2, em que os respectivos ambientes são dados por $env(c_1) = \{1,2\}$ e $env(c_2) = \{2,3\}$ então as dependências das variáveis são dadas por:

$$\begin{array}{rcl} dep(1) & = & \{2/c_1\} \\ dep(2) & = & \{1/c_1, 3/c_2\} \\ dep(3) & = & \{2/c_2\} \end{array}$$

A definição de dependências torna-se necessária para determinar quais e como são afectadas as variáveis dum sistema de restrições quando uma variável sofre uma redução de domínio. Se i é reduzido as variáveis afectadas serão, directamente, todas as variáveis $j : j/c_x \in dep(i)$, e o modo como são afectadas está expresso em c_x. As dependências permitiram o cálculo da consistência de arco, definindo os arcos e rotulando-os.

Definição 15 (Função de restrição) *Para cada restrição, c, deverá existir um função u, chamada função de restrição, que dada a variável j, cujo domínio foi alterado, gera o domínio concordante com v_j segundo a restrição, para todas as variáveis, i, do ambiente de c.*

$$\begin{aligned} u: \ & \mathcal{C} \times \mathbb{N} \times \mathbb{N} \longrightarrow \mathcal{P}^{\mathcal{D}} \\ & (c,i,j) \longmapsto v_i' \subseteq vi : \forall x \in v_i, x \notin v_i' \iff \nexists y \in v_j : c(x,y) \text{ é valida} \end{aligned} \tag{4.21}$$

■

Exemplo: Considere-se c, a restrição correspondente a $X \neq Y$. A função $u_1(c,i,j) = v_i \backslash v_j \cap v_i$, não é uma função de restrição dado eliminar dos domínios v_i valores para os quais existe em v_j valores que validam a restrição. Se $v_j = [1-2]$ e $v_i = [1-10]$ v_i seria actualizado para $[3-10]$ o que seria absurdo, dado que quando v_j é igual a 1, v_i pode ser 2 e vice-versa. A função $u_2(c,i,j) = \begin{cases} v_i \backslash v_j & se \ \#v_j = 1 \\ v_i & \text{senão} \end{cases}$, é uma possível função de restrição para a para a mesma restrição.

Definição 16 (Problema) *A formulação de um problema com restrições, contempla a definição dum conjunto de variáveis, cada uma das quais, tomando valores num domínio, e de um conjunto de restrições sobre essas variáveis. Englobando o conceito de estado(definição 4), o de variáveis e de domínio, um problema é definido por:*

$$p \equiv (s_{init}, C) \tag{4.22}$$

Onde:

- *s_{init} é um estado, designado por estado inicial, obtendo-se $p|_{s|_v}$ de $(v_1, ..., v_n)$, tomando para v_i o domínio inicial da variável i;*

- $C = \{c_1, c_2, \cdots, c_k\}$, *são as restrições do problema.*

Definição 17 (Estratégia de Pesquisa) *Uma estratégia representa uma escolha do modo como serão realizadas as reduções de domínio, no processo que conduzirá à determinação das soluções, i.e. o processo de pesquisa. É definida por um par, constituído por uma função e um grupo de funções, que determinam, uma, a variável que sofrerá a redução, as outras, o modo particular desta.*

$$\lambda \equiv (\delta, \Gamma) \text{ é uma estratégia } \iff \begin{cases} \delta \in \mathcal{S} \\ \Gamma = \{\gamma_1, \ldots, \gamma_n\}, n \geq 2, \gamma_i \in \mathcal{R} \end{cases} \tag{4.23}$$

O conjunto de funções Γ *deverá efectuar uma partição do domínio da variável ao qual se aplica i.e.:*

$$\forall X \in \mathcal{D}^n, \bigcup_{j=1}^{n} \gamma_j(X) = X$$

$$\forall k \neq l, \gamma_k(X) \cap \gamma_l(X) = \emptyset$$

Definição 18 (Aplicação duma estratégia) *Seja* Σ *o conjunto de todos os estados possíveis. A aplicação duma estratégia a um estado obedece a:*

$$(\delta, \Gamma): \begin{array}{lll} \Sigma & \longrightarrow & \mathcal{P}(\Sigma) \\ (V, i) & \longmapsto & \{s_j : s_j = (V', i')\} \text{ onde } \begin{cases} i' = \delta(V) \\ V' = \gamma_j(i', V) \end{cases} \end{array} \tag{4.24}$$

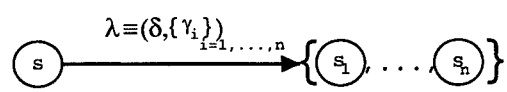


Figura 4.1: Aplicação de uma estratégia

■

Exemplo: Seja $\lambda = (\delta_d, \{\gamma_1, \gamma_2\})$ a estratégia definida por uma função de selecção em que a variável seleccionada é aquela que possui mais dependências, e as funções de redução partilham ao meio o domínio inicial:

$$\delta_d(V) = \min_i \left\{ i : \#dep(i) = \max_{\#dep(j)} \{ j : \#v_j > 1 \} \right\}$$

$$\gamma_1(\{x_1, \ldots, x_n\}) = \{x_1, \ldots, x_k\}$$
$$\gamma_2(\{x_1, \ldots, x_n\}) = \{x_{k+1}, \ldots, x_n\}, \quad k = \lfloor n/2 \rfloor$$

Suponha-se que pretendemos aplicar a estratégia λ ao estado s, definido por 3 variáveis de domínio igual a $\{1, ..., 5\}$, i.e., $s \equiv \Big((\{1, \ldots, 5\}, \{1, \ldots, 5\}, \{1, \ldots, 5\}), 0\Big)$. Suponha-se igualmente que o número de dependências é dado por $\#dep(1) = \#dep(2) = \#dep(3) = 2$.

$$\lambda(s) = \left\{ \Big((\{1,2\}, \{1, \ldots, 5\}, \{1, \ldots, 5\}), 1\Big), \Big((\{3,4,5\}, \{1, \ldots, 5\}, \{1, \ldots, 5\}), 1\Big) \right\}$$

O conjunto $\lambda(s)$ é constituído pelos dois estados que derivam da partição do domínio da variável 1, em dois subdomínios. Tendo todas as variáveis o mesmo número de dependências escolhe-se a primeira variável, cujo domínio esteja por reduzir, i.e. a variável 1.

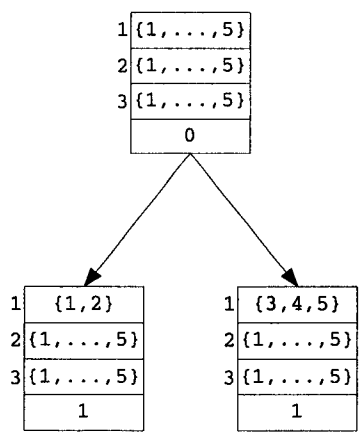


Figura 4.2: Exemplo de aplicação da estratégia $\lambda = (\delta_d, \{\gamma_1, \gamma_2\})$

4.3 Propagação

Uma redução de domínio, ao alterar os valores que determinada variável pode assumir, terá necessariamente de ser acompanhada duma propagação. Esta, per-

mitirá manter a consistência do estado, visto actualizar os valores das restantes variáveis, em função das restrições do problema.

Definição 19 (Propagação Parcial) *A função de propagação* Π, *é definida pela equação 4.25, aplicando-se a uma variável e a um estado, gerando um novo estado:*

$$\begin{array}{rcl} \Pi: \mathbb{N} \times \Sigma & \longrightarrow & \Sigma \\ \Big(j, s \equiv ((v_1, \ldots, v_n), l)\Big) & \longmapsto & ((v'_1, \ldots, v'_n), 0) \ \textit{onde} \\ & & \begin{cases} v'_x = u(c, x, j) & , \ \forall x/c \in dep(j) \\ v'_i = v_i & , \ \textit{para os restantes domínios} \end{cases} \end{array} \tag{4.25}$$

De modo análogo ao considerado anteriormente, reescreva-se $\Pi(j, s)$ como $\Pi_j(s)$.

■

Exemplo: Suponha-se que pretendemos propagar os efeitos duma redução efectuada no estado $s_1 \equiv \Big((\{1, \ldots, 5\}, \{2\}, \{1, \ldots, 5\}), 2\Big)$.

Sejam $\{c_1, c_2, c_3\} \equiv \{1 \neq 2, 2 > 3, 1 \neq 3\}$ as restrições do problema. O respectivo ambiente é definido por

$$\begin{aligned} env(c_1) &= \{1, 2\} \\ env(c_2) &= \{2, 3\} \\ env(c_3) &= \{1, 3\} \end{aligned}$$

sendo as dependências

$$\begin{aligned} dep(1) &= \{2/c_1, 3/c_3\} \\ dep(2) &= \{1/c_1, 3/c_2\} \\ dep(3) &= \{2/c_2, 1/c_3\} \end{aligned}$$

e as funções de restrição dadas por

$$u(c_1, i, j) = u(c_3, i, j) = u_2(c_1, i, j) \text{ dada no exemplo da definição 15}$$

$$u(c_2, i, j) = \begin{cases} v_i \backslash \{\omega(v_j), \ldots, \omega(v_i)\} & , ord(j, c_2) = 1 \\ v_i \backslash \{\alpha(v_i), \ldots, \alpha(v_j)\} & , ord(j, c_2) = 2 \end{cases}$$

Atendendo às dependências da variável 2, a variável reduzida, e por que $u(c_1, 1, 2) = \{1, 3, 4, 5\}$ e $u(c_3, 3, 2) = \{1\}$,

$$\Pi_1(s_1) = \Big((\{1, 3, 4, 5\}, \{2\}, \{1\}), 0\Big)$$

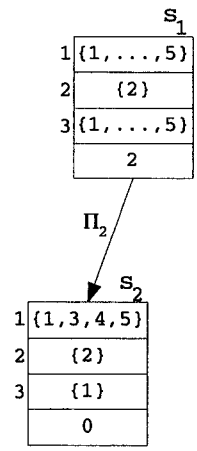


Figura 4.3: Exemplo duma propagação parcial

Definição 20 (Retorno duma propagação) *Designa-se por retorno duma propagação, o conjunto formado pelas variáveis cujo domínio foi alterado pela propagação.*

$$ret(\Pi_j(s)) = \{x : s|_{v_x} \neq \Pi_j(s)|_{v_x}\} \tag{4.26}$$

■

Como consequência da definição tem-se que $ret(\Pi_j(s)) = \{\} \iff \Pi_j(s) = s$

Para o exemplo anterior $ret(\Pi_2(s_1)) = \{1, 3\}$. Como é usual na determinação da consistência de arco (ver secção 1.3.1), não basta calcular a consistência dos arcos uma só vez, sendo necessário reavaliar os arcos cujos vértices tenham sido alterados. O retorno da propagação servirá para determinar quais os domínios alterados e que por esse motivo devam induzir novas propagações.

A propagação parcial, actualiza todas as variáveis que estão relacionadas directamente, através das restrições, com a variável reduzida. Se alguma destas variáveis se modificar, outras propagações têm de ser efectuadas para garantir a concordância dos novos domínios com as restrições do problema.

Definição 21 (Propagação Total) *Define-se propagação total de um estado s, por alteração de um conjunto de variáveis A, $\Pi_A^T(s)$, como:*

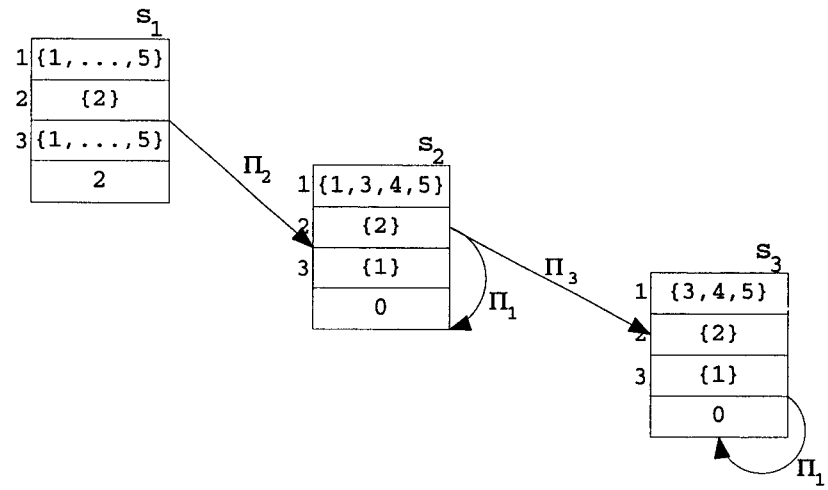
$$\begin{array}{rccl} \Pi_A^T: & \Sigma & \longrightarrow & \Sigma \\ & s & \longmapsto & \Pi^T_{A\setminus\{j\}\cup R}(\Pi_j(s)),\ para\ R = ret(\Pi_j(s)), j \in A, \end{array} \tag{4.27}$$

sendo:
$\Pi_A^T(s) = s, se A \equiv \{\}$
$\Pi_A^T(s) = \emptyset, se \exists i \in A : \Pi_i(s) = s',$ *com s' inconsistente*

■

Da definição apresentada sai que, a propagação só termina quando não existirem mais variáveis para propagar ($A = \{\}$), ou se na sucessiva propagação das variáveis for obtido algum estado inconsistente. Torna-se claro que existirá sempre um ponto fixo, que terminará a propagação total. O vazio é um deles, a demonstração do outro ponto fixo advém da definição da função de actualização. A dimensão máxima de A é dada pelo número de variáveis do estado. Este valor é um majorante para a cardinalidade das variáveis a propagar. Ao propagar uma das variáveis a dimensão de A só aumenta de forem modificados os domínios de variáveis que não estejam em A (por definição de união). Logo uma variável já propagada só voltará a ser novamente propagada se seu domínio for reduzido. Atendendo a que os domínios não são infinitos ou as propagações não alteram domínios, não existindo o retorno das propagações o que conduz a reduzir a dimensão de A ou obter-se-á um estado inconsistente e que termina a propagação total.

Exemplo: Suponhamos que pretendemos determinar o estado resultante da propagação total do estado s_1, do exemplo da definição 19, por redução da variável



$dep(1) = \{2/c_1, 3/c_3\}$ $dep(3) = \{2/c_2, 1/c_3\}$ $dep(1) = \{2/c_1, 3/c_3\}$

$u(c_1, 2, 1) = \{2\}$ $u(c_2, 2, 3) = \{2\}$ $u(c_1, 2, 1) = \{3, 4, 5\}$

$u(c_3, 3, 1) = \{1, 3, 4, 5\}$ $u(c_3, 1, 3) = \{3, 4, 5\}$ $u(c_3, 3, 1) = \{1\}$

$R_2 = ret(\Pi_1(s_2)) = \emptyset$ $R_3 = ret(\Pi_3(s_2)) = \{1\}$ $R_4 = ret(\Pi_1(s_3)) = \emptyset$

Figura 4.4: Exemplo duma propagação total.

2:

$$\Pi^T_{\{2\}}(s_1) = \Pi^T_{R_1}(\Pi_2(s_1)) = \Pi^T_{\{1,3\}}(s_2) = \Pi^T_{\{3\}\cup R_2}(\Pi_1(s_2)) = \Pi^T_{\{3\}}(s_2) = \Pi^T_{R_3}(\Pi_3(s_2)) = \\ = \Pi^T_{\{1\}}(s_3) = \Pi^T_{R_4}(\Pi_1(s_3)) = s_3 \tag{4.28}$$

Uma visualização do processo efectuado e os principais passos necessários à construção dos sucessivos estados obtidos durante propagação total constam na figura 4.3.

Teorema 1 (Comutatividade) *A propagação total é comutativa,i.e.*

$$\Pi^T_{\{a,b\}}(s) = \Pi^T_{\{b,a\}}(s) \tag{4.29}$$

A demonstração do teorema fazer-se-á por indução no retorno.

Caso 0: $R_a = ret(\Pi_a(s)) = \{\}$ $R_b = ret(\Pi_b(s)) = \{\}$

$\Pi^T_{\{a,b\}}(s) = \Pi^T_{\{a\}\cup R_b}(\Pi_b(s)) = \Pi^T_{\{a\}}(s) = \Pi^T_{R_a}(\Pi_a(s)) = \Pi^T_{\{\}}(s) = s$

$\Pi^T_{\{b,a\}}(s) = \Pi^T_{\{b\}\cup R_a}(\Pi_a(s)) = \Pi^T_{\{b\}}(s) = \Pi^T_{R_b}(\Pi_b(s)) = \Pi^T_{\{\}}(s) = s$
Caso 1: $R_a = ret(\Pi_a(s)) = \{x\}$ $R_b = ret(\Pi_b(s)) = \{\}$

$$\Pi^T_{\{a,b\}}(s) = \Pi^T_{\{b\}\cup R_a}(\Pi_a(s)) = \Pi^T_{\{b,x\}}(\Pi_a(s)) = \Pi^T_{\{x\}\cup R'_b}(\Pi_b(\Pi_a(s))) \quad (4.30)$$
$$\Pi^T_{\{b,a\}}(s) = \Pi^T_{\{a\}\cup R_b}(\Pi_b(s)) = \Pi^T_{\{a\}}(s) = \Pi^T_{\{x\}}(\Pi_a(s)) \quad (4.31)$$

Demonstrando que $R'_b = \{\}$ em 4.30 e que consequentemente também em 4.30 se tem $\Pi_b(\Pi_a(s)) = \Pi_a(s)$ temos 4.30=4.31 e o caso 1 está demonstrado.

Vamos então demonstrar que se $\Pi_x(s) = s \Rightarrow \Pi_x(s') = s'$, para $s \equiv (v_1, \ldots, v_x, \ldots, v_n))$ $s' \equiv (v'_1, \ldots, v_x, \ldots, v'_n)) : v'_i \subseteq v_i$

Se $\Pi_x(s \equiv (v_1, \ldots, v_x, \ldots, v_n)) = s$ é porque a redução de x não altera nenhum dos v_i para $i \neq x$, logo por definição de função de restrição $\forall i / c_k \in dep(x)$ tem-se $\forall y \in v_i \; \exists z \in v_x : c_k(y, z)$ é válido. Sendo as dependências as mesmas e tendo-se $v'_i \subset v_i \Rightarrow \forall y \in v'_i \; \exists z \in v_x : c_k(y, z)$ é válido, logo a propagação parcial de x não altera nenhum dos domínios de s' e $\Pi_x(s') = s'$.

A demonstração do caso 2: $R_a = ret(\Pi_a(s)) = \{\}$ $R_b = ret(\Pi_b(s)) = \{x\}$ é em tudo análoga à do caso 1, e portanto dispensa demonstração. A demonstração do caso 3:: $R_a = ret(\Pi_a(s)) = \{x\}$ $R_b = ret(\Pi_b(s)) = \{y\}$ fica demonstrada se for provado que $\Pi_b(\Pi_a(s)) = \Pi_a(\Pi_b(s))$. A demonstração não requer qualquer técnica além da uso da definição de função de restrição usada no caso 1. Além disso é intuitivo considerar que ao eliminar dum qualquer domínio v_i as inconsistências com os valores de a e depois as inconsistências com os valores de b se obtém o mesmo domínio que ao eliminar primeiro as inconsistências de b e depois de a .

4.4 Árvore de Estados (estrutura do espaço de soluções)

Definição 22 (Expansão de um estado) *Considera-se como, expansão de um estado s segundo uma estratégia $\lambda \equiv (\delta, \Gamma)$, o conjunto de estados derivados de s, por aplicação da estratégia λ, aos quais é aplicada a propagação total, por al-*

teração da variável $\delta(s|_V)$. Seja Λ o conjunto de todas as estratégias, a expansão de um estado é dada por:

$$\begin{array}{rccl} \chi: & \Lambda \times \Sigma & \longrightarrow & \mathcal{P}(\Sigma) \\ & ((\delta,\Gamma),s) & \longmapsto & \{s' : s' = \Pi_j^T(s''), \forall s'' \in (\delta,\Gamma)(s), \wedge j = \delta(s|_V)\} \end{array} \tag{4.32}$$

A expansão de um estado é um conjunto ordenado, usando a ordem a dos índices que geram os elementos da expansão. Sejam $x, y \in \chi(\lambda, s)$, então

$$\exists i,j,k : \left\{ \begin{array}{rcl} x & = & \Pi_j^T(\gamma_i(s)) \\ y & = & \Pi_j^T(\gamma_k(s)) \end{array} \right. \quad considera-se\ x < y \iff i < k$$

Definição 23 (Grafo de estados) *A definição dum problema através dum estado inicial, s_{init}, e dum conjunto de restrições C, permite, conjuntamente com uma estratégia definir o grafo de estados do problema. Sejam p um problema e $\lambda = (\delta, \Gamma)$ a estratégia elegida. O grafo de estados do problema $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ é construído iterativamente por:*

1. $s_{init} \in \mathcal{N} \iff s_{init}$ *não é solução*
2. $\forall s' \in \mathcal{N} \backslash \{Soluções\}, s'' \in \mathcal{N} \wedge (s', s'') \in \mathcal{A} \iff$
   $\delta(s') = i \wedge \exists j : s'' = \Pi_i^T(\gamma_j(i, s'|_V))$ *é consistente*

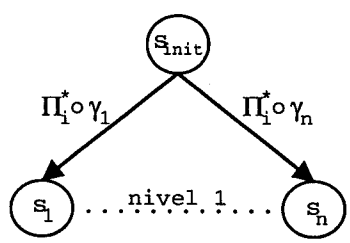


Figura 4.5: Grafo de Estados

A construção do grafo de estados, é iniciada usando o nó s_{init}, e gerando todos os sucessores deste estado. Estes nós são obtidos escolhendo no estado inicial uma variável, i, para reduzir, (i.e. aplicar δ), e efectuar sobre o domínio correspondente à variável, $s_{init}|_{v_i}$, as partições motivadas pela aplicação das funções γ_j. Considerando n a cardinalidade de Γ, os estados assim obtidos são os estados

$s_1, \ldots, s_n$ da figura 4.5. Caso algum dos estados assim obtidos seja inconsistente, por definição, não pertence ao grafo que só engloba os estados que após propagadas sejam consistentes. Os estados do nível 1 são todos distintos, visto serem igualmente distintas as respectivas colecções. Pelo menos o domínio v_i, é distinto em todos eles, e mais ainda, os diferentes v_is não possuem nenhum elemento em comum, atendendo a que formam uma partição. Tal significa que o grafo da figura 4.5 é uma árvore, de raiz em s_{init}.

Os nós do nível 2 são obtidos expandindo os nós do nível 1 que não sejam solução, de modo análogo ao realizado para o nó inicial. O grafo da figura 4.7 ilustra a construção. Os nós $s_{11}, \ldots, s_{1m}$, são distintos entre si, pelos motivos expostos para o nível 1, agora aplicados à variável j, e equivalentemente os nós $s_{n1}, \ldots, s_{nl}$, são igualmente distintos. De modo análogo não poderá existir nenhum estado, que seja derivável de antecessores diferentes (arcos do tipo 1 na figura 4.7), visto que se os antecessores são diferentes e do mesmo nível, o domínio da variável que foi reduzida no nível é disjunto dos domínios dos correspondentes irmãos. Usando a figura 4.7 podemos concluir que não existem arcos entre,

tipo 1: determinado nó, e um qualquer irmão do pai;

Se demonstrarmos que não existem arcos do tipo 2, i.e, arcos entre:

tipo 2: nós do mesmo nível, mas com antecessores diferentes;

temos demonstrado que o grafo de estados é uma árvore, o que para efeitos de travessias é claramente positivo, dado ser significativamente menor a informação que é necessário armazenar para atravessar uma árvore do que um grafo.

Um caso particular dos arcos do tipo 1, está identificado na figura 4.7 como o arco tracejado com o número 1, sendo uma particularização do caso 2, o arco a tracejado com o número 2.

Teorema 2 (Árvore de estados) *O grafo da definição 23 é um árvore.*
Dem:

$$\textit{Suponhamos que } s^* = s^{**} \Rightarrow \forall k, s^*|_{v_k} = s^{**}|_{v_k},$$

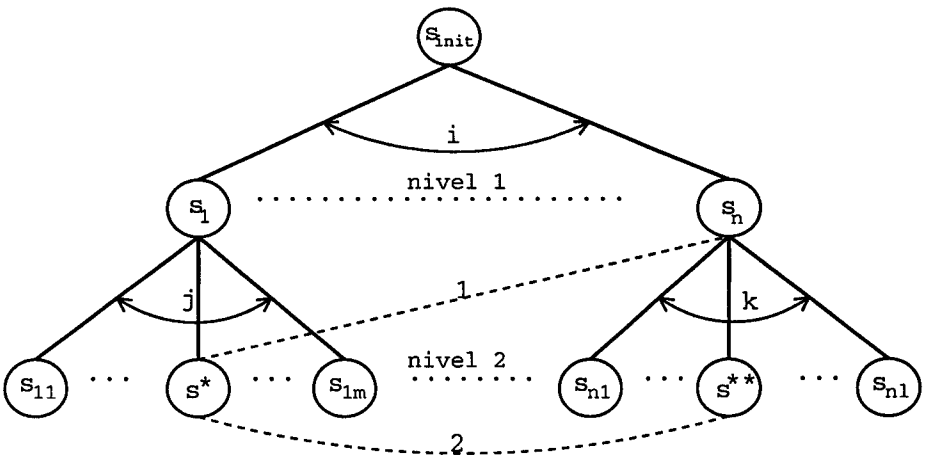


Figura 4.6: Árvore de Estados

$$\textit{em particular } s^*|_{v_i} = s^{**}|_{v_i} \tag{4.33}$$

$$\textit{Como por hipótese } (s_1, s^*) \in \mathcal{A} \Rightarrow s^*|_{v_i} \subseteq s_1|_{v_i} \tag{4.34}$$

$$\textit{Como também } (s_n, s^{**}) \in \mathcal{A} \Rightarrow s^{**}|_{v_i} \subseteq s_n|_{v_i} \tag{4.35}$$

$$\textit{De } 4.33 \textit{ , } 4.34 \textit{ e de } 4.35 \textit{ tem-se que } s^*|_{v_i} \subseteq s_1|_{v_i} \cap s_n|_{v_i} \tag{4.36}$$

Por definição de partição $s_1|_{v_i} \cap s_n|_{v_i} = \emptyset$ *logo* 4.36 *é um absurdo.*

■

O teorema ??

4.5 Pesquisa (exploração do espaço de soluções)

A existência duma árvore de estados cujas folhas são potenciais soluções implica que o processo de pesquisa passa por realizar travessias nessa árvore. As travessias são efectuadas por uma entidade designada agente. A tarefa dum agente é realizada globalmente, através de visitas pontuais aos nós da árvore. Cada nó é processado individualmente, e caso não seja solução, os seus sucessores são adicionados a um conjunto global de nós a processar proximamente. Um nó desta

lista, e não necessariamente um sucessor directo do nó processado, é eleito para ser visitado de seguida pelo agente, e assim sucessivamente. A tarefa do agente termina, quando é encontrada uma solução, ou quando todos os nós da árvore foram processados sem sucesso, em consequência da propagação resultar em falha.

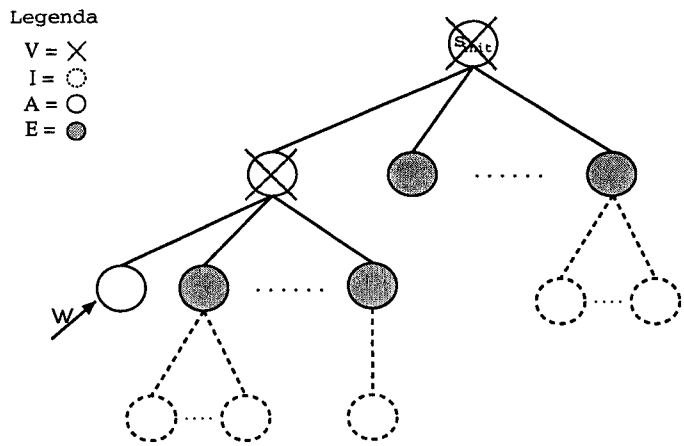


Figura 4.7: Árvore de Estados

A árvore de estados é caracterizada por 4 conjuntos de nós que permitem avaliar o processo de pesquisa. Inicialmente, um ou mais estados estão à espera de que algum agente os processe, e constituem o conjuntos de nós à espera. Os restantes estados são o conjunto de nós ocultos. O processamento de um estado por algum dos agentes da pesquisa, torna-o num nó activo e posteriormente, findo o processamento, num nó visitado. As travessias da árvore pelos agentes, actualizam estes conjuntos. Mais do que um nó pode ser processado simultaneamente, se existir mais do que um agente. A ilustração deste processo está na figura 4.7, onde `A` representa o conjunto de nós activos, `I` o conjunto de nós ocultos, `V` o conjunto de nós visitados e `E` o conjunto de nós em espera.

Definição 24 (Agente) *Um agente é uma entidade unicamente caracterizada pelo seu estado (ver definição 26). A tarefa do agente é efectuar percursos na árvore de estados. O modo como são realizados estes percursos é definido pelo algoritmo do agente.*

Definição 25 (Percurso dum agente) *Designa-se por percurso dum agente, a sequência de estados, $s_1 \ldots s_n$, correspondentes aos nós da árvore visitados pelo agente. A ordem pela qual figuram os estados no percurso corresponde à ordem de visita.*

Definição 26 (Estado do agente) *Designa-se por estado (actual) do agente, o estado que está nesse instante a ser processado pelo agente. São múltiplos os estados de um agente no decorrer de todo o processo, e constituem como foi dito na definição 25 o percurso do agente, mas é único o estado actual de um agente, num instante particular, i.e:*

$$\begin{array}{rccc} estado: & W \times T & \longrightarrow & \Sigma \\ & (w,t) & \longmapsto & s \end{array} \tag{4.37}$$

A semântica da função estado é dada por:
$estado(w,t) \equiv s \iff s$ é o estado actual do agente w, no instante t

Definição 27 (Estados Activos) *Num determinado instante, um estado é activo, se é o estado actual de algum agente. O conjunto de estados activos, A, representa os estados que estão nesse instante a ser processados por algum agente. Seja s o estado actual do agente w no instante t. Quando w terminar o processamento do nó s, o conjunto de estados activos, A, sofre a actualização:*

$$A_{t+1} = A_t \backslash \{s\} \tag{4.38}$$

Definição 28 (Estados Visitados) *Quando um agente termina o processamento do seu nó, este é adicionado ao conjunto de estados visitados. Seja $s = estado(w,t)$. O conjunto de estados visitados, V, sofre a actualização,*

$$V_{t+1} = V_t \cup \{s\} \tag{4.39}$$

após w acabar de processar s.

Definição 29 (Estados em Espera) *Os estados cuja existência já foi determinada, mas que ainda não foram visitados, nem são o estado actual de nenhum agente, estão em espera. A existência destes estados é determinada pela expansão*

dos nós activos dos agentes. Seja E o conjunto de estados em espera, num determinado instante t, seja $s = estado(w,t)$, e $\chi(\lambda, s)$ a expansão do estado actual do agente w. Todos os estados de $\chi(\lambda, s)$, são adicionados a E, conjunto de estados em espera. Após w terminar o processamento do nó actual, E sofre a actualização:

$$E_{t+1} = E_t \cup \chi(\lambda, s) \tag{4.40}$$

Definição 30 (Funções de selecção dum nó em espera) *A existência de um conjunto de estados em espera, motiva a existência de funções que determinem qual dos estados deste conjunto seja eleito para ser o próximo estado actual do agente. Estas funções, designadas por funções de selecção de nó em espera, serão as responsáveis pelo percurso efectuado pelos agentes. De um modo genérico uma função de selecção de nó em espera é uma função que, dado um estado activo, determina o próximo estado actual do agente em causa, i.e.:*

$$\begin{array}{rccc} \rho: & A & \longrightarrow & E \\ & s_a & \longmapsto & s_q \end{array} \tag{4.41}$$

Muitas funções de selecção de nó são passíveis de ser definidas, umas elementares, outras mais complexas. Se existir um só agente, a função de selecção definirá a ordem pela qual os nós da árvore serão visitados, sendo a ordem possível de definir, i.e. podemos garantir que determinados nós são visitados antes de outros, e travessias de tipo, Deepth-first-search e Breath-first-search são implementáveis.

Podemos considerar uma função inversa da função de selecção de nó, como

$$\begin{array}{rccc} \rho^{-1}: & A & \longrightarrow & V \\ & s_a & \longmapsto & s_v, \quad s_a \in \chi(\lambda, s_v) \end{array} \tag{4.42}$$

A função de selecção de nó ρ_1, elege o sucessor mais à esquerda na árvore de estados, do nó actual, para constituir o próximo estado actual do agente. Existindo um só agente, esta função permite implementar a estratégia Deepth-first left-to-right.

$$\begin{array}{rccl} \rho_1: & A & \longrightarrow & E \\ & s & \longmapsto & \begin{cases} min\ \chi(\lambda, s), & se\ \chi(\lambda, s) \neq \emptyset \\ min\ Q \cap \chi(\lambda, \rho^{-j}(s)), & se\ \chi(\lambda, \rho^{-i}(s)) = \emptyset, \forall i = 1, \ldots, j-1 \end{cases} \end{array}$$

Definição 31 (Algoritmo dos agentes) *Todos os agentes executam a mesma tarefa, que define o método global utilizado para pesquisar soluções na árvore de estados. Os nós são processados pelos agentes, segundo o seu algoritmo. A árvore de estados encontra-se dividida nos quatro conjuntos de nós, Activos, Visitados, em Espera e Implícitos. A cada agente cabe a missão de processar o nó que lhe está atribuído,i.e. o seu estado actual. Se o estado do agente é uma solução o processo termina disponibilizando a solução, caso contrário são gerados os sucessores do estado actual que incrementam o conjunto de estados em espera. Deste conjunto é eleito o próximo estado actual do agente, e o processo repete-se. Na figura 4.8 encontra-se o código do algoritmo descrito.*

Algoritmo WORK(e)
{input: e, o estado activo do agente}
{output: solução s}
$A = A \cup \{e\}$
if solução(e) then
  $V = V \cup \{e\}$
  $A = A \backslash \{e\}$
  $Return(e)$
else
  $E = E \cup \{\chi(\lambda, e)\}$
  $n = \rho(e)$
  $E = E \backslash \{n\}$
  $V = V \cup \{e\}$
  $A = A \backslash \{e\}$
  $Return(WORK(n))$

Figura 4.8: Algoritmo recursivo executado pelos agentes

4.5.1 Exemplo

Considere-se a árvore de estados da figura 4.9 com os nós rotulados para melhor exemplificar o processo da pesquisa. Considere-se o problema da determinação da primeira solução, que, no caso exemplificado corresponde ao nó `10`, e é a única solução. Considere-se também que a pesquisa é executada por um único agente, e que o algoritmo executado pelo agente correspondente ao da figura 4.8. A tabela da figura ?? mostra as actualizações nos conjuntos `Activos`, em `Espera` e `Visitados`, bem como a selecção de nó em espera efectuada pelo algoritmo, que no exemplo dado corresponde à aplicação de ρ_1. O processo inicia-se com a execução de `work`(s_{init}). Na tabela cada linha corresponde à visita ao nó correspondente e portanto à execução do algoritmo `work` para esse nó. As eliminações de nós dos respectivos conjuntos estão assinalados com um corte sobre o nó eliminada. A figura 4.9 apresenta a árvore após a determinação da solução e o consequente fim da pesquisa.

4.6 Conclusões

Foi apresentado no decorrer do capítulo a formalização teórica do modelo computacional subjacente ao Ajacs. Neste modelo, uma estrutura básica, o estado, armazena um conjunto de valores que representam o domínio das variáveis dum problema de restrições sobre domínios finitos.

Pelo uso duma estratégia, um estado é expandido sendo produzidos novos estados concordantes com o conjunto de restrições do problema com a particularidade dos seus domínios serem de cardinalidade inferior aos do seu antecessor. Tal facto advêm do modo como o novos estados são gerados: por partição de algum dos seus domínios e por propagação. Este processo induz ao espaço de pesquisa uma estrutura de árvore cujas folhas são potenciais soluções do problema de restrições.

A travessia do espaço de pesquisa pode ser efectuada sequencialmente por um agente ou em paralelo por mais do que um. A adequação do modelo a implementações paralelas será objecto de análise nos capítulos 5, 6 e 7.

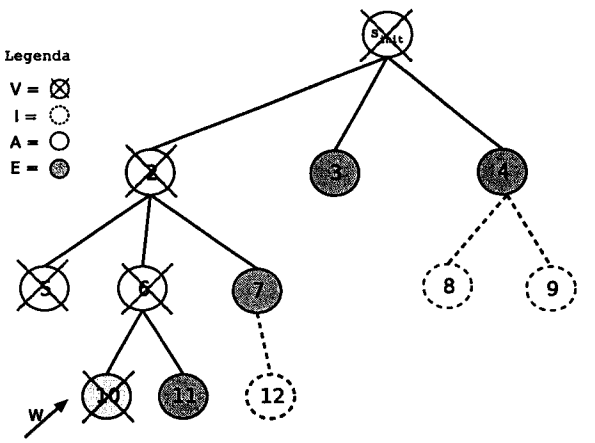


Figura 4.9: Árvore de pesquisa do exemplo da secção 4.5.1

	Activos	Espera	ρ_1	Visitados
work(s_{init})	$\{s_{init}\}$	$\{\cancel{2},3,4\}$	2	$\{s_{init}\}$
work(2)	$\{\cancel{2}\}$	$\{3,\cancel{4},5,6,7\}$	5	$\{s_{init},2\}$
work(5)	$\{\cancel{5}\}$	$\{3,\cancel{4},6,7\}$	6	$\{s_{init},2,5\}$
work(6)	$\{\cancel{6}\}$	$\{3,4,7,\cancel{10},11\}$	10	$\{s_{init},2,5,6\}$
work(10)	$\{\cancel{10}\}$	$\{3,4,7,11\}$	-	$\{s_{init},2,5,6,10\}$

Capítulo 5

Sistemas distribuidos e DSMs: aplicabilidade ao AJACS

A execução do modelo distribuído do AJACS usando um cluster de computadores, requer o uso duma aplicação que distribua os threads do Java pelas diferentes CPUS do cluster, implemente um sistema de memória partilhada e distribuída e o consequente modelo de memória do Java. A aplicação usada foi o Hyperion. Neste capítulo são abordados os temas das arquitecturas multiprocessador, modelos de consistência e o modo de funcionamento do Hyperion.

5.1 Introdução

É aceitável considerar ([HP96]) que os microprocessadores permanecerão provavelmente como a tecnologia uniprocessador dominante e que um modo evidente para aumentar o desempenho é conectar múltiplos microprocessadores. Trata-se obviamente duma abordagem mais vantajosa em termos de custos, do que o desenho dum processador com este propósito específico. Desde 1985 as inovações arquitectónicas têm contribuído para termos microprocessadores cada vez mais rápidos, mas esta razão de crescimento não poderá ser mantida eternamente. Sem dúvida que o crescimento continuará embora a um ritmo mais lento. Os ganhos de desempenho têm sido obtidos construindo sistemas cada vez mais complexos e está-se a atingir um estado em que o aumento de complexidade não trará necessariamente muito maior desempenho. Por estes motivos as máquinas de processamento paralelo terão no futuro um papel de destaque. A vulgarização do uso de máquinas paralelas está mais condicionada pela inexistência de software paralelo do que por limitações de hardware.

5.2 Multiprocessadores

Nos últimos anos as máquinas MIMD(Multiprocessor Instruction Streams, Multiple Data Streams), isto é organizações em que cada processador busca as suas próprias instruções e opera os seus próprios dados, emergiram como a arquitectura de eleição para multiprocessadores de propósito generalizado. As MIMD oferecem flexibilidade pois desde que acompanhadas de hardware e software apropriados podem funcionar como máquinas monoutilizador com enfoque orientado para o desempenho duma aplicação, como máquinas multiprogramação que realizam múltiplas tarefas em simultâneo ou como uma qualquer combinação destes dois usos. Por outro lado as MIND podem apresentar uma relação custo/desempenho bastante apelativa dado que quase todos os multiprocessadores existentes são construídos com microprocessadores existentes em estações de trabalho e pequenos servidores uniprocessador.

A classificação das MIMD recai em duas categorias distintas, em função do

número de processadores conectados, o que por sua vez condiciona a organização de memória subjacente e a estratégia de conexão entre eles. Dado que a consideração do número de processadores correspondente a "muitos processadores" ou "poucos processadores" é de difícil quantificação (o que hoje é considerado um grande número de processadores na próxima década poderá não o ser) estas arquitecturas são caracterizadas pela sua organização de memória.

5.2.1 UMA's e DSM's

A primeira categoria de sistemas multiprocessador, as *Centralized Shared-Memory architectures* conectam (ou conectavam na década de 90) algumas dúzias de microprocessadores. Atendendo ao pequeno número de processadores envolvido, é possível a partilha duma única memória centralizada, conectando os processadores à memória através de um bus (ver figura 5.2.1. Com grandes caches, o bus e a memória (única) podiam satisfazer os requisitos do pequeno número de processadores. Por se tratar duma arquitectura com uma única memória central e um tempo de acesso uniforme para cada um dos processadores estas máquinas são muitas vezes designadas de UMA's (Uniform Memory Acess).

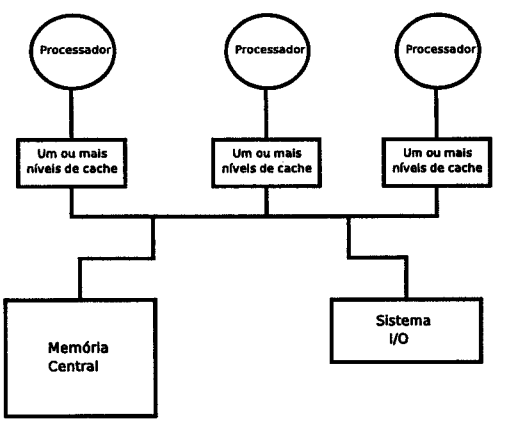


Figura 5.1: Modelo arquitectónico duma CSM

A segunda categoria de sistemas multiprocessador, as DSM's (*Distributed Shared-Memory architectures*), é constituída por máquinas cuja memória está fisicamente distribuída. Juntamente com a memória é igualmente distribuído um

sistema I/O , um processador e uma cache local. Cada uma destas unidades (memoria+processador+I/O) é designada por nó (ver figura 5.2). De modo a suportar a conexão de um grande número de processadores a memória tem de estar distribuída pelos diferentes nós, dado que a largura de banda para suportar os requisitos de tão grande número de processadores ligados a uma única memória centralizada é inaceitável.

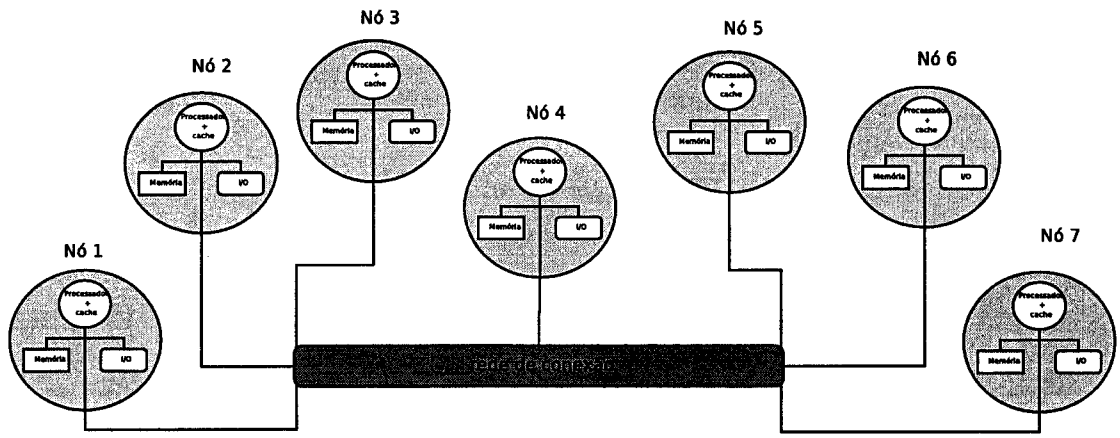


Figura 5.2: Modelo arquitectónico duma DSM

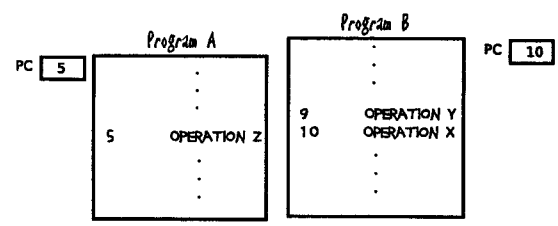
A distribuição de memória pelos nós duma DSM tem duas vantagens: primeiro, trata-se dum modo eficiente de escalar a largura de banda da memória, desde que a predominância dos acessos sejam feitos à memória do nó; segundo, existe uma menor latência dos acessos à memória local. A tendência actual é para termos processadores cada vez mais rápidos, que requerem mais largura de banda da memória e que apresentam menores latências de memória. Esta tendência faz as DSM's mais atractivas mesmo para sistemas com um não tão grande número de processadores, diluindo a escala de número de processadores apropriada a uma e a outra categoria.

A desvantagem desta arquitectura reside na comunicação de dados entre os diferentes processadores, já que agora, estas estão dependentes da rede de conexão subjacente à DSM, apresentando uma maior latência do que no caso das UMA's.

Uma das particularidades das DSM's é que apesar de existirem unidades de memória fisicamente distribuídas pelos diferentes nós, a memória é endereçada

como se de um único espaço lógico se tratasse. Tal significa que uma referência de memória pode ser feita por qualquer dos processadores para qualquer localização de memória desde que o processador possua o apropriado direito de acesso. O mesmo endereço físico em dois processadores corresponde assim à mesma localização de memória, daí o nome de "memória partilhada". Outra organização possível para o endereçamento de memórias distribuídas é tomá-las como privadas, considerando-as logicamente disjuntas e impossíveis de serem acedidas directamente por processadores remotos, este é o caso dos multicomputadores. Atendendo a que nas DSM's o espaço de endereçamento é partilhado, os endereços podem ser usados implicitamente na comunicação entre os processadores através das operações de *load* e *store*. No caso de máquinas com múltiplos espaços de endereçamento a comunicação de dados é obtida explicitamente por troca de mensagens entre os processadores envolvidos, por este motivo estas máquinas são usualmente designadas por máquinas de passar mensagens.

5.2.2 Modelos de Consistência



O assunto dos modelos de consistência prende-se com as garantias que os sistemas podem sobre a ordem de execução das operações dos programas. Esta garantia é importante visto os diferentes programas executarem-se em paralelo e possivelmente efectuarem partilha de dados. Considere-se o problema da figura anexa: Quando o processador responsável pela execução do programa A executa a operação Z, tem ou não conhecimento da execução da operação Y, realizada por outro processador que esteja a executar B? Atendendo a que o processador que executa B, já passou à instrução seguinte, parece lógico que a resposta seja sim, no entanto se a operação Y for realizada sobre uma localização de memória distinta da operação X, uma implementação pode permitir que a operação X se inicie sem que a operação Y esteja completada e portanto quando a operação Z for realizada o processador que a executa não tem conhecimento (ainda) da realização da

operação Y, ou das suas consequências.

As operações que interessam são obviamente as operações de escrita e leitura, visto ser através das escritas que os dados são modificados e através das leituras que os processadores vêm as alterações efectuadas pelos outros processadores.

Que propriedades devem ser forçadas entre as leituras e escritas feitas por um processador, para diferentes localizações de memória? Quando determinado processador passa à instrução seguinte (leitura/escrita) que garantias tem (e/ou pode dar) de que os restantes processadores têm conhecimento da instrução que acabou de realizar? Estas são as questões às quais os modelos de consistência dão resposta.

O modelo básico de consistência é a Consistência Sequencial. Sobre este modelo o resultado da execução dum programa (paralelo) é o mesmo como se os acessos de cada processador fossem executados ordenadamente e os acessos dos diferentes processadores fossem intercalados. A maneira mais simples de implementar a *CS* é exigir que um processo atrase a finalização dos acessos à memória até que todas as invalidações causadas por esse acesso estejam completas. Isto é equivalente a atrasar um acesso à memória até que o anterior acesso esteja completamente terminado. Sobre o modelo de consistência sequencial por exemplo, não é possível colocar a escrita no write buffer e continuar com uma leitura. A desvantagem deste tipo de consistência é a consequente redução de desempenho do sistema, sobretudo se o número de processadores envolvido é grande.

Outros modelos de consistência, cujo intuito é a melhoria do desempenho são passíveis de serem definidos, sendo menos restritivos nas ordens que impõem. O relaxamento de algumas das ordens afectam o modo como o programador vê a memória, e consequentemente a escrita de programas. Um modelo de consistência não estabelece quais as ordens que devem ser mantidas, antes define que o comportamento observado é o equivalente à manutenção dessas ordens.

Diz-se que um programa está sincronizado, se qualquer acesso a dados partilhados está ordenado por operações de sincronização. Existem dois tipos de operações de sincronização, uma respeitante à aquisição("acquire") da tranca("lock") do objecto, a outra à sua restituição ("release"). Todos os objectos têm uma só tranca.

Se diferentes processadores concorrem para obter a tranca dum objecto, um deles obtê-la-á ficando com uso exclusivo desse objecto até que a restitua, os outros processadores esperam até que a tranca esteja novamente disponível. A parcela de código compreendida entre a aquisição e a devolução da tranca, é designada por região crítica. Assim enquanto um processador estiver na região crítica respeitante a determinado objecto, o seu acesso àquele objecto é exclusivo. Uma referência aos dados está ordenada por uma operação de sincronização, se em qualquer das execuções possíveis, a escrita de uma variável por um dos processadores e um acesso à variável (para leitura ou escrita) por parte de outro processador, estão separadas por um par de operações de sincronização. Uma das operações de sincronização é feita após a escrita, a outra antes da leitura.

Casos em que as variáveis possam ser actualizadas sem a serialização imposta pelas operações de sincronização são chamados corridas de dados ("data races"), já que o resultado da execução pode depender de qual dos processadores "chega" primeiro, sendo o resultado não-determinista. Programas sincronizados são, por este motivo, chamados de livres de corridas de dados ("data-free-races").

É largamente aceite que os programas sejam sincronizados, e que a consistência pretendida fique nas mãos do programador. Este usará os mecanismos de sincronização ao seu dispor (providenciados pela linguagem) para implementar a consistência.

Modelos de Consistência relaxados

Pretende-se que programas sincronizados tenham sobre estes modelos a mesma semântica que sobre o modelo de consistência sequencial embora sem a ineficiência imposta por este modelo. Os diferentes modelos de consistência variam na restrições impostas às sequências possíveis entre as operações de leitura (Read) e escrita (Write)) e sincronização(Synchronized) processadas individualmente por um processador. As sequências básicas possíveis são,

R$\mapsto$R Leitura após leitura

R$\mapsto$W Escrita após uma leitura.

W$\mapsto$W Escrita após escrita

W$\mapsto$R Leitura após escrita

e as não básicas aquelas que envolvem acções de sincronização. Relaxar uma ordem ($X \mapsto Y$), significa que é permitido ao processador, iniciar a operação Y, sem a garantia de que operação X, esteja completamente terminada, mantê-la, significa que X é completada na totalidade antes de Y se iniciar. Os modelos de consistência não especificam quais as ordens que devem ser mantidas, significa que podemos assumir que a ordem foi cumprida (tal assunção diz respeito a comportamento dos programas), embora na prática as coisas não tenham de passar-se exactamente desse modo. No caso duma escrita, ela está completamente terminada quando todas as invalidações o estiverem.

Consistência de Processador ou "Total Store Ordering" (TSO)

Trata-se do modelo de consistência obtido quando relaxamos a ordem $W \mapsto R$. Tal significa que o processador pode iniciar uma operação de leitura sem que a operação de escrita realizada anteriormente esteja completamente terminada. O facto de a operação de escrita não estar completa significa que não são dadas garantias de que os outros processadores "vejam" a escrita realizada pelo processador. Fazendo uso do "write buffering" o processador pode passar à operação de leitura, diminuindo a latência das operações de escrita. A contrapartida da abordagem é que a garantia de a ordem que o sistema relaxa pode ser forçada usando operações de sincronização. O uso duma operação de sincronização entre a escrita e a leitura ($W..S..R$) permite assegurar a ordem já que as ordens $W \mapsto S$ e $S \mapsto R$ são asseguradas pelo modelo.

"Potal Store Ordering" (TSO)

Este modelo de consistência obtém-se relaxando também a ordem $W \mapsto W$. É agora permitido que escritas não conflituosas sejam executadas desordenadamente. Neste modelo só as ordens básicas $R \mapsto W$ e $R \mapsto R$ são mantidas. Só será obrigatório completar uma operação de escrita, se existir seguidamente uma operação de sincronização, já que $W \mapsto S$ é a única ordem mantida que contem W como primeira operação.

"Weak Ordering" É um modelo de consistência onde todas as ordens básicas estão relaxadas. Não há preservação da ordem salvo através das operações de sincronização.

Só podemos garantir de que uma dada operação de escrita/leitura seja completada, se esta operação figurar na ordem natural do programa, antes duma operação de sincronização. A garantia é dada pela preservação das ordem implícitas $R \mapsto S$ e $W \mapsto S$).

Uma operação de sincronização é sempre completada, antes de se iniciar uma escrita, leitura ou sincronização.

As vantagens inerentes ao facto de relaxarmos as ordens com reads na primeira operação (i.e. $R \mapsto X$) só serão sentidas em processadores que utilizem "nonblocking reads". Quando os processadores não têm esta facilidade forçam estas ordens visto não poderem prosseguir até ser completada a leitura. Mesmo em processadores com "nonblocking reads" as vantagens que advêm de relaxar estas ordens poderão não se sentir visto que quando ocorre um cache miss, pode ser impossível manter o processador ocupado durante o período de tempo que é necessário para o solucionar. Regra geral a vantagem dos modelos de consistência fracos, está em esconder as latências da escrita e não da leitura.

"Release Consistency" Neste tipo de consistência é feita uma distinção nas operações de sincronização. Há uma operação correspondente à aquisição ("acquire") do "lock" identificada por S_A e outra correspondente à sua libertação ("release"), S_R.

Este modelo de consistência assenta na observação de que uma operação de "acquire" precede a manipulação de dados partilhados, e uma operação de "release" sucede a uma actualização das variáveis partilhadas e precede também o próximo "acquire". Esta observação permite relaxar as ordens que se seguem antes do acquire e após o release. Assim numa release consistency os reads e writes que precedem um "acquire" não necessitam estar completados antes do "acquire" e os reads/writes após o "release" não necessitam esperar que este termine. As ordens agora mantidas são $S_A \mapsto R/W/S_A/S_R$, $S_R/W/R \mapsto S_R$ e $S_R \mapsto S_A$.

Consistência Java

O modelo de memória do Java(JMM) tal como descripto em [JG96] no capítulo referente a "Threads and Locks", especifica de modo operacional o modelo de consistência do Java. O JMM pressupõe a existência duma cache local a cada thread. As caches contêm cópias de trabalho das variáveis, existindo uma única cópia mestra das variáveis residente na memória principal. Os threads interagem com a cache através das operações *use* e *assign* e com a memória principal através das operações *load* e *store*. Estas operações têm a seguinte semântica:

- *use* obtem o valor da variável da cache para uso do "motor de execução do thread" ;

- *assign* atribui um novo valor, obtido do motor de execução, a uma variável previamente carregada na cache;

- *load* para obter da memória principal o valor da cópia mestra duma variável. Para tal a memória principal deve efectuar uma operação *read* que transmitirá o valor da variável ao thread, que o aceita e faz uma operação de load para carregar na cache o valor transmitido; A leitura duma variável para a cache é assim realizada pelo par de operações read/load, a primeira realizada pela memória principal a segunda pelo thread;

- *store* para transmitir à memória principal a actualização duma variável na cache. Esta operação lê o valor da cache e transmite-o à memória principal, esta por sua vez realiza uma operação *write* para actualizar a cópia mestra com o valor recebido. Tal como no caso das leituras, a escrita é também efectuada por um par de operações, desta vez store/write;

Cada objecto Java tem associado um lock, que está também ele na memória principal. O lock pode ser obtido por um só thread de cada vez. As operações de lock e unlock solicitadas pelos threads são tratadas pela memória.

A interacção entre variáveis e locks está definida no Java através de duas regras cuja transcrição informal passamos a apresentar:

- Se um thread vai realizar uma acção de release deve antes transmitir à memória central todas atribuições feitas às variáveis na cache;

- Se um thread vai realizar um acquire deve flushar todas as variáveis da cache. Os subsequentes usos de variáveis após o acquire devem corresponder a valores obtidos da memória central;

Como consequência, as referências feitas às variáveis após entrar num monitor, i.e na secção crítica, correspondem a valores actualizados, já que a cache foi fushada e portanto os valores são lidos da memória e colocados na cache do thread. Antes de libertar o lock as alterações efectuadas às variáveis que estão na cache são transmitidas à memória principal, sendo portanto visíveis para os outros threads.

A ordem de execução de determinadas acções está estipulada no Java do seguinte modo: as acções realizadas por um tread estão totalmente ordenadas, as acções realizadas pela memória para a mesma variável estão totalmente ordenadas, bem como as acções realizadas pela memória sobre determinado lock. Fica portanto por especificar a ordem entre as acções realizadas pela memória principal relativas a operações sobre diferentes variáveis, operações sobre diferentes locks e operações sobre locks e variáveis.

5.3 Programação concorrente em Java

A utilização de threads em Java(como parte integrante da API), juntamente com a utilização de monitores asseguram uma fácil escrita de programas concorrentes em Java. Um thread, como aliás tudo o resto em Java, é um objecto e a classe *java.lang.Thread* fornece os métodos para manipular estes objectos (inicializar, correr, suspender, etc.). A utilização de monitores visa proteger secções de código e evitar "race conditions", i.e. evitar, na medida do possível que o resultado dum programa seja não-determinista, em função de qual dos threads do programa realiza primeiro as suas operações.

A utilização de monitores é facultada pela palavra reservada "synchronized" que pode ser usada de duas formas:

- Como um statment, onde é especificada a referência ao objecto e o bloco de código a proteger (usualmente identificada por região crítica);

- como modificadora dum método, caso em que o objecto protegido, é aquele que dentro do método é identificado por *this* e a região crítica o corpo de método.

Cada objecto Java, tem um e um só "lock", sendo em ambos os casos adquirido o lock, executada a secção crítica e libertado o "lock".

5.3.1 Execução de Programas concorrentes em Java, em Clusters distribuídos

Esta secção foi elaborada recorrendo às publicações [A+01], [MMH98], [TKB01], [ABH+00], [AB01], [AH] sobre o Hyperion e a DSM-PM2 .

A noção de concorrência aparece no Java desde os primórdios, quer ao nível do utilizador quer ao nível do bytecode. A concorrência é facultada aos utilizadores através do uso de threads, que partilham um espaço de enderessamento comum. A livraria standard dos threads fornece as facilidades para manipular estes objectos, enquanto que o modelo de Memória do Java especifica o modo como os threads comunicam com a memória central. A execução de programas concorrentes num cluster pode ser linear: Os threads do programa podem ser mapeados nos threads nativos disponibilizados pelo cluster permitindo uma concorrência efectiva, o modelo de memória do Java pode ser implementado por uma camada DSM subjacente. Estas duas vertentes permitem a execução de programas concorrentes num cluster mantendo a semântica original da linguagem.

O hyperion é um sistema, desenvolvido na Universidade de New Hampshire, que permite a execução de programas paralelos em Java, através duma distribuição de threads pelos diferentes processadores dum cluster de workstations. A singularidade deste sistema é a utilização do cluster para executar uma única máquina virtual(JVM) possibilitando assim a execução transparente de threads do Java num ambiente distribuído. A implementação do hyperion, é baseada na PM2(parallel multithreaded machine). A interface de programação da PM2,

permite que threads sejam criados localmente e remotamente, e que comuniquem entre si através de RPCs (remote procedure calls). É também facultado pela PM2 um mecanismo de migração de threads, que lhes permite serem transparentemente transladados para outro nó durante a sua execução. No topo da PM2, a camada DSM-PM2 providencia uma plataforma para implementar protocolos de consistência DSM multihtraded, como por exemplo a consistência sequencial, ou release consistency. A camada DSM-PM2 foi usada para implementar a consistência Java descrita em [JG96]

De modo a produzir um executável para uma aplicação é primeiramente gerado o bytecode Java correspondente, usando um compilador standard de java. O bytecode dos ficheiros class é traduzido para C, usando o java2c, sendo posteriormente compilado o resultante código C, usando um compilador nativo de C do cluster e linkado à livraria runtime do Hyperion e a outras livrarias externas que sejam necessárias. O java2c usa uma abordagem simples para a tradução: Cada instrução da máquina virtual é individulamente traduzida para um statement C ou para uma invocação de macro correspondente.

O sistema de runtime do Hyperion está estruturado em subsistemas que permitem suportar a portabilidade para as arquitecturas alvo e simultaneamente testar diferentes técnicas de implementação dos componentes individuais:

- O subsistema de threads, providencia o suporte para "lightweight threads", no topo dos quais os threads Java são implementados. Este suporte inclui obviamente a criação e a sincronização de threads. Por razões de portabilidade a interface a este subsistema foi modelado nas funções principais providenciadas pelos threads POSIX. A migração de threads está também disponível graças ao suporte dado pela PM2;

- O subsistema de comunicações, suporta a transmissão de mensagens entre os diferentes nós do cluster.

- O subsistema de memória, é responsável pela alocação, gestão (incluindo sistemas de sincronização) e garbage collection dos objectos Java. As primitivas básicas que permitem implementar a consistência Java estão na tabela

da figura 5.3. Numa implementação distribuída estas primitivas devem ser asseguradas pela camada DSM subjacente.

Consistência Java

O compilador gera chamadas explícitas às primitivas `get/put` para aceder a objectos partilhados. Para implementar estas primitivas o Hyperion usa a DSM-PM2, uma camada DSM genérica multiprotocolo, construída no topo da PM2. A consistência Java requer o protocolo (MRMW - Multiple Reader, multiple Writer); isto é um objecto pode estar replicado e múltiplas cópias podem ser concorrentemente modificadas nos diferentes nós. A consistência é garantida através do uso de monitores. À entrada num monitor, a primitiva `invalidateCache` é chamada. Os objectos são lidos da memória usando o `loadIntoCache`. À saída dum monitor as alterações efectuadas a objectos na cache são transmitidas à memória principal usando a primitiva `updateMainMemory`. Anterior à chamada da primitiva `invalidateCache` à entrada num monitor ocoore uma chamada de `updateMainMemory`, sem a qual as alterações presentes na cache não seriam nunca reportadas à memória principal, o que violaria o modelo de consistência do Java(em [JG96] secção **Rules about Variables**) que especifica que deve ocorrer um *store* entre um *assign* e um posterior *load*(Informalmente tal significa que um thread não pode perder os seus mais recentes assignments).

Memória principal e caches

Para implementar o conceito de memória principal especificado no modelo de memória do Java, o sistema runtime do hyperion associa a cada objecto um "home node". Este nó fica encarregue de gerir a cópia mestra do objecto. Os objectos são inicialmente armazenados no seu home node sendo posteriormente replicados noutros nós quando forem acedidos remotamente. Em cada nó existe no máximo uma cópia dum objecto que pode será partilhada por todos os threads que correm nesse nó. Esta associação cache $\leftrightarrow$ nó em vez duma associação cache $\leftrightarrow$ thread evita um desperdício de memória, mas suscita algumas considerações:

Pré-ferching Uma das consequências de vários threads partilharem uma cache local, é que o pré-fetching é extendido para além das fronteiras dos threads, contrariamente ao especificado no JMM. Qualquer dos threads do nó podem fazer com que um objecto seja trazido para a cache, e o mesmo objecto pode ser acedido por qualquer dos threads que executa nesse nó. Existe portanto um pré-fetching do objecto em relação a determinado thread.

Outra situação de pré-fetching verifica-se quando ao carregar para a cache um determinado objecto, todos os outros objectos que partilham a(s) mesma(s) página(s), são carregados antecipadamente. Este carregamento antecipado é consistente com o modelo de consistencia Java, desde que seja realizado depois da última acção de lock. Como os locks são implementados com uma invalidação da cache, todos os carregamentos antecipados são descartados na altura do lock. Consequentemente todos os valores correspondentes a um 'cache-hit' correspondem a valores carregados após o último lock. Está também de acordo com as especificações do Java, o 'caching' de páginas em vez de objectos já que o que é exigido para obter um modelo concordante com a especificação do JMM, é que os valores modificados na cache, sejam transmitidos à memória principal, quando da realização de locks/unlocks.

sincronização Atendendo a que vários threads manipulam a mesma cache, são implementadas medidas cujo intuito é sincronizar acessos ou acções concorrentes por parte dos múltilplos threads que correm no mesmo nó. A utilização de mutexes permite tratar este tipo de situações.

Cada página que está na cache tem um mutex, que permitir serializar as actualizações ao bitmap de modificações da página(resultado das operações de *put*). Quando as modificações efectuadas na página são transmitidas ao home-node o mutex é também activado. As operações de *get* não são sensíveis ao mutex da página,uma vez que é perfeitamente compatível a leitura dum campo dum objecto e a actualização do bitmap, ou a leitura e a actualização do home-node do objecto. Também o próprio nó tem um mutex, que permite evitar invalidações de cache ou actualizações de

memória concorrentes (por parte dos vários threads). As invalidações de cache necessitam de mecanismos que permitam que as mesmas ocorram em alturas propícias aos threads que correm no nó e não solicitaram a invalidação. Não podem existir threads a aceder à cache quando esta é invalidada. Cada thread possui um 'reader lock' que será desactivado na prespectiva da cache ser modificada, i.e. quando o thread faz um lock ou um *new*. Este mecanismo adia uma invalidação de cache, por um thread estar à espera que uma página seja entregue, salvaguardando a anulação da entrega da página e um consequente pedido novo.

O sistema runtime do Hyperion

Num programa Java a única coisa que reside na memória heap são os objectos. Todos os tipos primitivos usados num método são guardados na stack do thread que executa o método. As referências aos objectos vão para a stack, mas o objecto em si está na heap.

Programas com threads escritos em Java assumem que existe uma única heap, consequentemente um objecto criado por um thread pode ser referenciado por outro thread da mesma aplicação. Por este motivo uma máquina virtual distribuída deve providenciar uma forma de endereçamento global de objectos. Isto é um objecto pode ser referenciado por qualquer thread do sistema independentemente da sua localização.

O Hyperion deve providenciar um ambiente de memória partilha ao programa Java. Um objecto criado num nó deve poder ser lido/escrito noutro nó distinto. Existe um só objecto real, mas poderão existir espalhados pelos nós múltiplas cópias do original. Quando um thread acede a um objecto desse nó, o objecto diz-se local, quando o thread acede a um objecto doutro nó, este diz-se remoto.

O design do Hyperion reflecte a ideia de que os objectos locais são claramente referenciados mais vezes do que os objectos remotos, e portanto o overhead introduzido por aceder a objectos remotos deve ser maior.

De modo a permitir as leituras/escritas de qualquer objecto do sistema o hyperion usa uma tabela centralizada de objectos em cada nó, sendo as referências

aos objectos índices da tabela. Cada entrada na tabela tem dois campos, um correspondente a um apontador para um objecto local, o outro, um apontador para o objecto remoto. Inicialmente estes apontadores são ambos inválidos. A cada nó corresponde uma parte específica de tabela, sendo o tamanho de cada parte determinada pela razão entre o tamanho da tabela e o número de nós da configuração.

Na criação dum objecto é alocada uma referência correspondente a uma entrada na parte da tabela correspondente ao nó onde será criado o objecto. Tal significa que a criação de objectos não provoca sincronização global. Na tabela de objectos do nó local, no índice correspondente ao objecto criado, o campo correspondente ao apontador para o objecto local conterá o apontador para a referência de memória alocada para aquele objecto. Um thread que aceda aquele objecto naquele nó, indexa a tabela correspondente à referência do objecto e usa o apontador local para o aceder. Nos outros nós o índice da tabela correspondente ao objecto não é alterado. Quando um thread noutro nó acede àquela referência de objecto, verifica o campo correspondente ao apontador para o objecto local e obtem um apontador inválido. Tal significa que o objecto pertence a outro nó. O sistema de comunicações obtém o objecto do nó apropriado cuja determinação depende da referência do objecto (p.e. numa tabela com 4 entradas($0 \ldots 3$) e numa configuração com 2 nós(0 e 1) as referências 0 e 1 pertencem ao nó 0 as referências 3 e 4 ao nó 1). Na tabela do nó remoto, na entrada correspondente à referência do objecto, o apontador para o objecto remoto conterá a localização de memória correspondente à cópia cache do objecto. Até ser invalidada esta cópia será usada pelos threads do nó em causa.

De modo a obter a consistência de memória , todas as escritas efectuadas sobre um objecto remoto devem ser registadas, de modo a mais tarde transmitir estas alterações ao nó local do objecto. Os objectos remotos têm um "bitfield" onde cada bit corresponde a um byte do objecto. Quando um byte do objecto é modificado o correspondente bit é activado, e na hora de transmitir as alterações só o bitfiled e o bytes modificados são transmitidos.

A consistência de memória é obtida pela utilização de monitores. Quando entra num monitor o thread deve transmitir as modificações efectuadas na cache

dos objectos remotos e "flushar" a memória remota. As próximas referências a objectos remotos obterão cópias actuais, usando novamente o processo descrito. Antes de sair do monitor só as alterações efectuadas a objectos remotos são transmitidas.

Objectos e páginas

Os objectos Java são implementados no topo das páginas DSM-PM2. Se um objecto abrange várias páginas, todas as páginas pelas quais o objecto se distribui são trazidas para a cache quando o objecto é carregado. Neste caso, outros objectos que não aquele que motivou o carregamento da(s) página(s), podem ser "antecipadamente" carregados se pertencerem às páginas trazidas. Esta situação de pré-fetching é compativel com o JMM, já que este modelo aceita antecipações das operações *read load* desde que sejam efectuadas depois do último *lock*. Dado que as operações de *lock* são implementadas com uma invalidadação de toda cache tem-se a compatibilidade assegurada.

Por outro lado também não existe o problema das actualizações efectuadas aos objectos previamente carregados na cache não serem transmitidas à memória principal, já que, tanto as operações de *lock* como as de *unlock* são implementadas actualizando o home-node de todos os objectos da cache que tenham sofrido alterações.

5.4 Conclusões

O Hyperion providencia uma forma clara de executar programas concorrentes em Java utilizando um cluster de computadores. Esta clareza advém do facto da execução duma única JVM sobre o cluster, permitindo que os múltiplos threads do programa sejam executados pelos difererentes processadores do cluster usando a PM2 e usando a a camada DSM-PM2 permite para implementar o modelo de memória(consistência) do Java.

O desempenho duma aplicação que corra sobre uma DSM está directamente relacionada com a frequência dos acessos aos dados. Estes podem ser de dois

tipos: locais ou remotos. Um acesso diz-se local se é referenciado um objecto cujo endereço corresponde a uma localização de memória local ao nó, caso contrário diz-se remoto. A utilização dum objecto remoto requer a sua presença na cache desse mesmo nó sendo consequentemente necessária a intervenção dos serviços da rede de interligação subjecente à DSM, no caso dum cache-miss. A invalidação duma cache requer que posteriores referências ao objecto requiram novamente o objecto do nó remoto para a cache local.

Os objectos partilhados, i.e. objectos que são acedidos pelos múltiplos nós da DSM ditam o desempenho duma aplicação:

- Se os objectos são partilhados só em leitura, as penalizações da sua utilização advêm das invalidações de cache;

- Se os objectos são partilhados em leitura e em escrita as penalizações da sua utilização advêm das penalizações inerentes às leituras, e da consistência pretendida. Sobre modelos de consistência relaxados categoria aos quais o JMM pertence, caso se pretenda que as alterações efectuadas sejam visíveis pelos processadores de outros nós há necessidade de proteger os acessos por acções de sincronização. Quem modifica faz uma operação sincronizada, que lê faz outra operação sincronizada. A implementação do Hyperion associa uma cache a cada nó e não uma cache a cada thread. Uma operação de "lock" provoca uma actualização para os respectivos home-nodes de todos os objectos que estejam na cache e que tenham sido modificados e um posterior inavalidadção de toda a cache do nó. Uma operação de "unlock" provoca uma actualização para o home-node dos objectos modificados entre as operações de "lock" e "unlock". A consistência é garantida dado que quem modifica assim como quem lê faz o par de operações lock/unlock;

Do aqui exposto resulta que o desempenho duma aplicação distribuída está fortemente relacionada com as operações de sincronização, invalidações de cache, percentagem de acessos remotos e partilha de dados efectuada. Evitar operações sincronizadas e tentar o mais possível que os acessos sejam predominantemente locais é a chave para um bom desempho duma aplicação distribuída sobre uma DSM.

Um modelo do AJACS distribuído será adaptado uma DSM se se respeitarem estas regras. Se a exploração do espaço de soluções for entregue aos vários processadores que figuram na DSM, processando cada um deles estados que sejam locais ao nó correspondente a adaptabilidade do modelo está garantida. Tal é possível visto que no modelo exposto no capítulo 4 para explorar uma parte do espaço de soluções basta ter um estado, expandi-lo, coleccionar os estados resultantes repetindo o processo. Se o estado com que cada processador inicia o processo for um objecto local os estados derivados também podê-lo-ão ser, já que derivam dos anteriores por alteração de alguns dos seus valores. Numa DSM com N nós o espaço de soluções pode ser percorrido por N processadores em paralelo desde que se "fabriquem" N sub-estados que serão processados em paralelo por cada um dos processadores (ver figura 5.4). Evitar mudanças de contexto, i.e. que um processador processe estados doutros processadores (consequentemente objectos não locais) poderá garantir um bom desempenho. A única restrição para esta imposição será a obrigatoriedade de manter uma quota de paralelismo, sendo necessário decidir o que fazer quando um processador expande toda uma sub-árvore ficando sem mais estados para continuar a processar. Estas questões serão abordadas nos capítulos 6 e 7.

`loadIntoCache`	Carrega na cache um objecto
`inavlidateCache`	Invalida todas as entradas na cache
`updateMainMemory`	Actualiza a memória principal, com as modificações presentes na cache
`get`	Devolve um campo dum objecto previamente carregado na cache
`put`	Modifica o campo dum objecto previamente carregado na cache

Figura 5.3: Primitivas chave DSM

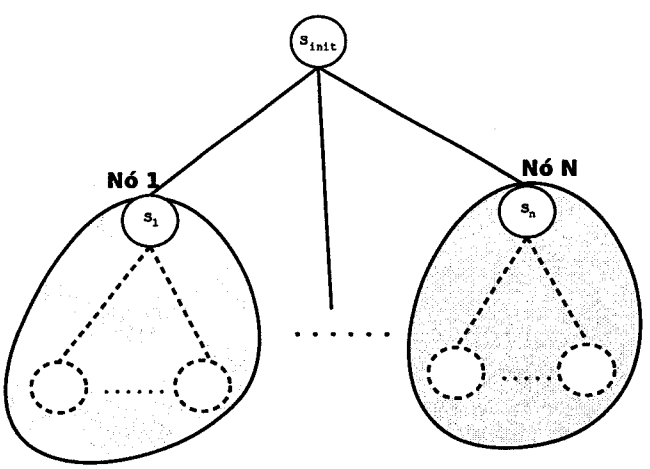


Figura 5.4: Pesquisa paralela

Capítulo 6

Especifcação e análise duma implementação distribuida do AJACS

Estruturar as implementações distribuídas do AJACS, definindo o modo como os diferentes threads que realizam a pesquisa comunicam entre si e analisar o desempenho de algumas aplicações sobre um cluster de computadores, utilizando o suporte fornecido pelo Hyperion é o objectivo deste capítulo.

6.1 Introdução

O modelo computacional distribuído do AJACS foi concebido para que uma implementação paralela num sistema distribuído apresente um bom comportamento. Da caracterização do modelo podemos concluir que:

- A resolução dum CSP é obtida no AJACS, gerando e atravessando a árvore de estados do problema;
- Os estados, figura central do modelo, são estruturas que encerram em si toda a informação necessária para que o processo de pesquisa seja autónomo e auto-suficiente a partir desse pedaço de informação, i.e. a partir de um estado é possível gerar a sub-árvore de estados derivados desse estado. A estrutura do espaço de soluções é definida por estados alternativos, independentes entre si, e que partilham com o seu antecessor informação só em leitura.
- As soluções são estados/folha desta árvore de pesquisa.

A nossa expectativa é de que se vários agentes operarem em simultâneo para realizar a pesquisa, sendo a cada um deles atribuída uma parte da árvore para pesquisar, parte essa que é função do estado sobre o qual vão trabalhar, teremos um melhor desempenho do que numa situação em que um só processo esteja encarregue de percorrer toda a árvore de estados do problema.

A independência entre os diferentes threads que executam uma aplicação estará assegurada reduzindo a comunicação entre os mesmos. A comunicação é introduzida através de objectos partilhados e a sua redução conduzirá à pretendida indepêndicia.

Sendo o AJACS um toolkit de programação por restrições em JAVA, e fornecendo o JAVA facilidades para a criação e manipulação de agentes de execução, vulgo "threads", a implementação paralela do modelo utilizando estes agentes e a sua execução num sistema distribuído, são o rumo natural a tomar.

As restantes secções deste capítulo apresentam: a implementação paralela do modelo na secção 6.2, na secção 6.3 são apresentados os resultados da execução

duma implementação paralela do Ajacs sobre um ambiente distribuído usando o Hyperion, na secção 6.4 é avaliado o desempenho do Hyperion através da execução de benchmarks fornecidos com o sistema, sendo finalmente na secção 7.6 retiradas as conclusões.

6.2 Implementações distribuídas do AJACS

A implementação paralela do modelo, baseia-se na utilização de "threads" do Java que efectuarão as travessias na árvore de soluções. A(s) soluções são determinadas utilizando dois tipos de threads: O controlador e os trabalhadores.

A resolução dum problema será obtida usando um thread controlador e um ou mais trabalhadores. O papel de controlador é gerir os seus trabalhadores, cabendo a cada um destes a tarefa de dar um passo do processo de pesquisa que globalmente permitirá atravessar o espaço de soluções. Um passo do processo de pesquisa corresponderá a:

1. Expandir um estado, gerando os estados derivados desse estado, e verificar se alguns destes estados derivados é ou não solução?

   (a) Se sim, o trabalhador notifica o controlador de que foi encontrada uma solução, e caso o problema em causa seja determinar apenas uma solução, uma ordem do controlador faz terminar os trabalhadores finalizando a pesquisa.

   (b) Caso nenhum dos estados derivados seja solução ou o problema em causa seja determinar todas as soluções, o processo repete-se sendo executado novo passo da travessia, agora sobre outro estado.

Existem dois modos distintos de realizar as travessias, dependendo de qual o estado escolhido para realizar o passo de travessia seguinte:

- O estado escolhido é um dos estados resultantes da expansão efectuada no passo anterior;

- O estado escolhido é um qualquer estado que esteja por expandir.

O significado prático destas escolhas é que, no primeiro caso, o trabalhador usa um dos estados resultantes da sua última expansão para expandir no passo seguinte, e no segundo caso, o trabalhador usa um qualquer estado que tenha sido derivado por si próprio ou por qualquer dos outros trabalhadores, em qualquer dos passos de travessias anteriores.

Em ambos os casos e porque um trabalhador deriva um estado de cada vez e gera por expansão um número indefinido de estados, existe a necessidade de coleccionar os estados derivados. Uma vez mais foram escolhidas duas abordagens distintas para coleccionar os estados: uma estrutura centralizada, no controlador, ou uma estrutura local no próprio trabalhador. A gestão centralizada de estados será abordada na secção 6.2.1 a gestão local de estados na secção 6.2.2.

6.2.1 Gestão Centralizada dos estados

A gestão centralizada é implementada usando um estrutura de dados no controlador (p.e. uma lista) que coleccionará os estados resultantes das expansões de cada um dos trabalhadores. O controlador tem associada uma variável *to_visit*[], que colecciona os nós da árvore que *ainda não foram visitados*, nem estão a ser visitados por nenhum dos trabalhadores. Os trabalhadores acedem à colecção do controlador para obterem um estado (sobre o qual possam realizar um passo da pesquisa), e para adicionarem novos estados (os remanescentes da pesquisa). Para o efeito, o controlador disponibilizará os métodos para manipular a colecção:

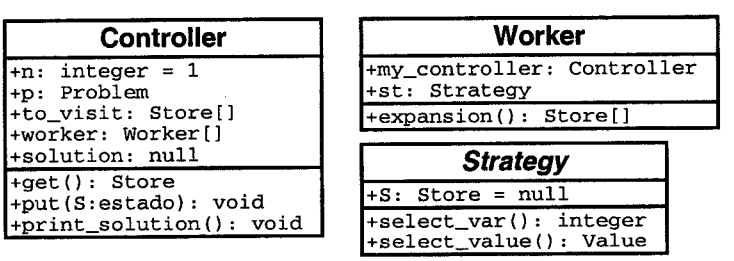


Figura 6.1: Estrutura das classes intervenientes na gestão centralizada

get(), permite obter um dos estados da colecção, *put(S)* permite adicionar um novo estado à colecção.

O método *print_solution*, permite ao controlador listar o resultado da pesquisa. Os algoritmos de controlador ("Controller") e do trabalhador ("worker") são apresentados respectivamente nas figuras 6.2 e 6.3.

$put(p.S_{init})$
FOR $i = 1$ TO n
    $worker[i] = new\ Worker(this, strat).start()$;
WHILE $< \neg\ finish >$ DO
    wait
$printSolution()$

Figura 6.2: Algoritmo do Controlador

O controlador inicia o seu ciclo de vida adicionando o estado inicial do problema ao conjunto de estados a visitar, e criando n trabalhadores controlados por si próprio. Todos os trabalhadores actuam do mesmo modo no processo de pesquisa possuindo por isso a mesma estratégia (referenciada no algoritmo do controlador por *strat*). Esta definirá o modo como cada trabalhador determina a variável a reduzir e o modo de realizar essa redução.

Após iniciar os seus trabalhadores o controlador ficará em espera até que termine a pesquisa. Assuma-se, por agora, que o significado da condição $\neg\ finish$ em ambos os algoritmos é esse.

Cabe aos trabalhadores percorrerem o espaço de soluções. Para tal vão processando sucessivamente estados, fazendo a sua expansão. Designa-se por expansão de um estado o conjunto de estados obtidos por aplicação da estratégia ao estado que está a ser processado. Estes estados são sempre sujeitos a propagação. A expansão é realizada utilizando o método *expansion* da classe trabalhador, os métodos *select_var* e *select_value* e o estado S da estratégia conjuntamente com o problema, para realizar a propagação destes estados, permitem implementar neste método a expansão de acordo com a definição 22 do capítulo 4.

Atendendo a que pretendemos encontrar uma solução o mais cedo possível[1], todos os estados da expansão são analisados, i.e. verifica-se se os estados são

[1]Embora o problema referido seja sempre o de determinar uma solução as propostas apre-

```
WHILE < ¬ finish > DO
    IF < st.S is NULL >
    THEN st.S = my_controller.get()
    let X = expansion()
    st.S = NULL
    FOREACH e in X DO
      IF < e is solution >
      THEN finish
      ELSE IF < st.S is NULL >
             THEN st.S = X.get()
             ELSE my_controller.put(e)
END
```

Figura 6.3: Algoritmo do Trabalhador

ou não solução. Omitir esta análise induziria um atraso no reconhecimento das soluções, por parte do sistema: Se um dos estados da expansão for uma solução e tal não for reconhecido, não existe nenhuma garantia de que este estado-solução seja o próximo estado do trabalhador. Esse estado poderá ser adicionado à colecção de estados em espera, não sendo possível determinar quando será atribuído a um dos trabalhadores.

Não foi feita nenhuma imposição às funções *get/put* do controlador. Como a colecção de estados em espera não possui nenhuma solução, a implementação da estrutura de dados *a_visitar* poderá seguir uma disciplina qualquer, por exemplo uma fila ou uma pilha. O tipo de estrutura escolhida não é determinante para a eficiência devendo ser implementadas as soluções mais simples.

Também, e porque um dos estados da expansão será o próximo estado a ser processado pelo trabalhador, uma função *get*, que devolva um dos estados da expansão é necessária. Qual dos estados eleger é discutível: Não sabemos à priori o local da árvore onde está a solução e portanto impor uma caminho não faz

sentadas são facilmente generalizáveis ao problema de encontrar todas (ou apenas algumas) das soluções.

sentido. No entanto algumas heurísticas podem ser consideradas para melhorar a pesquisa. É por exemplo possível "medir" a distância à solução. Um estado que gere muitas expansões estará potencialmente mais longe da solução do que um estado que gere poucas (um estado solução não gera quaisquer expansões). A função

$$h(S) = \sum_{i:\, \neg ground(i)} \#v(i)$$

permite contar as expansões possíveis de um estado. Este número é um limite superior para o número de expansões, já que algumas delas poderão por propagação degenerar em estados inconsistentes. Usando esta função, um estado com uma variável por iterar e cujo domínio tenha cardinalidade 10, por exemplo, está mais longe da solução do que um estado que possua 2 variáveis por iterar mas em que o cardinal de ambos os domínios seja 2 por exemplo ($2+2<10$). A distância à solução é neste caso medida pelo número máximo de estados que as futuras expansões do estado podem gerar. Esta, e outras funções que para um estado devolvam um valor que possa ser interpretado como a distância à solução, poderão ser usadas para escolher o estado mais "promissor" de todos os estados da expansão, condicionando assim o estado devolvido pela função *get*.

Com a estrutura imposta e pelos algoritmos descritos, o processo de pesquisa tem neste contexto o seguinte desenrolar: é criado um controlador C, sendo dados como parâmetros um número inteiro correspondente ao número de trabalhadores e o problema a solucionar: $C = new\ Controller(i, P)$. O controlador será activado fazendo $C.start()$, o que iniciará o algoritmo do controlador com as consequências anteriormente expostas.

Os trabalhadores iniciam o seu ciclo de vida sem nenhum estado para processar. Neste momento o único estado conhecido é o estado inicial do problema que foi adicionado pelo próprio controlador ao conjunto de estados a visitar. Os n trabalhadores tentarão obter do controlador um estado. Um deles ganhará a permissão de acesso ao controlador e obterá o estado inicial.

Poderá dar-se o caso, não só no início mas em qualquer fase da pesquisa, que um trabalhador ao aceder à lista do controlador esta esteja vazia. Existem dois motivos para tal acontecer: já não há mais estados a processar ou a lista

está vazia temporariamente. Desde que existam ainda trabalhadores a processar é possível que a lista seja novamente aumentada pelos estados produzidos pelos trabalhadores activos.

É necessário redefinir a comunicação entre os diferentes trabalhadores. Um trabalhador necessita saber se os outros agentes estão a processar, caso em que terá de esperar, ou se estão também à espera, caso em que terminará a pesquisa. Cada trabalhador possuirá três estados possíveis, a trabalhar ("work"), à espera ("wait") ou terminou ("finish") . O controlador deverá providenciar um método que permita a um trabalhador saber se os restantes trabalhadores estão à espera. É um método de fácil implementação bastando contar os trabalhadores cujo estado é de espera, e compará-lo com o número de trabalhadores existentes.

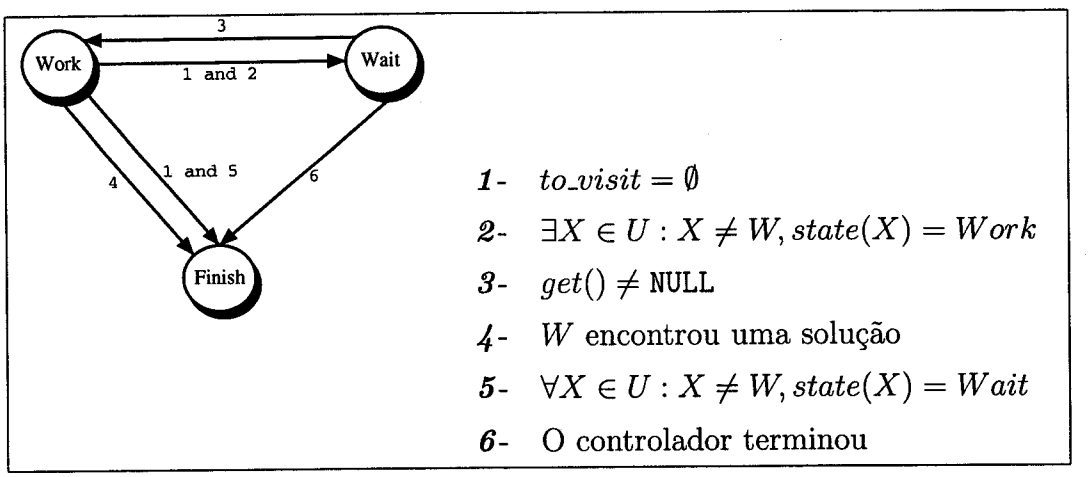


Figura 6.4: Diagrama de transição de estados de um trabalhador W, no modelo de gestão centralizada.

A figura 6.4 mostra as possíveis transições de estado por parte de um trabalhador e as acções que induzem as mesmas[2]:
Um trabalhador transita do estado *work* se ao tentar obter de *to_visit* um estado para processar tal não for possível (por a lista estar vazia). A transição dá-se

[2]Assume-se que o trabalhador mantém o seu estado se nenhum dos acontecimentos que provocam uma transição ocorrer. Nas acções que desencadeiam as transições de estado, considera-se U o universo dos trabalhadores.

para o estado *wait*, se existir algum trabalhador que não ele próprio a "trabalhar" (i.e. um trabalhador no estado *work*). Caso contrário, (i.e. todos os outros trabalhadores estão no estado *wait*), transitará para o estado *finish*. Outra possível transição do estado *work* para o estado *finish* dá-se quando o trabalhador encontra uma solução.

Um trabalhador só abandona o estado *wait* por um dos dois motivos:
- Porque o controlador terminou, caso em que o trabalhador também irá terminar passando ao estado *finish*;
- Porque a lista possui novos estados, caso em que o trabalhador obtém o seu estado e passa ao estado *work*.

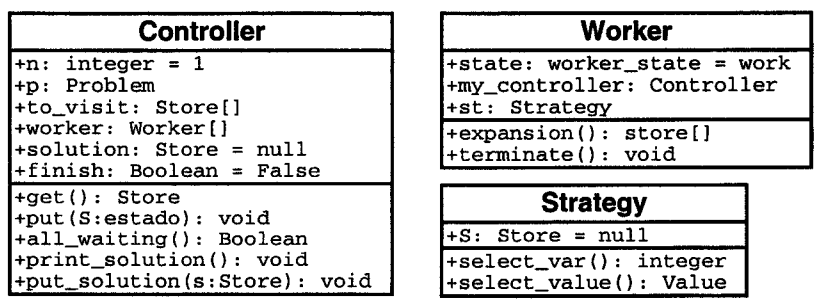


Figura 6.5: Estrutura de classes usadas na gestão local

O algoritmo do controlador é exactamente igual ao apresentado na figura 6.2. Agora o significado da condição $\neg finish$ é inequívoco: O controlador tem associado a variável $finish$, cujo valor será modificado para *true*, quando for encontrada uma solução ou quando um trabalhador que não consegue obter um estado do controlador verificar que todos os outros trabalhadores estão à espera.

O algoritmo do trabalhador sofre agora algumas alterações que visam sobretudo sincronizar a actividade dos diferentes trabalhadores e evitar situações de "deadlock".

A execução dos algoritmos do controlador e dos trabalhadores das figuras 6.2 e 6.6 garante agora a sincronização necessária para efectuar a pesquisa e encontrar um solução, caso exista, ou concluir que o problema não tem solução. Como referido anteriormente o controlador introduz na lista de nós a visitar o estado inicial do problema, e inicia os seus trabalhadores. Os trabalhadores quando

iniciam o seu algoritmo estão a trabalhar (no estado "work"). Todos os trabalhadores,à excepção daquele que acede com sucesso à lista do controlador e obtém o estado inicial, poderão, ao aceder à lista encontrá-la vazia e obter como resultado do acesso o objecto `NULL`. Todos os acessos que decorrerem no espaço de tempo que medeia a aquisição do estado inicial e a adição dos estados resultantes da expansão do estado inicial, têm esta consequência .

Em qualquer fase da pesquisa, um trabalhador ao aceder à lista do controlador pode obter:

- `Null`, caso em que recorre ao controlador através do método *all_waiting* para saber se todos os outros trabalhadores estão também à espera. Se sim, este trabalhador usa o método *terminate* para se terminar a si próprio e ao controlador. Na listagem da solução por parte do controlador, e uma vez que não lhe foi comunicada nenhuma solução, este conclui que a mesma não existe. É o caso em que o espaço de soluções foi varrido na totalidade;

- `um estado` válido para processar. Neste caso após a expansão desse estado um desses estados será eleito como o próximo estado. Caso o expansão resulte num conjunto vazio o trabalhador terá na próxima iteração que recorrer à lista do controlador.

Só um dos trabalhadores (aquele que encontra a solução ou que termina por todos os outros estarem à espera) e o controlador, terminam com as respectivas variáveis de "fim" activadas (*finish* no primeiro caso e *state* no segundo), os restantes trabalhadores terminam o seu ciclo de vida por contingência do controlador terminar.

6.2.2 Gestão local dos estados

Neste modelo de gestão, todos os trabalhadores possuem uma estrutura de dados local,*my_list*, que colecciona os estados por expandir e produzidos em anteriores expansões do próprio trabalhador. Existe um só estado inicial que será atribuído a um dos trabalhadores, todos os outros trabalhadores iniciar-se-ão com a respectiva lista vazia. De modo a determinar qual dos trabalhadores está mais

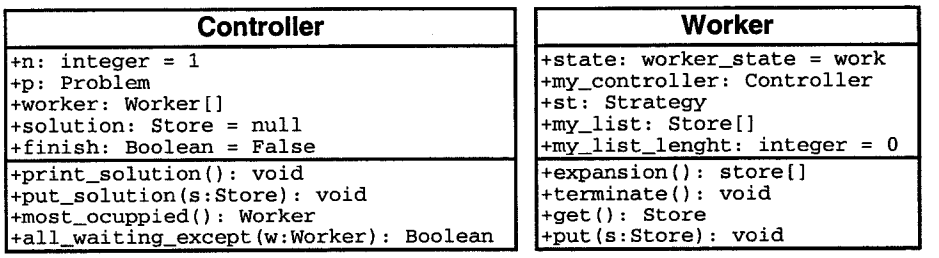
```
WHILE < state isn't finish and my_controller isn't finish > DO
    IF < st.S is NULL >
    THEN let T = my_controller.get()
        IF < T isn't NULL >
            state = work
            st.S = T
            let X = expansion()
            st.S = NULL
            FOREACH e in X DO
              IF < e is solution >
              THEN my_controller.put_solution(e)
                   terminate()
              ELSE IF < st.S is NULL >
                   THEN st.S = X.get()
                   ELSE my_controller.put(e)
        ELSE IF my_controller.all_waiting()
             THEN terminate()
             ELSE state = wait
END
```

Figura 6.6: Algoritmo do trabalhador na Gestão Centralizada

ocupado, o controlador possui o método *most_occupied* que determinará qual dos seus trabalhadores possui o maior valor na variável *my_list_lenght.* Quando o comprimento das listas é, em todos os trabalhadores, 0, o valor de retorno do método deverá ser *NULL.* O método *all_waiting(w)* devolverá *true* se todos os trabalhadores à excepção do próprio w estiverem no estado a trabalhar (*"work"*).

Os métodos *get, put, get_from* permitem respectivamente, obter um estado da lista, adicionar um estado à lista, e obter um estado da lista de outro trabalhador. Estes métodos devem actualizar de forma adequada os valores correspondentes ao comprimento das listas.

Os algoritmos do controlador e do trabalhador para este modelo de gestão



figuram respectivamente em 6.7 e 6.8.

FOR $i = 1$ TO n
    IF $<$ i is 1 $>$
    THEN $worker[i] = new\ Worker(this, strat, p.S_{init}).start();$
    ELSE $worker[i] = new\ Worker(this, strat).start();$
WHILE $< \neg\ finish >$ DO
    wait;
$printSolution();$

Figura 6.7: Algoritmo do Controlador

O processo de pesquisa, tal como no modelo anterior, inicia-se com a criação e activação dum controlador com dois parâmetros: o problema e o número de trabalhadores. O controlador iniciará os seus trabalhadores. Todos, à excepção dum deles, iniciarão o seu ciclo de vida com uma lista vazia de estados para processar. Todos iniciarão o seu ciclo de vida no estado a trabalhar.

O controlador passará como parâmetro de criação o estado inicial do problema a qualquer um dos seus trabalhadores (sem perda de generalidade foi escolhido o primeiro trabalhador no algoritmo 6.7). Este trabalhador cuja lista não é vazia expandirá de modo habitual o estado e adicionará os estados que não forem solução, à sua própria lista privada de estados.

Independentemente de produzir ou não novos estados para a sua lista, continuará a executar novas iterações do seu algoritmo desde que quando reinicie não tenha entretanto sido descoberta nenhuma solução por nenhum dos outros trabalhadores. Pode acontecer que um trabalhador ao determinar qual dos seus

```
WHILE < state isn't finish and my_controller isn't finish > DO
    IF < my_list isn't empty >
    THEN st.S = get()
    ELSE let w = my_controller.most_occupied()
        IF < w is NULL >
        THEN IF < all_waiting_except(this) >
            THEN terminate()
            ELSE state = wait
        ELSE st.S = get_from(w)
    IF < st.S isn't NULL >
    THEN let X = expansion()
        st.S = NULL
        FOREACH e in X DO
            IF < e is solution >
            THEN my_controller.put_solution(e)
                terminate()
            ELSE put(e)
```

Figura 6.8: Algoritmo do Trabalhador

"colegas" tem a maior lista (para lhe passar trabalho), todas as listas estejam vazias.

Por exemplo, quando do início da pesquisa, e no espaço de tempo entre retirar da lista o estado inicial e gerar novos estados para a sua lista, todos os trabalhadores têm as suas listas vazias. Neste caso existe necessidade de verificar se todos os trabalhadores estão ou não à espera. Se sim, a pesquisa tem de terminar porque já não existem mais estados para processar, se não, o próprio trabalhador entrará no estado de espera, visto existir pelo menos um trabalhador que não ele próprio a processar um estado, existindo portanto possibilidades da pesquisa continuar.

Um trabalhador sairá do estado de espera, quando obtiver para a sua estratégia um estado diferente de *NULL* (a função *get_from* encarregar-se-á de ac-

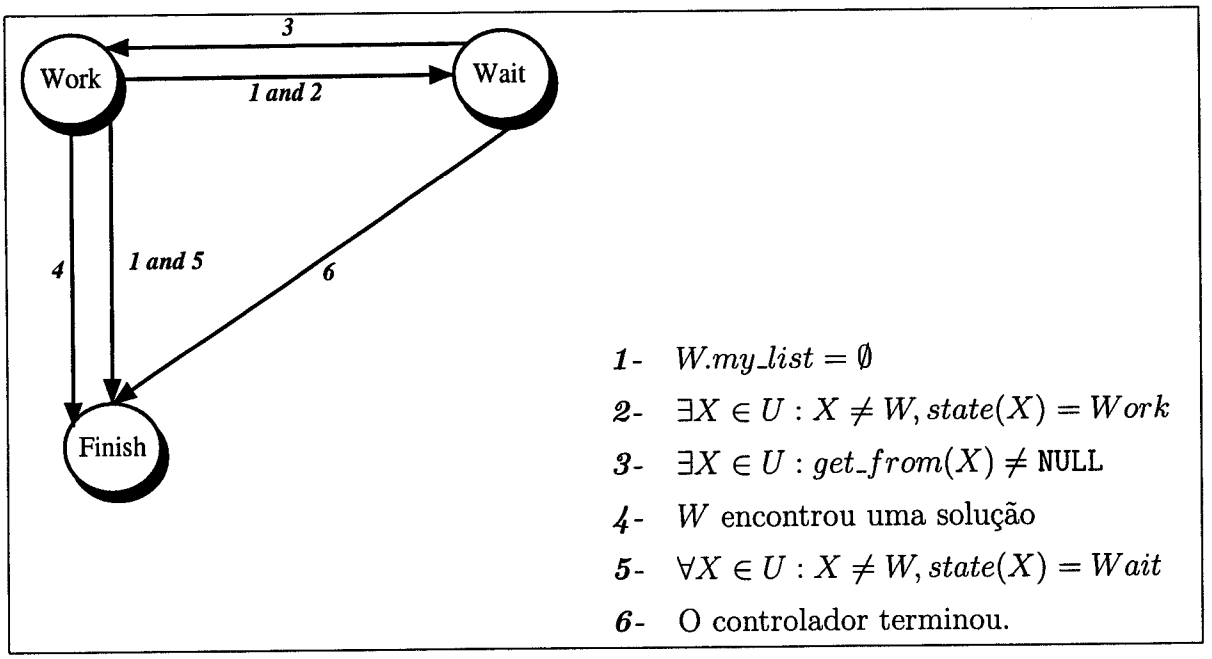


Figura 6.9: Diagrama de transição de estados de um trabalhador W, no modelo de gestão privada. Assume-se que o trabalhador mantém o seu estado se nenhum dos acontecimentos que provocam uma transição ocorrer.

tivar correctamente o estado do trabalhador). A função *get* não necessita activar o estado do trabalhador, uma vez que só a contingência de um trabalhador ter vazia a sua lista, e necessitar de obter através de outro trabalhador um estado para processar o fará entrar no estado à espera.

As transições de estado para o modelo estão apresentadas na figura 6.9 e são sensivelmente idênticas às do modelo anterior, embora com as particularidades inerentes à gestão privada.

6.3 Resultados da execução do AJACS sobre o Hyperion

De modo a avaliar o comportamento da(s) implementações distribuídas foi construída uma pequena aplicação. A aplicação escolhida, foi, pelos motivos já apresentados de escalabilidade, simplicidade e reduzido número de restrições, as N-

Queens.

A construção de aplicações que utilizem threads está no contexto do AJACS, bastante simplificada. Seja P o problema (cuja construção é análoga à do AJACS sequencial), w o número de threads trabalhadores que irão procurar a(s) soluções e p um parâmetro que especifica se o problema em causa é determinar uma ou todas as soluções. A criação dum thread controlador juntamente com a aplicação do método "start" desencadeia o processo de pesquisa paralela.

```
new Controller_GL(P,w,p).start();
```

De modo a analisar o desempenho dos modelos (Gestão Centralizada/Gestão Local) foram construídos e testados ambos. Os testes realizados visam sobretudo concluir sobre as potencialidades que a versão paralela possa apresentar e discutir o comportamento de ambos modelos.

Os testes foram realizados num cluster LINUX com 4 bi-processadores, inicialmente 4×2 Celeron a 433MHz, sendo posteriormente actualizados para 4 × Pentium 4 com HyperThreading a 2.8G.

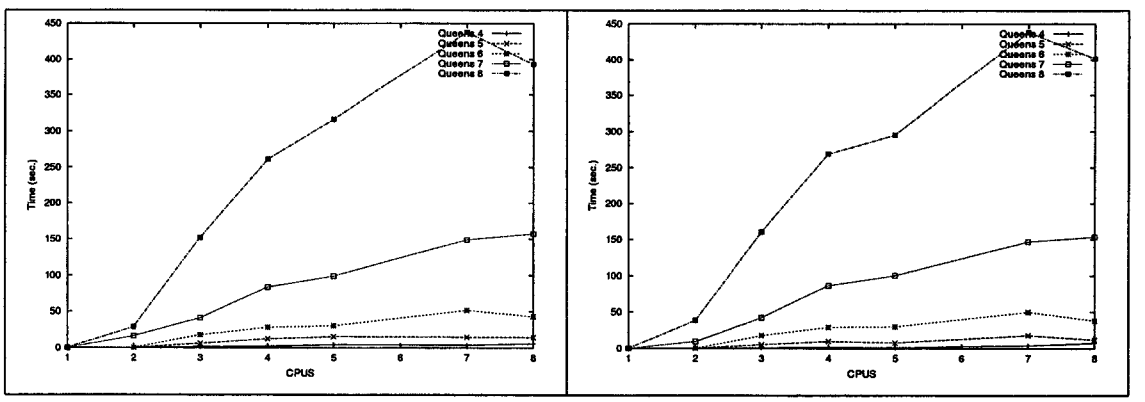


Figura 6.10: Tempos de execução do programa Queens, cálculo de todas as soluções usando 12 trabalhadores.

Os gráficos da figura 6.10 ilustram alguns dos resultados coleccionados. O gráfico mais à esquerda mostra os resultados obtidos com o modelo de gestão global, sendo o da direita respeitante ao modelo de gestão local. Em ambos os gráficos estão representados 6 problemas, respectivamente as Queens 4, 5 6 7 e

Queens

N°CPUS	4	5	6	7	8
1	0.004	0.012	0.034	0.151	0.569
2	0.141	0.169	0.151	7.191	7.544
2*	0.379	0.716	0.680	26.051	50.243
3	1.509	6.236	18.011	40.918	152.132
4	2.157	10.600	25.410	81.119	224.703
4*	2.632	14.459	31.021	86.017	296.983
5	4.515	15.700	30.509	98.842	316.394
7	4.060	15.078	51.477	149.035	438.149
8	5.965	14.699	43.009	157.398	393.059

Figura 6.11: Tempos de execução dos programas Queens 4, 5, 6, 7 e 8, para o modelo de gestão global.

8, sendo o problema solucionado o da determinação de todas as soluções, usando 12 trabalhadores.

As medições(em segundos) que estão na base da construção dos gráficos apresentam-se nas tabelas das figuras 6.11 e 6.12. Nas configurações correspondentes ao mesmo número de CPUS(2 e 4), o primeiro caso refere-se a configurações que utilizam as duas CPUS do mesmo nó, e no segundo caso as CPUS pertencem a nós distintos e são assinaladas com *.

O desempenho do AJACS contradiz as expectativas que o modelo sugeriria. Em nenhuma situação um aumento do número de CPUS nas configurações do Hyperion, se traduz numa redução do tempo de execução. Mais ainda o desempenho de ambos os modelos é similar (i.e. nenhum dos modelos tem um desempenho superior ao outro), o que contraria a hipótese que o modelo de gestão privada, seria mais eficiente que o modelo de gestão global, pelo que existem outros factores que estão a ocultar a variação expectável de desempenho, nomeadamente a técnica utilizada para IPC no Hyperion (se TCP, se UDP, VIA ou outra). Este assunto será abordado nas secções e capítulos seguintes.

	Queens				
N°CPUS	4	5	6	7	8
1	0.004	0.016	0.048	0.160	0.598
2	0.190	0.226	0.143	3.307	18.779
2*	0.455	0.640	0.737	16.300	58.468
3	1.254	5.561	18.066	42.356	160.554
4	1.531	7.933	31.568	85.845	233.077
4*	2.335	12.133	27.289	87.383	304.290
5	1.368	8.079	30.211	100.322	295.256
7	4.155	18.252	49.973	146.786	438.577
8	7.819	12.262	38.458	153.669	401.537

Figura 6.12: Tempos de execução dos programas Queens 4, 5, 6, 7 e 8, para o modelo de gestão local.

6.4 Análise do desempenho de benchmarks do Hyperion

Um dos benchmarks fornecidos com a distribuição do Hyperion é o conhecido problema do caixeiro viajante (TSP). Na implementação facultada, o problema em causa é o da determinação do valor da rota de custo mínimo que contemple um determinado número de cidades. Uma tabela de distâncias é fornecida e são geradas todas as possíveis rotas, designadas por Jobs, num total de 2184. Estas figuram numa lista centralizada à qual os diferentes threads que executam o programa acedem para ir buscar trabalho (um Job). Cada thread processa iterativamente um Job e caso a rota processada tenha um custo menor que o melhor valor encontrado até ao momento, este valor é actualizado. Esta variável(o mínimo)é obviamente partilhada pelos diferentes threads em execução tendo a sua consistência de ser garantida. Também a tabela de distâncias é partilhada, mas como se trata dum objecto acedido só em leitura, o objecto é localizado (criando uma variável local ao método *run* que referencia a tabela). Outro processo usado

	Número de threads por nó						
N°Nós	1	2	4	8	16	32	64
1	–	–	–	3.34	3.94	4.74	6.68
2	–	–	5.25	5.70	6.30	7.60	–
4	–	6.1	7.2	8.5	10.5	–	–
8	5.60	9.6	14.0	71.0	–	–	–

Figura 6.13: Tempos de execução do problema TSP, em diferentes configurações com diferentes números de Threads.

para localizar os objectos é a criação dum novo, cópia do anterior. Este processo é aplicado aos Jobs quando acedidos pelos threads que os processam: é criado um novo objecto, cópia do Job retornado da lista e sobre o qual o thread processará. Este processo permite obviamente os acessos evitando as subjacentes intervenções da rede para trazer do home-node os objectos processados pelos threads, por motivos das invalidações de cache.

A tabela 6.4 mostra os resultados da execução do problema variando o número nós da configuração e o número de threads que a executam, os tempos apresentados estão em segundos.

O comportamento observado através dos tempos de execução revela que existe um problema com a configuração do Hyperion, que não nos permite obter um melhor desempenho por aumento do número de nós das configurações.

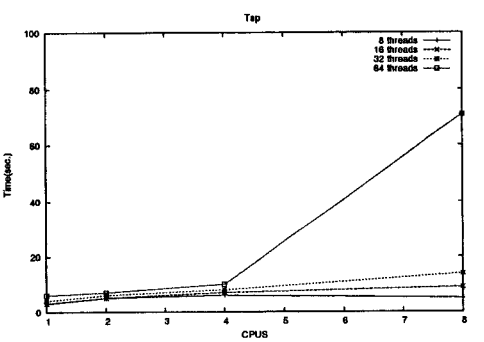


Figura 6.14: Gráfico dos tempos de execução para o problema TSP

6.5 Ainda as Queens8

De modo a testar o impacto dos acessos partilhados sobre o Hyperion, foi desenvolvido um modelo das Queens sem qualquer partilha. Inicialmente são criados tantos estados iniciais quantos os threads que executarão o programa, mas localmente em cada thread.

Para o programa testado, as Queens-8 realizadas com 8 threads, cada trabalhador i (`w(i)`) cria um estado cujo valor no índice 0 é i e explora a sub-arvore correspondente a esse estado. Assim o trabalhador 1 explorará 93 estados e encontrará as 4 soluções cuja primeira rainha está na primeira coluna. O trabalhador 2 explorará 98 estados e encontrará as 8 soluções correspondentes a posicionar a primeira rainha na segunda coluna, etc. O número de estados processados e o número de soluções encontradas por cada trabalhador estão na tabela da esquerda da figura 6.16.

Cada trabalhador gera o seu problema, a sua estratégia, a sua lista de soluções e a sua lista de estados a processar, lista esta que conterá inicialmente o estado acima descrito e todos os estados derivados da exploração local dum estado. Todos estes objectos são locais ao método *run*. Quando um thread esgota a sua lista de estados, termina, esperando por todos os outros de modo a contabilizar o tempo de execução. A sincronização dos threads para que iniciem a exploração de seu espaço de soluções todos ao mesmo tempo e seja possível contabilizar o tempo de execução é realizada com uma barreira.

A distribuição dos threads pelos nós da configuração faz-se "round-robin". Se uma configuração tem 1 nó todos os 8 trabalhadores correm nesse nó (nó 0). Se uma configuração tem 2 nós (nó 0 e nó 1) os trabalhadores w(2),w(4), w(6) e w(8) correm no nó 0 e os trabalhadores w(1),w(3),w(5) e w(7) correm no nó 1. As restantes distribuições dos threads do programa pelos nós das configurações estão na tabela da direita da figura 6.16.

Na figura 6.17 é mostrado o tempo de execução e a percentagem de carga em cada nó (relativamente aos estados processados pelos trabalhadores afectos a esse nó).

Worker	# estados	# soluções
w(1)	93	4
w(2)	98	8
w(3)	82	16
w(4)	65	18
w(5)	65	18
w(6)	82	16
w(7)	98	8
w(8)	93	4

Figura 6.15: Estados processados e soluções encontradas

Nós	w(1)	w(2)	w(3)	w(4)	w(5)	w(6)	w(7)	w(8)
1	0	0	0	0	0	0	0	0
2	1	0	1	0	1	0	1	0
3	1	2	0	1	2	0	1	2
4	1	2	3	0	1	2	3	0
5	1	2	3	4	0	1	2	3
6	1	2	3	4	5	0	1	2
7	1	2	3	4	5	6	0	1
8	1	2	3	4	5	6	7	0

Figura 6.16: Distribuição dos trabalhadores pelos nós das configurações.

Tempo	0	1	2	3	4	5	6	7
0,23	100%							
7,4	50%	50%						
5,9	24%	38%	38%					
6,2	23%	23%	27%	27%				
6,15	10%	26%	29%	26%	10%			
6,9	12%	28%	28%	12%	10%	10%		
8,25	14%	28%	14%	12%	10%	10%	12%	
10,1	14%	14%	14%	12%	10%	10%	12%	14%

Figura 6.17: Tempos de execução e percentagens da distribuição do trabalho pelos nós das configurações

6.6 Conclusões

A utilização do Hyperion, permite executar programas em Java com threads, num cluster de computadores. Conceptualmente a ideia é simples: a livraria do Java fornece os métodos necessários para a criação e manipulação de threads, o modelo de memória do Java especifica como os threads comunicam com a memória principal. Se os threads Java forem mapeados em threads nativos das máquinas do cluster(por uma questão de eficiência) e se o modelo de memória do Java for implementado por uma camada DSM, então do ponto de vista do utilizador do Hyperion, o cluster é uma única JVM, e as diferentes máquinas que o constituem são simplesmente recursos a utilizar para executar o seu programa com threads num ambiente verdadeiramente paralelo, atingindo este objectivo de forma transparente.

Dos testes realizados relativamente aos modelos centralizado e local, podemos concluir que não são observáveis diferenças significativas entre os modelos. A explicação para esta constatação advêm do facto de em os ambos casos os acessos poderem ser remotos, i.e. as listas ditas privadas ou locais dos trabalhadores de gestão local, não são necessariamente objectos locais ao thread que executa a pesquisa. Um acesso é local se o objecto está no nó onde corre o thread,

caso contrário é remoto. Embora seja possível saber qual o nó onde corre um thread usando primitivas do sistema "runtime" do Hyperion o mesmo não é possível relativamente ao home-node dum objecto. Observa-se que o "overhead" do sistema é da ordem dos 80 a 90% o que significa que há muito a fazer ao nível das camadas inferiores (da DSM, nomeadamente) para obter um desempenho satisfatório.

Sobre o Hyperion, o home-node dum objecto é o nó onde estiver a correr o thread que o criou (correspondente a uma invocação do método *new*). Os trabalhadores são objectos e o seu home-node corresponderá ao nó onde correr o thread que os cria. As listas dos trabalhadores são objectos e o seu home-node é determinado da mesma forma. Os trabalhadores são também threads, a execução dum thread corresponde ao código implementado no método *run*, se dentro deste método forem acedidas as variáveis de instância do objecto, que é o caso das listas "privadas", não há qualquer garantia da localidade das mesmas. A única vantagem relativamente à lista global do controlador é que os acessos às listas locais não necessitam ser sincronizados para manter a consistência já que um só trabalhador lhes acede. Para que esta forma de acesso seja sinónimo de aumento de desempenho, será necessário que a camada DSM subjacente saiba reconhecer a situação de independência entre os acessos efectuados em diferentes nós.

De modo a avaliar o desempenho duma aplicação onde seja necessário realizar alguma partilha de variáveis foi testado um benchmark fornecido com a distribuição do Hyperion, o TSP(ver secção 6.4). Também aqui o comportamento do Hyperion é similar ao apresentado pelo modelos do AJACS implementados: não são obtidos quaisquer speedups com a introdução de nós nas configurações, o que parece indicar problemas com a configuração ou instalação do Hyperion.

Para avaliar o impacto da partilhada(não partilha) foi elaborado um novo teste, usando novamente o problema das Queens-8 (ver secção 6.5). Neste teste a única variável partilhada é uma barreira para a sincronização dos threads. Uma vez mais o comportamento do Hyperion é similar ao já observado: não existem quaisquer speedups, e o comportamento observado não pode ser imputado ao excesso de partilha ou à invalidação de caches. Note-se que em [A+01] é apresentado um gráfico de speedups para o problema "Graph-coloring" realizado com

64 threads com eficiências na ordem dos 90%. Trata-se também dum problema similar ao TSP com partilhas de variáveis e pools de jobs. Não foi possível confirmar este benchmark do Hyperion no nosso cluster dado não ser distribuído com a implementação.

Embora os acessos sincronizados tenham necessariamente de sofrer penalizações na execução sobre uma DSM, não parece ser a partilha que motiva os desempenhos apresentados por todas as aplicações testadas. Repare-se que todas as aplicações têm um forte peso na manipulação de objectos e um fraco peso nos cálculos que executam (operações sobre inteiros ou vírgula flutuante).

Por insuficiência de meios não foi possível testar transportes com menor latência. Seria uma forma de aumentar o desempenho do PM2, logo do Hyperion e portanto do AJACS . A utilização do M-VIA, uma implementação livre do protocolo VIA (Virtual Interface Architecture) que visa oferecer comunicação numa SAN (System-Area Network) mais rápida do que a que advém do uso dos protocolos TCP/IP e UDP/IP, seria uma solução por apresentar latências muito reduzidas frente a estas. Para mais informação sobre o VIA pode-se consultar por exemplo [CR02].

Capítulo 7

Estratégias de pesquisa no AJACS: Outra abordagem distribuída.

A definição/utilizção de estratégias permite no AJACS configurar a árvore de estados do problema e consequentemente modelar a pesquisa. Neste capítulo é apresentado um modelo distribuído para o AJACS, que assenta na parametrização da árvore estados utilizando as estratégias induzindo uma pesquisa paralela com reduzida comunicação entre os diferentes agentes que a executam.

7.1 Introdução

Uma estratégia usa um estado e configura uma subárvore de estados cuja raiz é o estado da estratégia. A configuração da árvore de estados é definida:

- no nível, pela partição considerada para determinado valor;
- no nível imediatamente inferior, pela escolha do valor a particionar.

Considerando a expansão dum estado como o conjunto de estados derivados por aplicação da estratégia ao estado e propagados, os estados da expansão são na árvore de estados do problema os filhos desse estado. Será sobre estas árvores que é efectuada a pesquisa. Atribuir a cada agente que efectue a pesquisa um estado, e deixá-lo trabalhar esse estado gerando novos estados (e novas subárvores) é definir a pesquisa no AJACS.

As diferentes configurações que é possível obter para a árvore de estados dum mesmo problema, conduzem a diferentes pesquisas. Neste capítulo vamos apresentar modelos de pesquisa motivados pelas estratégias, sendo no secção 7.2 revisto o conceito de estratégia, em 7.3 são apresentadas as dependências entre estratégia e pesquisa, com a apresentação dos modelos de pesquisa paralela e sequencial. Na secção 7.4 é apresentada uma estimativa para o trabalho dum agente e um nova proposta para a pesquisa sequencial, sendo na secção 7.5 apresentado o modelo OO dos agentes. Finalmente na secção 7.6 são apresentadas as conclusões.

7.2 Estratégias

Uma estratégia (ver no capítulo 4, definições 17 e 18 as definições formais) aplica-se a um estado e definirá de modo inequívoco os estados derivados da aplicação da estratégia. As definições da variável seleccionada e da partição são suficientes para construir os estados derivados. Estes constroem-se deixando inalterados os domínios correspondentes às variáveis que não a seleccionada e assumindo a variável seleccionada, nos estados resultantes, os valores correspondentes aos subdomínios obtidos da partição. Uma estratégia é assim definida através duma

função de selecção e duma partição possível para o domínio correspondente ao valor seleccionado, sendo estas as matérias das secções seguintes.

7.2.1 Função de Selecção

É possível escolher uma multiplicidade de valores a reduzir, podendo uma estratégia seleccionar qualquer dos valores do estado cuja cardinalidade seja superior a 1. Seja γ_0 a estratégia que toma como valor a reduzir o primeiro valor de cardinalidade maior do que 1, seja γ_1 a estratégia que toma como valor a reduzir o valor de maior cardinalidade. É de esperar que o nível da árvore correspondente às expansões motivadas pela aplicação de uma e outra estratégia sejam diferentes num caso e noutro, embora possam ocasionalmente coincidir (o primeiro valor pode ser o de maior cardinalidade), mesmo que ambas as estratégias coincidam no modo como realizam as partições a árvore de estados resultante será distinta num caso e noutro. A escolha de valores duma estratégia apenas tem de garantir que enquanto houver valores a reduzir, há um que é escolhido.

7.2.2 Definição das Partições

As partições são a outra componente que caracteriza a estratégia. Suponha-se que pretendemos uma estratégia para efectuar partições de determinada dimensão. Seja s o estado da estratégia e seja n a dimensão da partição. Uma possível implementação desta estratégia poderá escolher o primeiro valor cuja cardinalidade seja superior ou igual a n, e fazer n-1 partições com valores singulares e uma com os restantes valores do domínio. Outras implementações são possíveis, por exemplo escolher um domínio com uma cardinalidade múltipla da dimensão da partição, e gerar n domínios de cardinalidade mais homogénea. Qualquer que seja a implementação considerada é necessário decidir como é realizada a partição no caso de todos os domínios terem uma cardinalidade inferior à dimensão da partição, as abordagens seguintes são possíveis, embora nem sempre o segundo caso garanta uma cardinalidade fixa, já que podem não existir mais valores para particionar:

- no domínio de maior cardinalidade e a partição ficará com uma dimensão inferior ao pretendido, visto não ser possível obter uma partição de cardinalidade superior usando um só domínio;

- usa-se mais do que um domínio, o que corresponde a seleccionar outra variável e efectuar as partições sobre as novas variáveis até obter a cardinalidade pretendida.

7.2.3 Exemplos

Seja $\lambda_0 = (\delta_0, \{\gamma_{01}, \ldots, \gamma_{0n}\})$ a estratégia definida por uma função de selecção em que o valor seleccionado para efectuar a partição é o primeiro cuja cardinalidade seja superior a 1, e as funções de redução particionam o domínio em n domínios singulares considerando n a cardinalidade do valor seleccionado.

$$\delta_0(V) = \min_i \{i : \#v_i > 1\}$$

$$\gamma_{0j}(\{x_1, \ldots, x_n\}) = \{x_j\}, \quad j = \{1, \ldots, n\}$$

A aplicação da estratégia λ_0 ao estado s, definido por 3 variáveis de domínios $\{1, ..., 3\}$,$\{1, ..., 5\}$, e $\{1, ..., 10\}$, i.e., $s \equiv \Big(\{1, \ldots, 3\}, \{1, \ldots, 5\}, \{1, \ldots, 10\}\Big)$, é dada pela colecção de estados

$$\lambda_0(s) = \Big\{\Big(\{1\}, \{1, \ldots, 5\}, \{1, \ldots, 10\}\Big), \Big(\{2\}, \{1, \ldots, 5\}, \{1, \ldots, 10\}\Big),$$

$$\Big(\{3\}, \{1, \ldots, 5\}, \{1, \ldots, 10\}\Big)\Big\}$$

O conjunto $\lambda(s)$ é constituído pelos três estados que derivam da partição do domínio da variável 1, em três subdomínios. Tendo todas os domínios cardinalidade superior a 1, escolhe-se o primeiro, ver fig 7.1.

Seja agora $\lambda_N = (\delta_1, \{\gamma_{11}, \ldots, \gamma_{1N}\})$ a estratégia definida por uma função de selecção em que o valor seleccionado para efectuar a partição é o de maior cardinalidade, e as funções de redução particionam o domínio em exactamente N

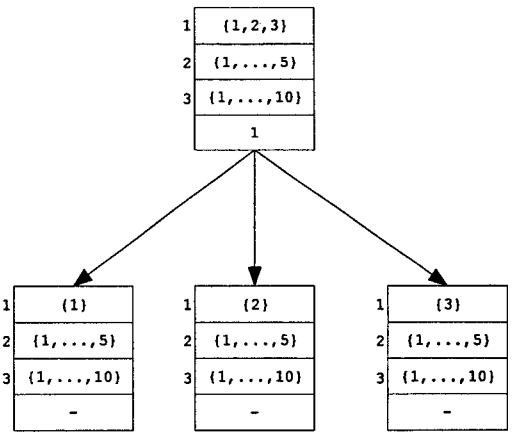


Figura 7.1: Exemplo de aplicação da estratégia λ_0 a $s \equiv \Big(\{1,\ldots,3\},\{1,\ldots,5\},\{1,\ldots,10\}\Big)$

(caso seja possível) subdomínios, sendo os primeiros $N-1$ subdomínios singulares, e o N-ésimo subdomínio constituído pelos restantes valores.

$$\delta_1(V) = \min_i \{i : \#v_i \geq \#v_j, \forall j\}$$

$$\begin{aligned} &\gamma_{1j}(\{x_1,\ldots,x_l\}) = \{x_j\}, && j = \{1,\ldots,N-1\} \\ &\gamma_{1N}(\{x_1,\ldots,x_l\}) = \{x_N,\ldots x_l\} \end{aligned}$$

A aplicação da estratégia $\lambda_{N=4}$ ao mesmo estado s, definido na secção 7.2.3 é agora constituída por constituído pelos 4 estados, que derivam de escolher o valor $\{1,...,10\}$ para ser particionado em 4 subdomínios. O resultado da aplicação desta estratégia pode ser visualizado na figura 7.2.

Suponha-se agora que pretendemos particionar um domínio eleito em M subdomínios, com cardinalidades equiparadas, isto é se queremos particionar em 4 subdomínios um domínio de cardinalidade 20 por exemplo, queremos 4 domínios de cardinalidade 5 e não 3 domínios de cardinalidade 1 e um de cardinalidade 17, como na estratégia definida anteriormente. Seja $\lambda_M = (\delta_1, \{\gamma_{21},\ldots,\gamma_{2M}\})$ tal estratégia, as funções de partição para λ_M são agora dadas por:

$$\begin{aligned} &\gamma_{2j}(\{x_1,\ldots,x_l\}) = \{x_{(j-1)a+1},\ldots,x_{ja}\}, \quad j = \{1,\ldots,M-1\}, a = \lfloor l/M \rfloor \\ &\gamma_{2M}(\{x_1,\ldots,x_l\}) = \{x_{(M-1)a+1},\ldots x_l\} \end{aligned}$$

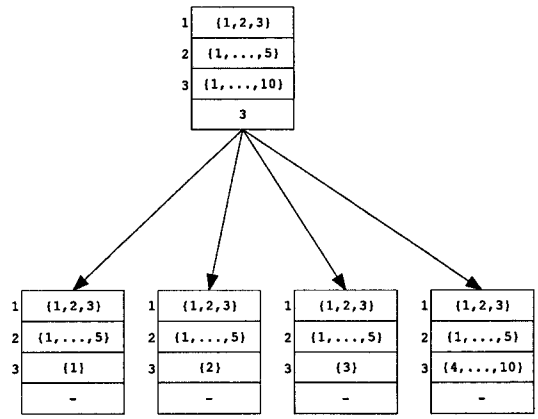


Figura 7.2: Aplicação da estratégia $\lambda_{N=4}$

O resultado da aplicação da estratégia $\lambda_{M=4}$ ao estado
$s \equiv \Big(\{1,\ldots,3\},\{1,\ldots,5\},\{1,\ldots,10\}\Big)$,
está na figura 7.3.

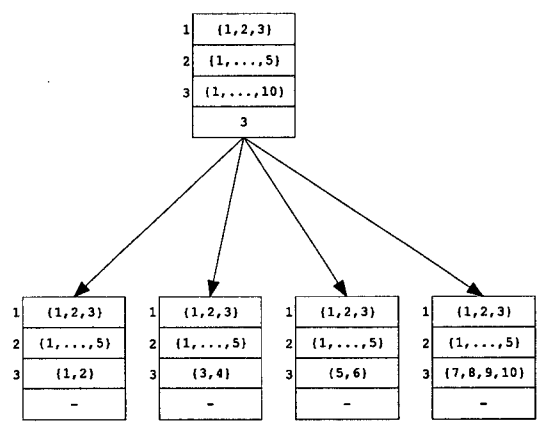


Figura 7.3: Aplicação da estratégia $\lambda_{M=4}$

7.3 Estratégias e Pesquisa

Definindo a estratégia a configuração de árvore de estados do problema, e sendo este o espaço de soluções, a pesquisa pode ser realizada de dois modos distintos: sequencialmente ou em paralelo por múltiplos agentes.

7.3.1 Pesquisa Sequencial

Na pesquisa sequencial o agente percorre a árvore de estados tomando iterativamente um estado. Em cada iteração o agente acede a uma lista local, retira um estado, expande-o, e colecciona os estados resultantes da expansão, que não sejam soluções. Estas, à medida que são detectadas vão para uma lista de soluções. A pesquisa do agente termina quando a lista de estados a processar está vazia. O algoritmo desta pesquisa está apresentado na figura 7.4.

```
Algoritmo THS(e,p, L, Sol)
{input: e uma estratégia, p um problema, L uma lista da estados,
Sol uma lista de soluções}
WHILE < L ≠ ∅ > DO
    let s' = L.get()
    let X = expansion(s, e, p)
    FOREACH x in X DO
        IF < x is solution >
        THEN add(x, Sol)
        ELSE add(x, L)
```

Figura 7.4: Algoritmo dum agente sequencial

7.3.2 Pesquisa Paralela

A pesquisa paralela é conseguida fazendo com que subárvores da árvore de estados sejam expandidas em simultâneo por vários agentes. Um modo simples de o conseguir é lançar um agente paralelo por cada estado da árvore que não seja solução: O processo da pesquisa paralela "ilimitada" é desencadeado lançando um agente paralelo(ver algoritmo 7.5) com: uma estratégia, o problema a solucionar, o estado inicial do problema e a lista de soluções. O agente expande o estado e da lista de estados resultantes retira as soluções e lança $n - k$ agentes paralelos (para n a cardinalidade da expansão e k o número de soluções).

Algoritmo THP(e,p,s, Sol)
{input: e uma estratégia, p um problema, s um estado,
Sol uma lista de soluções}
let $X = expansion(s, e, p)$
FOREACH x in X DO
IF $<$ x is solution $>$
THEN $add(x, Sol)$
ELSE THP(e,p,x,Sol)

Figura 7.5: Algoritmo dum agente paralelo de quota ilimitada

Esta abordagem tem como vantagens, além da simplicidade, a ausência de comunicações entre os agentes. Como desvantagens temos ausência de controlo sobre o numero de agentes que realizam a pesquisa. Situações em que este número seja imcomportável para o sistema são fáceis de idealizar, já que os CSP's são problemas NP completos. Um problema com 10 variáveis e domínios de cardinalidade 10, poderá ter uma árvore com 10^{10} estados. Uma estimativa para o número de agentes que operam em simultâneo é dificil de obter, já que existe uma sequencialidade no modo como são lançados, mas será certamente um número elevado. Outra desvantagem será não ser possível medir o grau de paralelismo visto não ser conhecido o número de agentes.

Outra abordagem possível à pesquisa paralela será pré-fixar o número de agentes que a implementarão, seja N esse número (N poderá corresponder ao número de CPU's disponíveis na configuração de hardware sob a qual a pesquisa será realizada). Cada um destes agentes terá permissão para "reproduzir-se" em N' sub-agentes distribuindo equitativamente (na medida da expansão) o seu trabalho pelos seus sub-agentes, após o que terminará. Caso um agente não possua quota de "reprodução", iniciará uma pesquisa sequencial. De modo a garantir que a pesquisa é executada por um número limitado de agentes, (no máximo N), é necessário especificar as relações entre N e N'. Seja f_N uma

distribuição de N, nas seguintes condições:

$$f_N = \{x_1, \ldots, x_k\} : \forall j, 1 \leq x_j < N \wedge \sum_{j=1}^{k} x_j = N$$

Por exemplo temos para $N = 5$, são 3 as distribuições possíveis, $\{1,1,1,1,1\}$, $\{2,1,2\}$, $\{3,2\}$. O algoritmo dum agente paralelo *THP*, é dado na figura 7.6, e a sua execução processa-se como se descreve de seguida: Seja n, a quota de reprodução do agente, e s o seu estado. Se n é igual a 1, o agente não pode reproduzir-se pelo que terminará, lançando previamente um agente sequencial em que a lista de estados a processar é constituída unicamente pelo estado do agente paralelo. Se o agente tem quota de reprodução (i.e. se $n>$ 1), calcule-se uma distribuição de n, e faça-se uma expansão do estado do agente em k estados, sendo k a cardinalidade da distribuição. Note-se que a cardinalidade dos dois conjuntos é à priori a mesma (aqui designada por k), embora tal possa não acontecer visto na expansão não figurarem estados que resultem em falha. Neste caso faz-se uma nova distribuição mas com cardinalidade fixa e igual à cardinalidade da expansão. O algoritmo da figura 7.4, apresenta uma implementação possível.

É portanto possível obter uma distribuição de cardinalidade igual à expansão, havendo uma correspondência biunívoca entre um inteiro da distribuição e um estado da expansão. Esta correspondência permite associar uma quota e um estado nos sub-agentes gerados. No algoritmo da figura 7.6 tal significa que à entrada do ciclo $\#F = \#X = m$ sendo $1 \leq m \leq k$, ou $\#F = k \wedge \#X = 0$. Neste último caso, por não haver elementos em X o ciclo não se executa (nem deve!).

De modo tentar assegurar que a quota de um agente não se perca, no caso dum estado ser solução é distribuída a "sua" quota pelas quotas dos restantes agentes. Seja F^i a distribuição actual correspondente à iteração de s_i. F^i obtem-se da distribuição anterior por:

$$F^i = \begin{cases} F^{i-1} \backslash x & \text{se } s_i \text{ não é solução;} \\ redistribut(F^{i-1} \backslash x, x) & \text{se } s_i \text{ é solução;} \end{cases} \quad \text{para } x = first(F^{i-1})$$

Tomando $F^0 = \{x_1, \ldots, x_m\}$ como a distribuição à entrada do ciclo, a última distribuição será $F^{m-1} = \{x_m^{(m-1)}\}$. Considera-se o índice $(m-1)$, para diferenciar

este valor de x_m(se ocorrem distribuições na construção dos F^is, estes valores são diferentes).

Seja

$$Peso(F^i) = \sum_{x_j \in F^i} x_j$$

então $Peso(F^0) = n$ (a quota inicial do agente). A relação entre o peso duma distribuição e a anterior é dada por:

$$Peso(F^i) = \begin{cases} Peso(F^{i-1}) - x & \text{se } s_i \text{ não é solução;} \\ Peso(F^{i-1}) & \text{se } s_i \text{ é solução;} \end{cases} \quad \text{para } x = first(F^{i-1})$$

Se o último estado da expansão, s_m é solução, a quota perdida é dada por $Peso(F^{m-1})$, podendo este valor coincidir com n no caso de todos os estados da expansão serem soluções:$Peso(F^{m-1}) = Peso(F^{m-2}) = \ldots = Peso(F^0) = n$

7.3.3 Exemplos

Suponhamos que pretendemos aplicar uma pesquisa paralela, na resolução do problema das Queens, de dimensão 8. O estado inicial(s_0) deste problema é um estado com 8 valores todos iguais a $[1, 8]$. Suponhamos que pretendemos solucionar o problema usando 5 agentes, e que λ_M será a estratégia usada. Crie-se um problema p_Q, cujo estado inicial é s_0, e adicionem-se a p_Q as restrições *NoAttack*. As soluções de p_Q serão o resultado da execução $THP(\lambda_M, p_q, s_0, 5, SOL)$. Uma distribuição de 5, f_5 poderá ser $\{2, 2, 1\}$. Neste caso temos uma distribuição de cardinalidade 3, e portanto faz-se uma expansão de s_0 usando este valor para a parametrização da estratégia, ($\lambda_{M=3}$). A expansão é constituída por $\{s_1, s_2, s_3\}$, sendo portando lançados $THP(\lambda_M, p_Q, s_1, 2, SOL)$, $THP(\lambda_M, p_Q, s_2, 2, SOL)$, e $THP(\lambda_M, p_Q, s_3, 1, SOL)$, terminando $THP(\lambda_M, p_Q, s_0, 5, SOL)$. O trace desta execução está na figura 7.10. Se por exemplo s_1 fosse uma solução a sua quota de agentes (2), seria distribuída sobre s_2 e s_3 e seriam lançados $THP(\lambda_M, p_Q, s_2, 3, SOL)$ e $THP(\lambda_M, p_Q, s_3, 2, SOL)$.

Agora para $THP(\lambda_M, p_Q, s_1, 2, SOL)$ e $THP(\lambda_M, p_Q, s_2, 2, SOL)$, e porque a única distribuição possível de 2 é $f_2 = \{1, 1\}$, é feita uma expansão de s_1 (respectivamente s_2), de cardinalidade 2. Em ambos os casos a partição é sobre o valor 2,

Algoritmo THP(e_Y, p, s, n, Sol)
{input: e_N uma estratégia, p um problema, s um estado, n um inteiro, Sol uma lista de soluções
IF $< n = 1 >$
THEN THS(e, {s}, Sol)
ELSE
let $f_n = \{x_1, \ldots, x_k\}$
let $X = expansion(s, e_{Y=k}, p)$
IF $0 < \#X < k$
THEN $F = distribuicao(f_n, \#X)$
ELSE $F = f_n$
FORALL s_i in X DO
let $x = first(F)$
let $F = F \backslash x$
IF $<$ s$_i$ is solution $>$
THEN
$add(s_i, Sol)$
let $F = resdistribut(F, x)$
ELSE THP(e_Y, p, s_i, x, Sol)

Figura 7.6: Algoritmo dum agente paralelo

Algoritmo redistribut(f_n,a)
{input: f_n uma distribuição, a um inteiro}
let $f_n = \{x_1, \ldots, x_l\}$
let $i = 1$
WHILE $i \leq a$ DO
$x_i = x_i + 1$
$i++$
return $f_{n+a} = \{x_1, \ldots, x_l\}$

Figura 7.7: Algoritmo duma redistribuição dum valor numa distribuição

Algoritmo distribuicao(f_n,c)
{input: f_n uma distribuição, c um inteiro}
let $F = f_n = \{x_1, x_2, \ldots\}$
WHILE $\#F > c$ DO
    F=redistribut($\{x2, \ldots\}, x_1$)
return F

Figura 7.8: Algoritmo duma redistribuição para obter uma cardinalidade fixa

obtendo-se $\{s_4, s_5\}$ (respectivamente $\{s_6, s_7\}$). São lançados $THP(\lambda_M, p_Q, s_4, 1, SOL)$ e $THP(\lambda_M, p_Q, s_5, 1, SOL)$ (respectivamente $THP(\lambda_M, p_Q, s_6, 1, SOL)$ e $THP(\lambda_M, p_Q, s_7, 1, SOL)$), terminando $THP(\lambda_M, p_Q, s_1, 2, SOL)$ (respectivamente $THP(\lambda_M, p_Q, s_2, 2, SOL)$).

Para $THP(\lambda_M, p_Q, s_3, 1, SOL)$, porque a quota de agentes é 1 é lançado o agente sequencial $THS(\lambda, \{s_3\}, SOL)$.

Todos os agentes no terceiro nível da árvore têm quota 1, e portanto são lançados os 4 agentes sequenciais $THS(\lambda, p_q, \{s_4\}, SOL)$, $THS(\lambda, p_q, \{s_5\}, SOL)$, $THS(\lambda, p_q, \{s_6\}, SOL)$ e $THS(\lambda, p_q, \{s_7\}, SOL)$. Estes 4 agentes mais o outro lançado no nível anterior ($THS(\lambda, p_q, \{s_3\}, SOL)$), percorrerão a árvore de estados, cada uma deles sequencialmente, até esgotarem as suas listas, passando as soluções do problema para uma lista(comum) de soluções. Quando todos os agentes tiverem terminado a árvore foi percorrida na sua totalidade pelos 5 agentes sequenciais. Num ambiente com múltiplos processadores estes agentes podem processar em paralelo varrendo na totalidade o espaço de soluções.

Suponhamos agora que pretendemos resolver o problema $X \geq Y$,para $X \in \{1, \ldots, 10\}$ e $Y \in \{3, \ldots, 15\}$, usando 3 agentes.

Este sistema de restrições tem as 36 soluções apresentadas na figura 7.9.

O estado inicial do problema é definido pelo estado s_0 na figura 7.11. Usando uma distribuição de $f_3 = \{1, 1, 1\}$, a expansão de s_0 segundo $\lambda_M = 3$, é constituída por 2 estados que derivam de s_0 por partição do valor 2 (o de maior cardinalidade 13 contra 10), em 3 sub valores, $[3, 6]$, $[7, 10]$, $[11, 15]$. A propagação

(3;3)	(4;3)	(5;3)	(6;3)	(7;3)	(8;3)	(9;3)	(10;3)
	(4;4)	(5;4)	(6;4)	(7;4)	(8;4)	(9;4)	(10;4)
		(5;5)	(6;5)	(7;5)	(8;5)	(9;5)	(10;5)
			(6;6)	(7;6)	(8;6)	(9;6)	(10;6)
				(7;7)	(8;7)	(9;7)	(10;7)
					(8;8)	(9;8)	(10;8)
						(9;9)	(10;9)
							(10;10)

Figura 7.9: Soluções do problema $X \geq Y$,para $X \in \{1, \ldots, 10\}$ e $Y \in \{3, \ldots, 15\}$. A primeira coordenada dos pares corresponde a X, a segunda a Y.

altera os valores da variável 1 para $[3, 10]$ em s_1, e $[7, 10]$ em s_2. O estado s_3 torna-se inconsistente não pertencendo portanto aos estados da expansão. A quota de s_3 é redistribuída, sendo lançado sobre s_1 um agente paralelo com quota 2, e sobre s_2 um agente paralelo com quota 1. s_2 induz a execução dum agente sequencial A_1, da qual resultarão as soluções da figura 7.9, correspondentes ao rectângulo do canto inferior direito.

O agente paralelo correspondente a s_1, expande este estado nos estados s_4, e s_5, que induzem a execução dos agentes sequenciais, A_2 e A_3, que determinam respectivamente as soluções correspondentes ao rectângulo superior esquerdo e rectângulo superior direito da figura 7.9.

7.4 Quantificação do Trabalho de um Agente

Pelo facto de serem lançados N agentes que irão processar independentemente uma parte da árvore, não existe qualquer garantia de que a pesquisa seja realizada pelos N agentes em paralelo. Como contra-exemplo tome-se a pesquisa da figura 7.10. Dentre os agentes sequenciais o agente s_3, será o primeiro a ser lançado, sendo os agentes relativos a s_4, s_5, s_6 e s_7 posteriores.

De modo a quantificar o trabalho de cada agente, considere-se o trabalho de um estado s, como o produto das cardinalidades dos seus valores (v_i), i.e.,

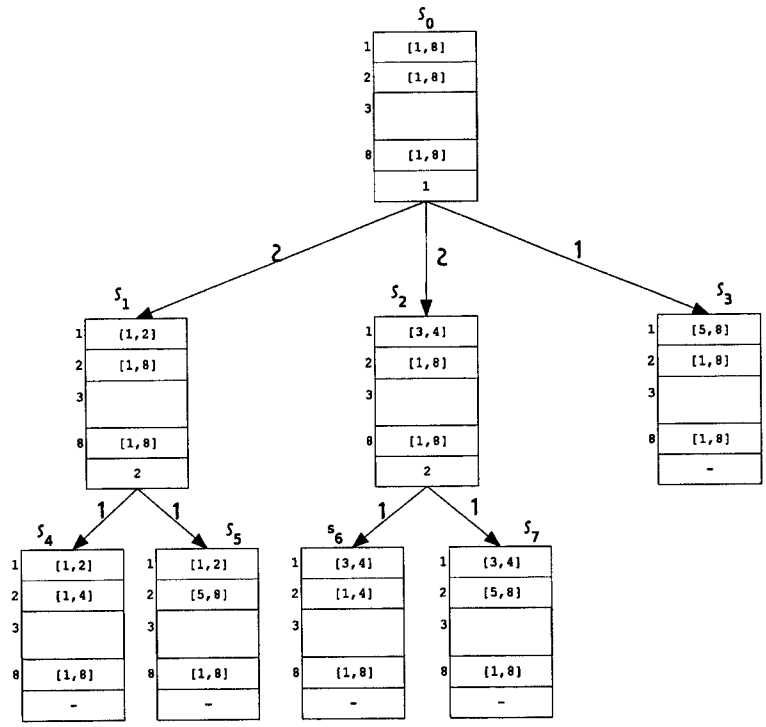


Figura 7.10: Trace duma execução paralela para o problema das Queens 8, usando uma distribuição de 5, $f_5 = \{2, 2, 1\}$

$W(s) = \prod \#v_i$. O trabalho dum agente sequencial pode ser quantificado pelo trabalho da sua lista de estados. Sendo s_{ini} o estado que com que se inicia o agente sequencial, A_{seq}, o trabalho do agente pode ser majorado por este estado.

$$W(A_{seq}) = W(s_{ini}) \geq W(L) = \sum_{s \in L} W(s)$$

$$\begin{aligned}
W(s_3) &= \#[5,8] \times \textstyle\prod_{i=2}^{8} \#[1,8] &&= 4 \times 8^7 \\
W(s_4) &= 2 \times \#[1,4] \times \textstyle\prod_{i=3}^{8} \#[1,8] &&= 2 \times 4 \times 8^6 = 8^7 \\
W(s_5) &= 2 \times \#[5,8] \times \textstyle\prod_{i=3}^{8} \#[1,8] &&= 2 \times 4 \times 8^6 = 8^7 \\
W(s_6) &= 2 \times \#[1,4] \times \textstyle\prod_{i=3}^{8} \#[1,8] &&= 2 \times 4 \times 8^6 = 8^7 \\
W(s_7) &= 2 \times \#[5,8] \times \textstyle\prod_{i=3}^{8} \#[1,8] &&= 2 \times 4 \times 8^6 = 8^7
\end{aligned}$$

O agente correspondente a s_3, tem 4 vezes mais trabalho que os outros agentes, sendo portanto natural que os outros terminem primeiro, perdendo-se assim

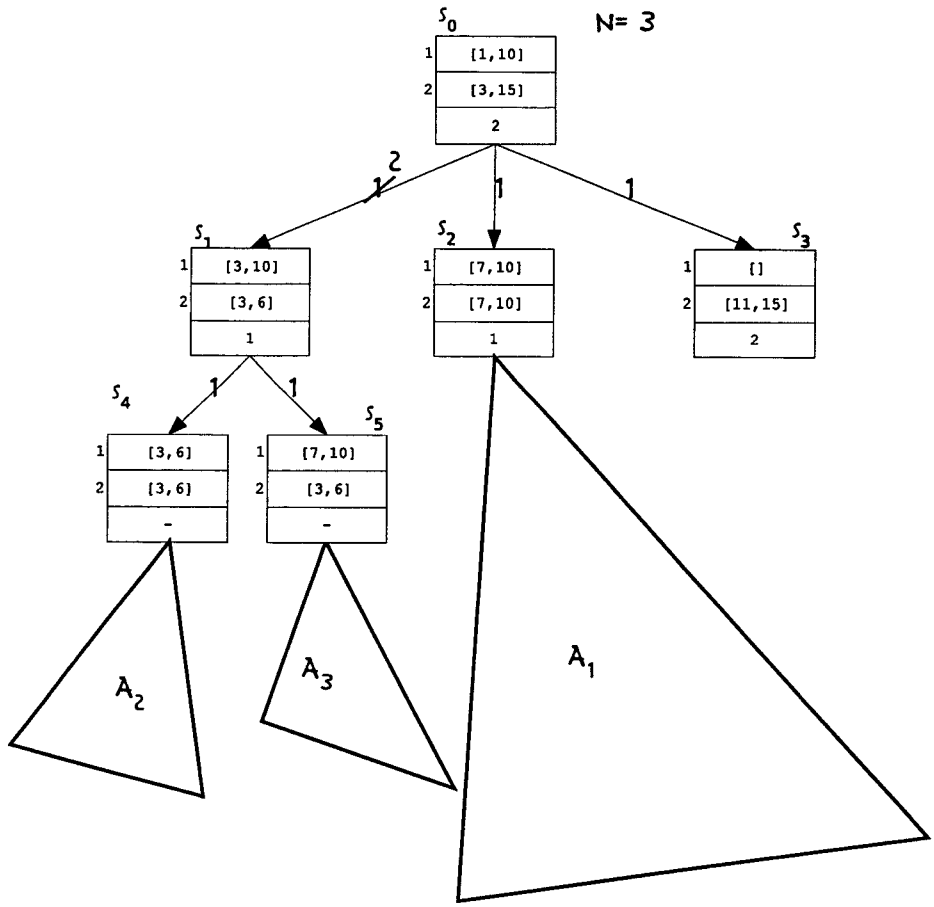


Figura 7.11: Trace duma execução paralela do problema $X \geq Y$,para $X \in \{1,\ldots,10\}$ e $Y \in \{3,\ldots,15\}$, usando 3 agentes e uma distribuição de 3, $f_3 = \{1,1,1\}$

alguma da capacidade de processamento que existia. Deixar que um agente termine imediatamente após esgotar a sua lista de estados, sem previamente notificar os outros agentes de que acabou não facultando a reutilização da sua capacidade de processamento, terá seguramente consequências negativas no desempenho do sistema. Dependendo da configuração da árvore de estados, poderá ter-se o caso limite de executar uma pesquisa que supostamente será realizada por um determinado número de agentes, e na prática a pesquisa ser toda sequencial e executada por um só.

7.4.1 Pesquisa Sequencial com Comunicação

Para não permitir que ocorram situações como as descritas na secção 7.4, i.e., que sejam permitidas situações em que um trabalho que era para ser efectuado por N agentes, seja, pelo menos durante algum tempo, efectuado por um número inferior de agentes, vamos especificar um comportamento diferente para os agentes sequenciais:

Um agente sequencial iniciar-se-á do mesmo modo que o descrito em 7.3.1, embora só possa terminar depois de dar conhecimento aos outros agentes de que tal irá acontecer.

Quando esgotar a sua lista, o agente informará que vai terminar. Dentre todos os agentes, aquele que estiver mais sobrecarregado, lançará um agente sequencial com metade do seu trabalho, mantendo-se assim a quota inicial de agentes sequenciais que operam em paralelo. Atendendo a que já quantificamos o trabalho dos agentes, resta especificar a forma de comunicação entre os agentes. Os agentes estão numa lista circular duplamente ligada, tendo cada agente conhecimento de dois outros: o seguinte, *next* e o anterior, *previous*. Para comunicarem entre si, os agentes enviam mensagens ao agente seguinte, que reenviará a mensagem. Sendo a lista de agentes circular, as mensagens percorrerão todos os agentes, chegando ao seu destino.

Embora para passar mensagens não fosse necessário uma lista duplamente ligada, para estabelecer as ligações entre os agentes sequenciais/paralelos que vão sendo criados, tal é necessário.

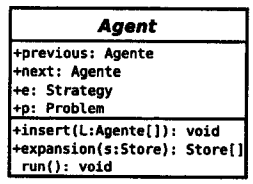
Quando um agente se prepara para terminar envia ao próximo agente uma mensagem de fim (*end*). Quando um qualquer agente A recebe uma *end*, existem duas hipóteses:

- foi ele próprio que iniciou o ciclo de mensagens, o que significa que a mensagem já deu a "volta", logo, todos os agentes já receberam e enviaram uma *end*, sendo portanto conhecido o agente mais sobrecarregado de trabalho, B. A informará então B de que se pode dividir e termina.

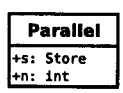
- foi outro agente, que não o próprio, a desencadear o ciclo de mensagens. O agente compara a sua quantidade de trabalho com a da mensagem e

actualiza-as se for caso disso, reenviando a mensagem.

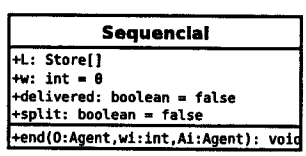
7.5 Modelo Orientado a Objectos dos Agentes



Um agente é uma entidade abstracta que possui uma estratégia, (correspondente ao modo como fará as partições), um problema (o problema de restrições a resolver) um agente antecessor (o agente "antes" na lista circular), e um agente seguinte (ordem na lista). O método *insert* permitirá estabelecer as relações de *next* e *previsous* para os novos agentes da lista, L. A ordenação $previous; this; next$ será modificada para $previous; L_1; \ldots; L_n; next$. O método *expansion* fará a expansão do estado dado como argumento, usando a estratégia do agente, e as restrições do problema.O método abstracto *run* especificará o modo de actuação dos agentes, sendo definido nas subclasses:

- 

  Parallel implementa a classe dos agentes paralelos, sendo definida, para além das características especificadas na superclasse, pelo estado, e pela quota. O seu método *run* é similar ao apresentado na figura 7.4, embora as execuções `THX` devam ser substituídas pela criação dos objectos correspondentes aos agentes paralelos ou sequenciais conforme o caso, e pelo uso do método `insert`, para actualizar as ligações entre os agentes. Finalmente aos objectos criados deverá ser aplicado o método `run`;

- 

  Sequencial implementa a classe dos agentes sequenciais. Está especializada com uma lista de estados, L, uma medição do trabalho do agente w, e duas variáveis booleanas, *delivered* e *split*. *delivered* é usada para salvaguardar a entrega da mensagem de fim e *split* quando activada corresponde a uma autorização para o agente se dividir. O método `end`, usado para determinar qual o agente mais sobrecarregado de trabalho tem a implementação na figura 7.12.

```
Algoritmo end(O, m, A)
IF < O == this >
THEN
    IF < A ≠ this >
    THEN
        let delivered = true
        A.split=true
ELSE
    IF < w > m >
    THEN next.end(O, w, this)
    ELSE next.end(O, m, A)
```

```
Algoritmo run()
let Sol = {}
WHILE < L ≠ ∅ > DO
    IF < split >
    THEN
        let L1 = {}; w1 = 0
        WHILE w1 < w DO
            let s = L.get()
            L1.add(s)
            w1+ = work(s)
            w− = work(s)
        let As= new Sequencial(
        e,L1,previous,this,w1,false)
        let previous = As
        let previous.next = As
        As.run();
    ELSE
        let s' = L.get()
        let X = expansion(s')
        FOREACH x in X DO
            IF < x is solution >
            THEN Sol.add(x)
            ELSE L.add(x); w+ =work(x)
next.end(this,0,this)
WHILE ¬delivered DO
    wait();
Sol.print();
```

Figura 7.12: Algoritmos correspondentes aos métodos da classe Sequencial

A execução dum problema de restrições usando os agentes, passa por, como é habitual no AJACS, criar o objecto problema definindo o estado inicial e adici-

onando as restrições pretendidas. A criação dum agente paralelo em que é dado o problema, uma estratégia e uma quota, é suficiente para criar um objecto *Parallel*, cujo estado inicial é o estado do problema. A aplicação do método *run* ao objecto criado desencadeia o processo de pesquisa, sendo as soluções lançadas directamente pelos agentes sequenciais quando terminam.

Não foi aqui usado nenhum esquema de recolha global de soluções. Tal passaria por criar um objecto do tipo *Controller* de modo análogo ao definido no capítulo 6. Este controlador conteria a lista global de soluções, e criaria o agente paralelo responsável pela pesquisa. Os agentes conteriam uma referência (através duma variável de instância definida no agente) ao controlador. Antes de terminar a pesquisa, os agentes passariam as soluções encontradas ao controlador. O controlador após lançar o agente paralelo que iniciaria a pesquisa ficaria em espera, até que todos os agentes terminassem. Atendendo a que o controlador não teria conhecimento da existência dos agentes, já que são os agentes que lançam os outros agentes, esta sincronização teria de ser feita contabilizando os agentes que terminam.

Atendendo a que a pesquisa pode ser realizada por mais agentes do que aquele predefinido pela quota (os agentes podem-se subdividir!), a contabilização dos agentes que terminam teria se ser ajustada para somente no caso do agente não dar ordem de split a outro agente notificar o controlador do seu fim.

Por último note-se que um agente dá ordem de split a outro agente desde que a quantidade de trabalho desse agente seja superior ou igual a 1, o poderá ser desnecessário. Pode-se facilmente parametrizar esta ordem adicionando a condição $m > k$ à condição $A \neq this$ no algoritmo *end*.

7.6 Conclusões

Consideremos como modelo de agentes de pesquisa uma representação das entidades que intervêm directamente no processo de pesquisa, sendo responsáveis por percorrer o espaço de soluções, i.e. a árvore de estados.

No capítulo 6 foi apresentado um modelo de agentes, com 2 variantes. O modelo do capítulo 6 é baseado nos conceitos de Controlador e Trabalhador. Os

agentes que intervêm na pesquisa são um Controlador e vários trabalhadores. O controlador contem a informação relativa ao problema (restrições, estado inicial, e estratégia) e uma quota de agentes trabalhadores que serão os responsáveis por percorrer o espaço de estados. O controlador criará tantos trabalhadores quantos a sua quota permitir, ficando à espera que lhe seja indicado que a pesquisa terminou. O fim da pesquisa é declarado quando todos os trabalhadores estão ociosos. Cada trabalhador tem uma referência para o seu controlador, e comunica com ele, passando-lhe soluções, estados e outras informações. O controlador é assim o objecto de ligação entre vários trabalhadores assegurando a comunicação entre estes. Por este motivo, mesmo não intervindo directamente no processo de pesquisa, tem de estar activo (mas em espera) durante todo o processo, sendo o último agente a terminar. Os trabalhadores vão expandindo estados e recolhendo soluções. A gestão dos estados dos trabalhadores motiva as duas variantes deste modelo: gestão centralizada (capítulo 6 secção 6.2.1) e gestão local (capítulo 6 secção 6.2.2).

Na gestão centralizada o controlador tem uma lista de estados à qual os trabalhadores acedem para ir buscar trabalho (os estados a expandir), e que actualizam com todos os estados que resultam da expansão dos estados que processam, exceptuando um dos estados, que será o próximo estado que expandem.

Na gestão local cada trabalhador tem a sua própria lista de estados, que tenta processar independentemente dos restantes trabalhadores. Só quando não tem mais estados a processar, ser-lhe-á passado do trabalhador com mais trabalho, um novo estado para processar.

Analisemos grosseiramente as comunicações no modelo "Controlador/Trabalhador" independentemente da variante considerada. Em cada iteração do ciclo, o trabalhador pergunta ao controlador "já acabou?": Se a resposta é não, o trabalhador vai cumprir mais uma iteração do seu ciclo vendo se tem trabalho. No caso afirmativo, expande e colecciona estados e volta ao início, perguntando novamente "Já acabou?". Se o trabalhador não tem trabalho, tem de saber se dos restantes trabalhadores há alguém a trabalhar ou se pelo contrário estão todos à espera. Se estão todos à espera o trabalhador comunica para o controlador que a pesquisa acabou, caso contrário o trabalhador ficará à espera, cumprindo uma

iteração do seu ciclo, e iniciando outra perguntando ao controlador "Já acabou?".

O motivo de tanta comunicação tipo:"já acabou?" advém da necessidade de sincronizar o fim da pesquisa e evitar situações de "deadlock/starvation", motivadas pelo facto de todos os trabalhadores estarem à espera, de algo que não irá acontecer, i.e. algum trabalhador produzir estados para dar trabalho a quem está à espera. Em ambas as variantes do modelo estas comunicações existem e correspondem a acessos sincronizados visto *finish* ser uma variável partilhada e passível de ser modificada por qualquer dos intervenientes na pesquisa e cuja consistência tem de ser assegurada. A execução do modelo controlador/trabalhadores sobre um ambiente distribuído sofre das penalizações inerentes aos acessos sincronizados diminuindo a sua performance. No caso do Hyperion temos como penalizações as invalidações da cache dos nós e as comunicações relativas às transmissões para os home-nodes das alterações efectuadas às variáveis da cache. Esta é a motivação que nos leva a desenvolver outros modelos em que a comunicação seja o mais reduzida possível, o que foi desenvolvido neste capítulo.

Neste capítulo foi desenvolvido um modelo de agentes de pesquisa com três variantes:

- No primeiro caso não foi considerado qualquer limite para o número de agentes, sendo lançado um agente paralelo por cada estado gerado na expansão. Estes agentes processam um estado, colecionam soluções e lançam outros agentes paralelos para processar os estados resultantes, terminando. Os agentes não comunicam entre si, sendo portanto uma boa abordagem neste sentido, pese embora não ser possível controlar o número de agentes que efectuam a pesquisa;

- No segundo caso existe um limite para o número de agentes sequenciais, que operam em simultâneo na árvore de estados. Um agente sequencial termina quando não tiver mais estados a processar. Deste facto resulta que na prática não existem garantias de que a pesquisa seja executada exactamente pelo número pré fixado de agentes, podendo-o ser em número inferior. Trata-se também esta, duma variante que não exige qualquer comunicação entre os agentes;

- No último caso, tenta manater-se até ao final a quota de agentes sequenciais pré-fixada. Para tal um agente, antes de terminar, envia um messagem a outro agente. Estas contêm informação sobre: qual o agente que iniciou o ciclo de mensagens; qual o agente, dentre aqueles pelos quais a mensagem já passou, com mais trabalho; e uma quantificação desse trabalho. O agente que inicia o ciclo de mensagens espera até ter retorno da mensagem enviada. Antes de terminar envia ao agente mais ocupado informação de que se pode dividir. Este, usando a sua lista de estados, constroi uma lista com sensivelmente metade do seu trabalho, e cria um agente sequencial para a explorar. É a única variante que exige comunicação entre os agentes, e de certo modo comparável aos modelos "Controlador/trabalhador", no entanto agora não existe transferência de estados entre os agentes, existe um autorização para um agente se dividir, criando outro agente, em virtude de um outro estar a acabar. As trocas de informação entre os agentes são realizadas passando entre eles uma mensagem, à qual cada agente responde com informação sobre si próprio.

Conseguimos assim um modelo para a Pesquisa no AJACS realizada por um número fixo de agentes e onde, pelo menos conceptualmente, a comunicação entre os agentes é inferior à do modelo apressentado no capítulo 6.

Capítulo 8

Uma Aplicação: Construção de Horários

Neste capítulo é apresentada a modelação duma aplicação que utiliza o AJACS como revolvedor de restrições associado. A criação de horários escolares foi o problema tratado, com particular ênfase para a criação de horários de professores, salas e alunos em função duma distribuição de serviço docente.

8.1 Introdução

No decorrer deste capítulo iremos ilustrar o uso do AJACS na construção duma aplicação real, a *Criação de Horários Escolares.* A utilização do AJACS na realização de horários escolares, permitirá mostrar a aplicabilidade do sistema a um problema real e concreto. Embora possam ser extrapoladas do exemplo outras aplicações idênticas (i.e. outro tipo de horários, ou num âmbito mais geral aplicações de escalonamento), o padrão definido para a construção dos horários é o usado pela Universidade de Évora.

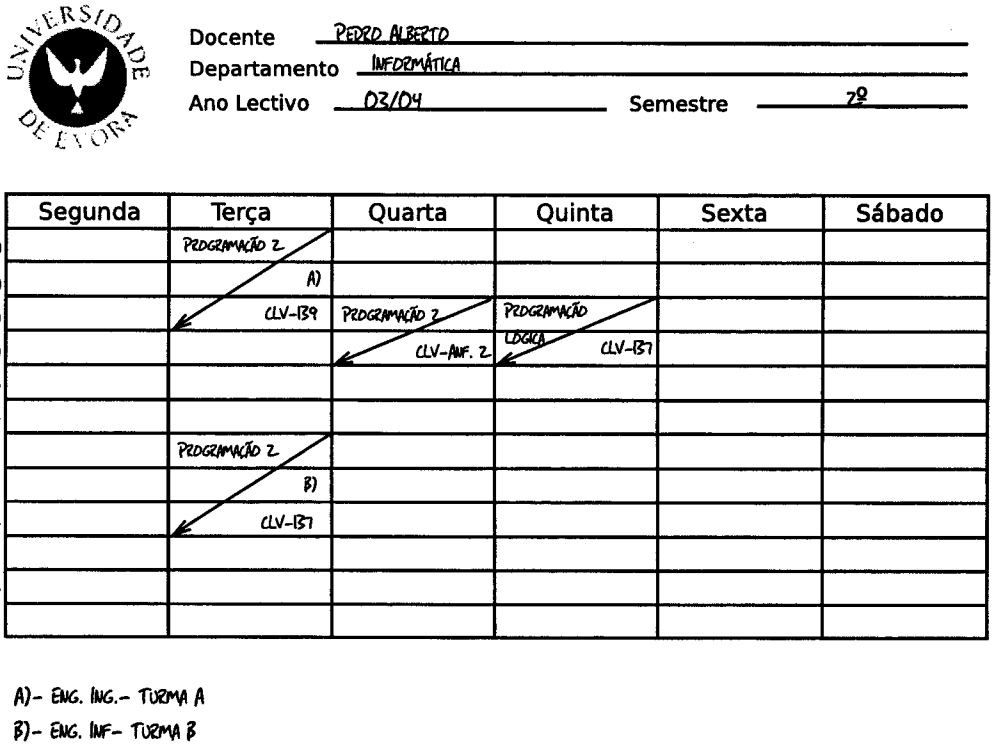
As motivações existentes para a escolha efectuada são óbvias: O conceito não é novidade para nenhum docente nem aluno: com periodicidade variável mas tipicamente anual ou semestralmente, docentes e alunos têm novos horários(ver figura 8.1 para exemplos de horários de professores); o acesso ao Sistema de Informação da Universidade de Évora para aceder a características que são fundamentais para montar uma aplicação credível está claramente facilitado, e finalmente o 'know-how' necessário para criar uma aplicação deste género está na posse de pessoas com quem diariamente trabalhamos.

O resto do capítulo está organizado da seguinte forma: na secção 8.2 é introduzido e formalizado o problema a tratar, na secção 8.3 é mostrado como é possível usar o AJACS para construir soluções para o problema e finalmente na secção 8.4 é feita uma análise de resultados.

8.2 Especificação do Problema

A construção de horários é essencialmente um problema de escalonamento. O problema é o de distribuir um conjunto de tarefas, as *aulas*, num espaço temporal (usualmente a semana). As tarefas têm uma duração, e envolvem recursos: *salas*, *professores* e *alunos.*

Estes recursos não são partilháveis, dado que professores e alunos não poderem estar à mesma hora em aulas diferentes (caso referenciado como sobreposição de horário), nem uma sala poder conter mais do que uma aula ao mesmo tempo. Excepções a esta regra poderão ser feitas: Como no exemplo apresentado na fi-

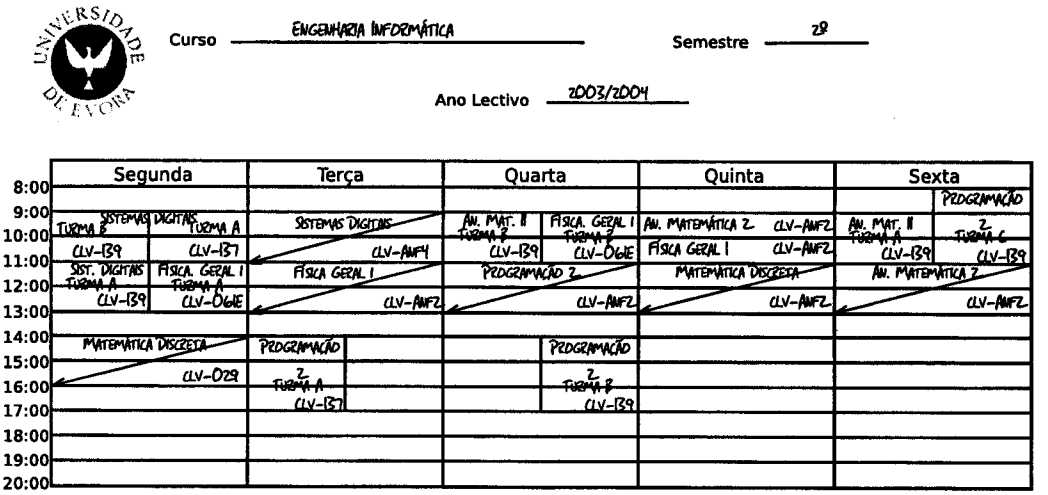


Docente PEDRO ALBERTO
Departamento INFORMÁTICA
Ano Lectivo 03/04 Semestre 2º

	Segunda	Terça	Quarta	Quinta	Sexta	Sábado
8:00		PROGRAMAÇÃO 2				
9:00		A)				
10:00		CLV-139	PROGRAMAÇÃO 2	PROGRAMAÇÃO		
11:00			CLV-ANF. 2	LÓGICA CLV-137		
12:00						
13:00						
14:00		PROGRAMAÇÃO 2				
15:00		B)				
16:00		CLV-137				
17:00						
18:00						
19:00						

A)- ENG. ING.- TURMA A
B)- ENG. INF- TURMA B

Figura 8.1: Hipotético horário de um docente da UE.



Curso ENGENHARIA INFORMÁTICA Semestre 2º
Ano Lectivo 2003/2004

	Segunda	Terça	Quarta	Quinta	Sexta
8:00					
9:00					PROGRAMAÇÃO
10:00	SISTEMAS DIGITAIS TURMA B CLV-139 / SISTEMAS DIGITAIS TURMA A CLV-137	SISTEMAS DIGITAIS CLV-ANF4	AN. MAT. II TURMA B CLV-139 / FÍSICA GERAL I TURMA B CLV-D6IE	AN. MATEMÁTICA 2 CLV-ANF2 / FÍSICA GERAL I CLV-ANF2	AN. MAT. II TURMA A CLV-139 / 2 TURMA C CLV-139
11:00	SIST. DIGITAIS TURMA A CLV-139 / FÍSICA GERAL I TURMA A CLV-D6IE	FÍSICA GERAL I CLV-ANF2	PROGRAMAÇÃO 2 CLV-ANF2	MATEMÁTICA DISCRETA CLV-ANF2	AN. MATEMÁTICA 2 CLV-ANF2
12:00					
13:00					
14:00	MATEMÁTICA DISCRETA CLV-029	PROGRAMAÇÃO 2 TURMA A CLV-137	PROGRAMAÇÃO 2 TURMA B CLV-139		
15:00					
16:00					
17:00					
18:00					
19:00					
20:00					

Figura 8.2: Hipotético horário dos alunos.

gura 8.2, existem de facto no horário dos alunos, em algumas condições, aulas sobrepostas. Tratando-se de turnos práticos, poderá existir sobreposição de turnos, optando o aluno pela turno (horário) que mais lhe convier. Esta sobreposição é possível no caso de existir mais do que um turno prático. No caso dos horários dos professores e das salas, esta excepção não é válida.

Assim, de um modo geral, temos uma exclusividade dos recursos à tarefa pelo tempo que a mesma durar. Garantir que os horários gerados não violam estas restrições de exclusividade é o requisito mínimo. Outras questões que se prendem com a gestão mais ou menos eficiente dos recursos (p.ex não atribuir um anfiteatro com capacidade para 200 alunos a uma turma de 30 alunos) poderão também se ser contempladas sendo necessário pré-definir quais as situações a considerar.

Sendo o AJACS uma biblioteca de restrições de domínios finitos, há necessidade de mapear as associações possíveis entre dias da semana e horas a um conjunto de inteiros. De um modo formal, considere-se $\mathcal{A}$, o conjunto de todas as aulas, $\mathcal{S}$, o conjunto das salas, $\mathcal{P}$, o conjunto dos professores e $\mathcal{T}$ o conjunto das turmas. Seja $f(\mathcal{D} \times H)$, para $\mathcal{D} = \{\text{dias da semana}\}$ e $H = \{\text{horas do dia}\}$, um subconjunto dos inteiros que represente as horas semanais (Segunda às 9:00, Terça às 15:00, etc) . A correspondência entre uma 'hora' n, e uma hora semanal terá será obtida através duma função invertível f,

$$\begin{aligned} f: \ \mathcal{D} \times H &\longrightarrow \mathcal{N} \\ (d,h) &\longmapsto n \end{aligned}$$

$$\begin{aligned} f^{-1}: \ \mathcal{N} &\longrightarrow \mathcal{D} \times H \\ n &\longmapsto (d,h) \end{aligned}$$

que verifique

$$(d,h) + 1 = (d_1,h_1) \iff f(d,h) + 1 = f(d_1,h_1) \tag{8.1}$$

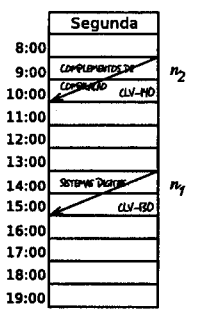
O mapeamento expresso na figura 8.3 exemplifica uma possibilidade na definição de f. Neste caso $f(\text{segunda}, 12{:}00) = 5$ e $f^{-1}(56) = (\text{sexta}, 15{:}00)$. Da equação 8.1, a hora seguinte a terça-feira às 10:00 horas é terça às 11:00 horas (uma vez que 16 é o inteiro seguinte a 15). Também pela mesma equação a hora seguinte a Segunda-feira às 19:00 é Terça-feira às 8:00.

	Segunda	Terça	Quarta	Quinta	Sexta	Sábado
8:00	1	13	25	37	49	61
9:00	2	14	26	38	50	62
10:00	3	15	27	39	51	63
11:00	4	16	28	40	52	64
12:00	5	17	29	41	53	65
13:00	6	18	30	42	54	66
14:00	7	19	31	43	55	67
15:00	8	20	32	44	56	68
16:00	9	21	33	45	57	69
17:00	10	22	34	46	58	70
18:00	11	23	35	47	59	71
19:00	12	24	36	48	60	72

Figura 8.3: Possível mapeamento das horas da semana num domínio inteiro.

Seja $(a, (s, p, t), n)$, para $a \in \mathcal{A}$, $s \in \mathcal{S}$, $p \in \mathcal{P}$, $t \in \mathcal{T}$ e $n \in \mathbb{N}$ a afectação dos recursos s, p, e t, à aula a, que inicia às n horas, com a seguinte semântica: na sala s, $f^{-1}(n)$, o professor p dá a aula a, à turma t. Considere-se ainda $dur(a)$ a duração da aula a. Um subconjunto $\mathcal{X}$ de $\mathcal{A} \times (\mathcal{S} \times \mathcal{P} \times \mathcal{T}) \times \mathbb{N}$ é um horário se:

1. $\forall x, y \in X : x|_p = y|_p \Rightarrow x|_n \geq y|_n + dur(y|_a) \vee y|_n \geq x|_n + dur(x|_a)$;
2. $\forall x, y \in X : x|_s = y|_s \Rightarrow x|_n \geq y|_n + dur(y|_a) \vee y|_n \geq x|_n + dur(x|_a)$
3. $\forall x, y \in X : x|_t = y|_t \Rightarrow x|_n \geq y|_n + dur(y|_a) \vee y|_n \geq x|_n + dur(x|_a)$



As condições anteriores realizam as restrições de exclusividade (relativamente à tarefa aula) de professores, salas e turmas, garantindo a não sobreposição das mesmas, nos horários correspondentes;

Em 1 é assegurada a não sobreposição duma aula para um professor: para que duas quaisquer aulas dadas pelo mesmo professor não se sobreponham, uma delas terá se realizar-se antes da outra podendo iniciar-se a segunda somente depois da primeira terminar (ver figura).

Em 2 são enunciadas, de modo análogo, as condições de exclusividade das salas para as aulas, devendo pois manter-se a restrição de exclusividade para os tuplos que afectem a mesma sala.

Em 3 a restrição é a mesma, agora validada para a turma. Esta restrição, pode em casos especiais, ser relaxada. Usualmente na U.E., quando a dimensão da turma assim o exige, existe uma turma teórica, que corresponde à totalidade dos alunos, e mais do que uma turma prática, designados por turnos. Neste caso, não é obrigatório que todos os turnos práticos de todas as disciplinas não se sobreponham, bastando garantir que esta exclusividade seja validada para os diferentes turnos práticos da mesma disciplina, e a também a exclusividade entre a teórica(única) e os turnos práticos.

Sejam $((\text{teórica}, P2), (129, lsf, A), 27)$ e $((\text{prática}, P2), (137, lsf, A_1), 28)$ dois tuplos duma distribuição, e considere-se:

$$\text{duração}((\text{teórica,P2}))=2 \text{ e } \text{duração}((\text{prática,SD}))=3,$$

dado que os dois tuplos correspondem a serviços do mesmo professor,*lsf*, e que $27 \ngeq 28+3$ nem $28 \ngeq 27+2$ estes tuplos violam as restrições especificadas 1 e correspondem a uma sobreposição de horário para o professor. Também por que a disciplina é a mesma *P2* a restrição 3 é violada significando que os alunos da turma prática A_1 não podem assistir à aula teórica (pelo menos na totalidade).

Uma distribuição $\mathcal{X}$, que valide as restrições 1, 2, e 3, é geradora de horários admissíveis para professores alunos e salas de aula:

- Horários dos professores: O horário de um qualquer professor, A, é obtido de todos os tuplos de $\mathcal{X}$, cujo professor seja A, seja $\mathcal{H}(A) = \{x \equiv (a,(s,A,t),h) \in X : x|_p = A\}$ O horário conterá as entradas (a,t,s) dispostas nas horas correspondentes. Na figura 8.5 está exemplificada a construção dum horário de docente, usando a especificação considerada.

- Horários das salas: Os horários das salas podem ser obtidos, de modo análogo aos horários dos professores. Considerem-se todos os tuplos de $\mathcal{X}$, cuja sala seja S, seja $\mathcal{H}(s) = \{x \equiv (a,(s,A,t),h) \in X : x|_s = S\}$. O horário da sala S, conterá as entradas (a) dispostas nas horas correspondentes.

- Horários dos alunos: Os horários dos alunos são construídos, usando todos os tuplos de $\mathcal{X}$, correspondentes às aulas de determinado ano e curso....

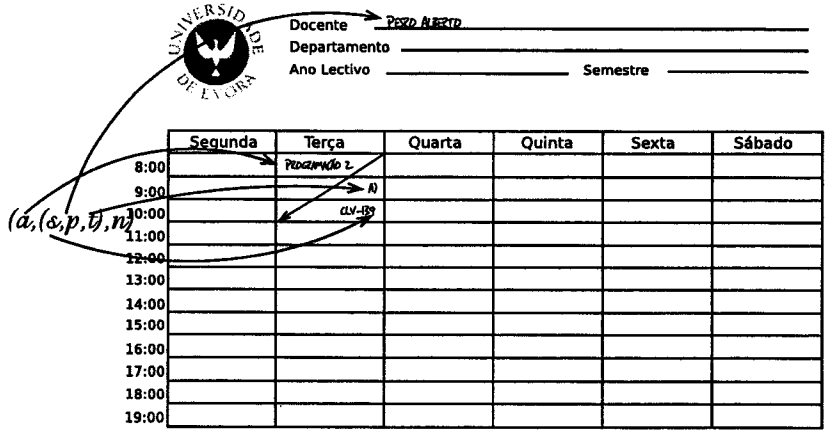


Figura 8.4: Construção do horário de um professor.

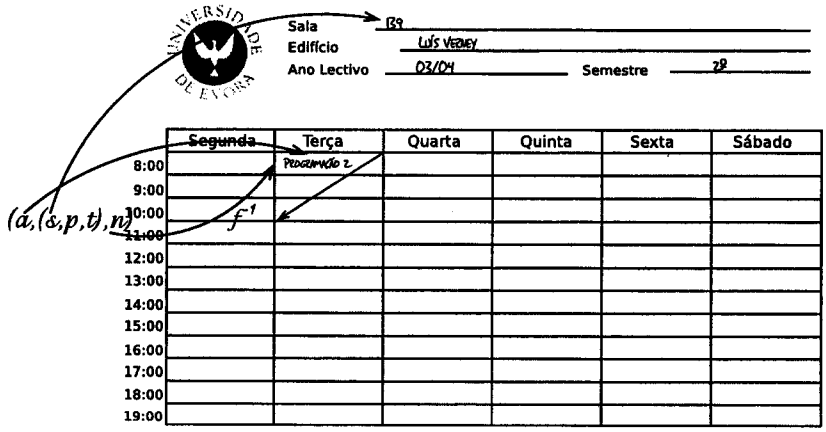


Figura 8.5: Construção do horário duma sala.

8.3 Modelação usando o AJACS

A resolução deste problema utilizando o AJACS, passa inevitavelmente por formalizá-lo em termos de restrições de variáveis sobre domínios expressas como objectos. A realização dos horários tem por base uma distribuição de serviço docente, na qual estão mencionados os docentes e as aulas que lhe estão atribuídas. A informação contida numa distribuição é regra geral bastante simplificada, dado existir implicitamente o conhecimento necessário para dela retirar a semântica apropriada.

Na figura 8.3, a tabela apresentada representa uma distribuição de serviço. Cada linha da tabela, à excepção do cabeçalho, diz respeito ao serviço afecto ao

Prof.	Disc.	Teo.	Pra.	Disc.	Teo.	Pra.	Disc.	Teo.	Pra.
AED	ES	1	0	IHM	1	0			
CC	ISI	1	0	STI	0	1			
CA	CG	1	2	IFN2	0	1			
IPR	ADA	1	0	BD	1	0			
LSF	P2	1	2						
LMR	RC	1	0	LP	1	0			
MB	TI	1	1	AS	0	1			
RT	A	1	1	RC	0	1			
SPA	ILD	1	0	COMP	1	0			
TG	BD	0	2	IMD2	0	1	IFN2	0	1
TR	AS	1	0	SD	1	2			
VP	LP	0	1	SD	0	1			
PP	COMP	0	1	ADA	0	2	TR2	0	1
JJG	ES	0	1	IHM	0	1			
NUD	P2	0	2	ILD	0	1			

Figura 8.6: Ficheiro com uma distribuição de serviço

professor correspondente. No caso apresentado, o professor AED (na 1^a linha), lecciona uma teórica de ES e uma teórica de IHM, correspondendo ES e IHM a siglas de disciplinas e AED a uma sigla identificadora do professor.

De modo a construir os horários derivados duma distribuição, é necessário saber informações relativas às disciplinas, por exemplo 'Quanto dura uma teórica de ILD? ' ou, 'Quantos alunos tem uma prática de P2?' etc. Também informações sobre as salas são necessárias, 'Quantos alunos cabem na sala 130?' ou 'A sala 140 é um laboratório, um anfiteatro ou uma sala normal? '.

8.3.1 Disciplinas

A informação relativa às disciplinas, está armazenada numa tabela de disciplinas, onde constam, entre outras informações(ver figura 8.8), a caracterização das aulas da disciplina e as turmas da disciplina.

A caracterização das aulas define, o número de unidades de cada tipo e a duração das mesmas. Por exemplo, uma disciplina que tenha por semana 1 teórica de duas horas e duas praticas de três horas, será caracterizada por uma *aula(0,1,2)* e *aula(1,2,3)*. A duração das aulas dado ser o AJACS um resolvedor de restrições de domínios finitos, é um número inteiro. Sem perda de generalidade considerou-se que as duração das aulas é um múltiplo da hora. Esta particularização não compromete outras situações, no caso de existirem aulas com durações de por exemplo, $1,5h$, teremos de considerar unidades de $1/2$ hora, e considerar tomar para 3 a duração da aula. Neste caso também o mapeamento das horas da semana num domínio inteiro seria diferente do apresentado na figura 8.3: teríamos o dobro das 'unidades', correspondendo Segunda-feira às 8:00 à unidade 1, segunda-feira às 8:30 à unidade 2, etc.

A definição das turmas está previamente definida e é aqui usada somente para gerar os horários do semestre de cada curso. A turma atribuída a um professor num serviço, é uma qualquer turma disponível(i.e. ainda não atribuída) de tipo coincidente com o serviço (teórico, prática, etc).

8.3.2 Serviços

A leitura dum ficheiro correspondente a uma distribuição de serviço, permite gerar uma lista de serviços, que se caracterizam pelo professor, pela disciplina e pela turma. Em cada serviço a sigla do professor permite aceder a uma tabela com a informação relativa aos professores, a sigla da disciplina à tabela de disciplinas e a turma à turma correspondente ao serviço, obtida através da tabela de disciplinas.

8.3.3 Salas

A caracterização das salas é fundamental, visto que estas condicionam a elaboração dos horários. Cada sala é caracterizada por um numero inteiro (para posterior utilização na construção de valores), pela capacidade, pelo tipo (teóricas, práticas ou outro tipo) e por designações correspondentes ao modo como estas são identificadas. O tipo das salas permitirá escolher quais das salas existentes podem ser usadas para determinado serviço, em função do tipo deste.

8.3.4 Estado Inicial e Soluções

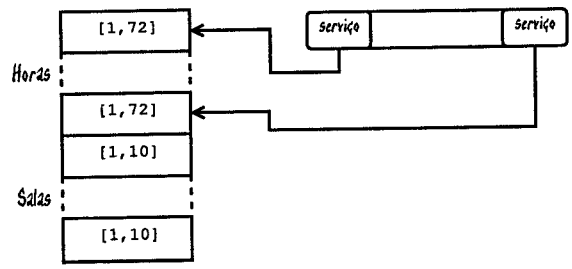


Figura 8.7: Semântica do estado inicial do problema.

A lista de serviços permitirá gerar o estado inicial do problema. Seja n a dimensão desta lista. As n variáveis do estado correspondentes aos índices $i \in \{1, \ldots, n\}$, correspondem à hora a que se inicia o serviço i, sendo inicializadas com um domínio inteiro correspondente às horas possíveis para a realização do serviço. Este domínio inicial poderá ser o mais abrangente possível correspondendo às 72 horas semanais, ou mais restrito, correspondendo a algumas restrições iniciais que se queiram fazer relativamente ao horário de um docente. Por exemplo poderá existir necessidade de especificar que determinado professor só poderá ter aulas às Quartas-feiras, caso em que o domínio inicial de todos os seus serviços será o domínio [25,35] (ver figura 8.7).

Uma solução que atribua a uma destas i variáveis o valor k, significa que o serviço correspondente se inicia àquela hora (e acaba à hora k+duração).

De modo a definir, além da hora a que se realiza cada serviço, também a sala em que este se efectua, as n variáveis do "estado" correspondentes aos índices $j \in \{n+1, \ldots, 2n\}$, são usadas para este efeito. A sala em que ocorre o serviço i corresponde ao valor da variável $i+n$. O valor inicial destas variáveis é obtido usando um valor que contenha os valores da salas correspondentes ao mesmo tipo. Por exemplo, se o serviço i corresponde a uma aula teórica, o valor inicial da variável $i+n$, é definido por todos os valores das salas cuja tipo é teórico.

Uma solução que atribua à variável i o valor k, e à variável $i+n$ o valor l, significa que o serviço i, se realiza à hora k, na sala l.

8.3.5 Restrições

Com a formalização usada, e de modo o obter soluções para o problema interessa garantir que as variáveis correspondentes a serviços relacionados não se sobreponham. Seja *NoOverlap(i,j,di,dj)* a restrição que relaciona os valores das variáveis i e j, usando as constantes di e dj. A definição do *update(S,i)*, i.e. o modo como será afectado o estado S por a variável no índice i ter mudado é obtida do seguinte modo: Considerem-se Vi e Vj os valores respectivamente das variáveis i e j de S, e di, dj as correspondentes durações, se $Vi = k$, Vj pode ser actualizado suprimindo do seu domínio os valores $\{k - dj - 1, \ldots, k + di - 1\}$. A remoção destes valores do domínio de Vj assegurará que futuras instanciações da variável j, corresponderão a valores que não geram sobreposições, uma vez que $\forall y \in V_j, y \geq k+di \vee y \leq k-dj$. Dado $Vi = \{k\}$ tem-se $Vj \geq Vi+di \vee Vj+dj \leq Vi$ que são as condições que devem ser asseguradas para as restrições de exclusividade (ver secção 8.2, restrições 1, 2 e 3).

Considere-se como exemplo um professor que tenha 3 serviços(aula 1, aula 2 e aula 3) cujas durações sejam respectivamente de 3, 2 e 2 horas. Considere-se ainda que este professor só pode aceitar um horário com horas às Quartas-feiras, começando às 9:00 e terminando às 17:00. Seja S o estado inicial do problema, definido pelos valores V[0],V[1] e V[2]. Atendendo às especificações dadas o estado inicial é definido por:

V[0]=[26,33]
V[1]=[26,33]
V[2]=[26,33]

Correspondendo V[i] à hora a que se inicia o serviço i, nem todos os valores deste domínio são admissíveis. Se o professor pretende terminar as aulas às 17:00 , qualquer dos seus serviços não poderá iniciar-se a esta hora(a última hora possível para iniciar um serviço será na pior das hipóteses a hora 33). Também a hora de almoço(13:00) não é admissível para afectar o início de um serviço. Retirados dos domínios tais valores inadmissíveis temos como valores iniciais de V[0],V[1] e V[2] os domínios:

V[0]={26, 27, 31}

V[1]={26, 27, 28, 31, 32}

V[2]={26, 27, 28, 31, 32}

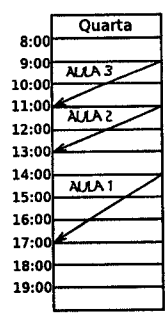
Resta agora garantir que as soluções apresentadas não contenham sobreposições. Para tal basta associar aos pares de serviços (0,1), (0,2) e (1,2), às restrições *NoOverlap*:

NoOverlap(0,1,3,2)

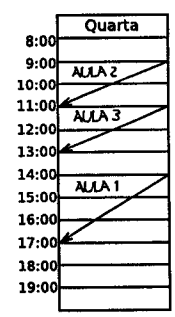
NoOverlap(0,2,3,2)

NoOverlap(1,2,2,2)

Adicionando estas restrições ao problema definido sobre o estado inicial, são obtidas 2 soluções: $\{(31;28;26);(31;26;28)\}$ que correspondem aos 2 horários possíveis para o professor:



Horário para (31;28;26)



Horário para (31;26;28)

A não sobreposição de salas é obtida usando restrições que envolvem 4 variáveis *NoOverlapRoom(Vi,Vj,Ri,Rj,di,dj)*, onde Vi e Vj são tarefas de duração di e dj respectivamente sendo Ri o recurso afecto à tarefa Vi e Rj o recurso de Vj. A restrição actualiza os valores de Vi e Vj do seguinte modo: Se Ri e Rj são valores singulares e iguais (isto é o recurso de ambas as tarefas é o mesmo) então Vi e Vj têm de ser tarefas exclusivas sendo estes domínios actualizados como em *NoOverlap*. A actualização dos domínios Ri e Rj em função das modificações de Vi e Vj é feita predeterminando se Vi e Vj são actividades que se sobrepõe. Em caso afirmativo se algum dos $R's$ for um valor singular, este valor será retirado do domínio do outro recurso já que não é possível partilha-los.

8.3.6 Soluções e Horários

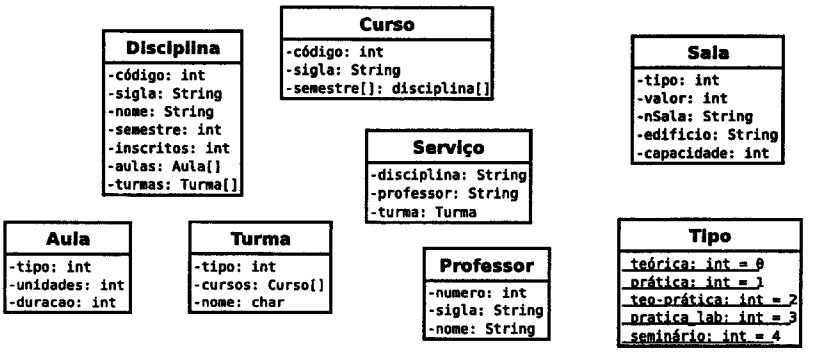


Figura 8.8: Organização das classes intervenientes na elaboração dos horários

O esquema de classes correspondente à formalização do problema é o apresentado na figura 8.8. Dada uma lista de serviços, L, de dimensão n e uma solução *sol* para o problema (*sol* é um estado tal como o especificado em 8.3.4), os horários são obtidos da seguinte forma:

- O horário dum professor X é obtido determinando em L quais os índices de serviços cujo professor seja X. Para cada um desses índices j, no horário de X, na entradas de $f^{-1}(sol[j])$ até $f^{-1}(sol[j]) + d - 1$, figurarão: serviço[j].disciplina, serviço[j].turma, e $sol[j + n]$, respectivamente a disciplina, a turma e a sala. O tipo de serviço(teórica/prática) é determinado pelo tipo associado à turma, bem como a duração da aula.

- O horário duma sala a, é obtido tomando todos os $j \in \{n, \ldots 2n-1\} : s[j] = a$. Nas entradas correspondentes a $f^{-1}(sol[j-n])$ até $f^{-1}(sol[j-n]) + d - 1$ figurarão serviço[j-n].disciplina e serviço[j-n].turma, respectivamente, a disciplina e a turma. A duração d é obtida do mesmo modo que em 8.3.6.

- Os horários dos alunos são obtidos em função do curso. Dado um curso c cada semestre m compreende uma lista de disciplinas($\{d_1, \ldots, d_k\}$) e por sua vez cada disciplina d compreende uma lista de turmas $\{t_p, \ldots, t_r\}$. Para cada uma destas turmas t procure-se em L o serviço j que lhe corresponde (serviço[j].turma = t). No horário dos alunos do curso c para o semestre m, nas entradas de $f^{-1}(sol[j])$ até $f^{-1}(sol[j]) + d - 1$, figurarão:

serviço[j].disciplina e serviço[j].turma e $sol[j+n]$. A realização deste processo para todas as turmas de todas as disciplinas do semestre m do curso c, gera o horário pretendido.

8.4 Conclusões

A modelação apresentada permite elaborar horários a partir duma distribuição de serviço, desde claro está que exista a(uma) hierarquia de classes de suporte ao sistema, análoga à especificada na figura 8.8. A distribuição de serviço permite gerar uma lista de serviços, o estado inicial e consequentemente o problema. As restrições são adicionadas ao problema fazendo uso das restrições definidas, para manter a exclusividade de determinados recursos (professores, alunos/turmas e salas). Exemplos de restrições de exclusividade foram definidos na secção 8.3.5.

A resolução do problema e as soluções, são obtidas do modo usual no AJACS: definindo um Controlador que contem o problema e especificando o número de trabalhadores que o executam. A elaboração de horários finais de professores, salas e turmas a partir das soluções do problema está descrita em 8.3.6.

Capítulo 9

Conclusões e Trabalho Futuro

Trata-se do último capítulo desta tese. Impõe-se uma retrospectiva do trabalho realizado e uma análise conclusiva do mesmo. Passado, presente e futuro são analisados. O que foi feito, o que ficou por fazer, o que foi alcançado, o que se esperava, o que se obteve é resumido. Os projectos para o futuro são delineados.

9.1 O trabalho desta tese

Esta tese aborda o tema da implementação de domínios finitos numa biblioteca escrita em Java, para o Java. A implementação de bibliotecas de restrições para o Java é nos dias que correm, uma ideia com já algumas implementações, inclusivamente de âmbito comercial, mas que à data em que os trabalhos desta tese se iniciaram era um conceito inovador.

A primeira implementação duma biblioteca de restrições para o Java abordada nesta tese, foi o *GC*([FA00b], [FA99]). Esta implementação possui algumas particularidades que nos fizeram evoluir para uma outra, tentando preservar do *GC* as suas características mais interessantes e introduzindo novas ideias que se concretizaram numa nova concepção ("design") e numa implementação, o *AJACS*([FA01], [FA00a]) com características adicionais que a diferenciassem claramente do *GC*, tornando-a susceptível de obter um melhor desempenho, nomeadamente na execução sobre sistemas paralelos e distribuídos.

Assim, o sistema de propagação do *GC* foi mantido no *AJACS*, mas foi alterado o conceito de variável, tendo sido introduzidos os conceitos de valor e estado. Os valores/domínios, à semelhança das variáveis no *GC*, têm três implementações distintas mas são apresentados ao utilizador como possuindo uma única representação, que poderá internamente ir mudando em função da sua adequação .

A intuição é criar estados independentes entre si ou que, a haver referências entre estados, estas sejam só de leitura: um modelo quasi-funcional em que a operação básica é a re-escrita de estados, nunca havendo lugar a modificações a um estado existente. Esta independência entre estados, obtida intencionalmente na concepção do modelo, permitirá a exploraração concorrente de sub-espaços de estados independentes, sem que haja interferência entre estes processos.

Os estados aparecem para agrupar um conjunto de valores, e é sobre estes que todas as operações relativas ao sistema de restrições são efectuadas. Um problema é construído dando o estado inicial, i.e. os domínios associados às variáveis do problema e um conjunto de restrições sobre estas variáveis. A determinação das soluções dum problema faz-se evoluindo para outros estados em função da estratégia da pesquisa.

As estratégias, são outro dos novos conceitos introduzidos no *AJACS*: São apresentadas duma forma clara ao utilizador, que para definir novas estratégias precisa simplesmente de definir uma subclasse duma classe abstracta e implementar os dois métodos correspondentes à sua interface: o método correspondente à escolha do domínio a particionar e o método correspondente à obtenção das partições.

A aplicação duma estratégia a um estado permite gerar um novo estado em que o valor da variável seleccionada é, no novo estado, um dos domínios da partição, mantendo-se inalterados os domínios das restantes variáveis. A propagação desta afectação gerará o estado para o qual se evolui. Este processo permite construir uma árvore de estados em que a raiz é o estado inicial do problema e as soluções são folhas. Nesta árvore, os estados no mesmo nível não têm dependências com os irmãos, só com o seu antecessor. A árvore de estados é configurada no nível inferior pela escolha da variável, no mesmo nível pela definição das partições.

A determinação das soluções do sistema de restrições, faz-se por pesquisa:

1. O algoritmo de pesquisa sequencial do AJACS repete o processo: dado um estado, escolhe uma variável (em função da estratégia) e escolhe uma partição (também em função da estratégia), e propaga. Se o estado gerado é inconsistente, escolhe outra partição e se tal não for possível volta ao estado antecessor e escolhe outra variável. Note-se que mantendo o mesmo algoritmo de pesquisa podemos obter uma ordem diferente na determinação das soluções se variarmos a estratégia.

2. O algoritmo de pesquisa paralela do AJACS repete o processo: Dado um estado escolhe uma variável(em função da estratégia) e determina todos os estados derivados desse estado tomando para a variável escolhida em cada um deles as diferentes partições e propaga. Este processo é designado por expansão do estado. Coleccionam-se os estados da expansão e elege-se um novo estado, para repetir o processo.

A pesquisa paralela poderá ser executada por mais do que um agente. Neste caso, cada um dos agentes que intervêm na pesquisa executará o algoritmo des-

crito em 2. Está garantido, como foi demonstrado no capítulo 4, que os múltiplos agentes que percorrem o espaço de soluções (a árvore) não repetem estados nem ignoram estados e que a conjunção dos esforços dos agentes realiza a pesquisa pretendida. Esta garantia é dada pela estrutura em árvore do espaço de soluções em que não há estados repetidos e que a partir dum estado é possível gerar a sub-árvore de estados com raiz nesse estado.

A implementação da pesquisa paralela com múltiplos agentes é facilitada no Java dada a existência de threads. Executar a pesquisa paralela com verdadeiro paralelismo só é possível numa arquitectura multiprocessador, quer de memória partilhada, quer de memória distribuída.

Uma arquitectura de memória partilhada e escalável, é um recurso que não possuímos mas uma arquitectura de memória distribuída seria possível implementar dado que possuímos um cluster de 4×bi-processadores (inicialmente 8)

.

Testar uma implementação distribuída do AJACS foi possível recorrendo ao Hyperion que, fazendo uso da PM2 e da camada DSM_PM2 subjacente, implementa uma arquitectura de memória distribuída no cluster. O Hyperion fornece-nos também a implementação duma única JVM sobre o cluster o que nos permite uma total transparência na definição e implementação dos AJACS distribuído e das suas aplicações.

Os resultados experimentais atestam da correcção desta abordagem. Os testes do AJACS distribuído sobre o Hyperion e o conhecimento desta aplicação, mais do que a definição dos sistemas, sua modelação teórica e as respectivas implementações, consumiram muito do tempo dedicado a esta tese. Sobretudo porque as observações de desempenho efectuadas não correspondem às expectativas que tínhamos do modelo. Observámos overheads da ordem dos 80 a 90% e uma fraca paralelização, resultados que são discordantes com os apresentados nas publicações do Hyperion, que apresentam ganhos de desempenho quase lineares, isto é, para configurações com n nós são obtidos speedups próximos de n. Estes overheads são sensivelmente idênticos mesmo para as aplicações de teste distribuídas com o Hyperion e para uma aplicação em que não existem objectos partilhados.

Esta evidência sugere que existe necessidade de investigar, modificar ou criar novas implementações para as camadas subjacentes ao Hyperion, em particular a DSM. Sem este trabalho não será possível obter com a AJACS o desempenho satisfatório.

9.1.1 Apreciação

Das propostas apresentadas no capítulo 1, na secção 1.4, o JACK e o JSolver, são abordagens comparáveis ao trabalho apresentado nesta tese.

JACK diferencia-se das nossas abordagens no esquema apresentado para a definição das restrições, o uso do JCHR, uma implementação da CHR em Java, poderá limitar a sua aplicabilidade dado ser necessária uma familiarização com os conceitos ("HEADS", "GUARDS", "GOALS" e "SUSPENSION") subjacentes à edição das regras para a sua utilização. No entanto tal como o AJACS, providencia formas de (re)definir a pesquisa, o que o torna numa aplicação versátil.

O JSolver, tal como o AJACS ou o GC,são bibliotecas escritas totalmente em Java. O JSolver uma aplicação comercial, tal como o Koalog. A comparação entre as abordagens foi realizada com base na primeira publicação do JSolver, retirada dos proceedings da *PACLP99*. O ano de 99 foi também o da apresentação do GC, no workshop de " Parallelism and Implementation Technology for (Constraint) Logic Programming Languages" satélite da realização do *ICLP99* [FA00a]. O JSolver é comparável ao GC, na abordagem efectuada à pesquisa, mas mais completo que o GC por permitir a definição de heurísticas na escolha das variáveis e valores a iterar. Esta limitação do GC foi ultrapassada no AJACS. O AJACS foi apresentado em 2000, no AGP, tendo sido selecionado para publicação em [FA01].

O DJ apresenta uma abordagem diferente ao problema da resolução de sistemas de restrições em Java. É uma linguagem construída para auxiliar a construção de aplicações gráficas que usa restrições para a especificação das relações entre as componentes da aplicação. Além da especificidade que a caracteriza, a resolução das restrições é assegurada pelo B-Prolog.

A diferença fundamental entre o AJACS e todas estas livrarias de restrições

em Java é a possibilidade que o AJACS fornece de providenciar execução paralela em ambientes distribuídos, de forma transparente. Esta característica advém da utilização dos threads do Java. Outras possibilidades do Java, como por exemplo a da criação de interfaces gráficas, foram em alguns casos exploradas noutras abordagens, a utilização de threads e as execuções paralelas, a nosso conhecimento não foram.

9.2 Trabalho Futuro

Nesta secção apresentaremos algumas das direcções que o trabalho futuro poderá tomar:

- A apresentação de resultados positivos (de ganhos de desempenho) duma implementação distribuída do *AJACS*, requer outros tipo de trabalhos: dado que o Hyperion providencia uma forma transparente de executar um programa com threads num cluster de computadores, e dado que não existem outras formas evidentes de o fazer, há que investigar a camada DSM subjacente ao Hyperion e trabalhá-la. Trata-se dum trabalho que sai um pouco do âmbito deste tese, que é o implementação de sistemas de restrições em Java. No entanto, não conseguir mostrar os resultados esperados é uma motivação forte para investigar outras áreas, mesmo que não estejam na mesma linha de estudos. Também a execução do modelo distribuído do AJACS noutras arquitecturas, em particular em arquitecturas multiprocessdor de memória partilhada é outra das linhas a seguir.

- A experiência obtida na execução paralela de sistemas de restrições em Java, poderá ser uma mais valia a utilizar na paralelização de outros sistemas de restrições em Java, por exemplo a paralelização da JACK é outra das hipóteses de trabalho futuro a considerar.

- Outra dos aspectos deixado em aberto nesta tese, foi a redirecção dos trabalhos para a área das interfaces gráficas e das Linguagens de Programação Visuais. Dado o Java providenciar os meios para o fazer seria sem dúvida

interessante dotar o *AJACS* de sub-sistemas que permitissem monitorizar os estados e a propagação das restrições.

- Produzir traços de execução susceptíveis de serem utilizados pelos visualizadores de propagação de restrições conformes ao formato de traço genérico, especificado pelo projecto OADymPPaC [Der04].

- A conjugação do uso de Linguagens Visuais e do AJACS poderá também ser explorado, em particular definir uma Linguagem Visual para construir problemas, estratégias, algoritmos de pesquisa e porque não definir restrições deste modo? Dado que as restrições são definidas quer no AJACS quer no *GC* definindo o método *update* para a restrição, não seria de todo irrealista defini-las de modo visual através das operações sobre domínios que estão associadas aos valores. A introdução do operador de cardinalidade no AJACS, facilitaria em muito esta possibilidade de definir restrições de modo visual.

  A integração do AJACS numa Linguagem de Programação Visual poderia ainda ser orientada no sentido do uso das restrições para especificar as relações entre os compenentes da linguagem, em termos de disposição ("layout").

Bibliografia

[A+01] Gabriel Antoniu et al. The Hyperion system: Compiling multithreaded Java bytecode for distributed execution. *Parallel Computing*, 27(10):1279–1297, 2001.

[AB01] Gabriel Antoniu and Luc Bougé. DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. *Lecture Notes in Computer Science*, 2026:55–??, 2001. citeseer.nj.nec.com/article/antoniu01dsmpm.html.

[ABH+00] Gabriel Antoniu, Luc Bougé, Philip Hatcher, Mark MacBeth, Keith McGuigan, and Raymond Namyst. Implementing Java consistency using a generic, multithreaded DSM runtime system. *Lecture Notes in Computer Science*, 1800:560–??, 2000. citeseer.nj.nec.com/article/antoniu00implementing.html.

[AH] G. Antoniu and P. Hatcher. Remote object detection in Cluster-Based java. pages 108–108. citeseer.nj.nec.com/antoniu01remote.html.

[BCH94] S. Janson B. Carlson and S. Haridi. Akl(fd): A concurrent language for fd programming. In *In Proceedings of the 1994 International Logic Programming Symposium, MIT Press Series in Logic Programming*, 1994.

[Bes94] C. Bessiére. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994.

[BvB98] F. Bacchus and P. van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *Proceedings of the 15th*

National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98), pages 311–318, Menlo Park, July 26–30 1998. AAAI Press.

[CB99] J.-Ch. Régin Ch. Bessiere, E. Freuder. Using constraint metaknowledge to reduce arc-consistency computation. *Artificial Intelligence*, 107:125–148, 1999.

[CD96a] P. Codognet and D. Diaz. Compilling constraints in `clp(fd)`. *Journal of Logic Programming*, 27(3), 1996.

[CD96b] Philippe Codognet and Daniel Diaz. Compiling Constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, June 1996.

[CD01] P. Codognet and D. Diaz. Another local search method for constraint solving. Berlin, Germany, 2001. Stochastic Algorithms, Foundations and Applications (SAGA).

[Chu99] Andy Hon Wai Chun. Constraint Programming in Java with JSolver. In *In Proceedings of the First International Conference and Exhibition on The Practical Application of Constraint Technologies and Logic Programming (PACLP-99), London, April, 1999.*, 1999.

[Cod70] E.F. Codd. A relational model of data for large shared databanks. *Communications of the ACM*, 13(6):377–387, 1970.

[CR02] Don Cameron and Greg Regnier. *The Virtual Interface Architecture.* Intel Press, 2002.

[CS94] P.R. Cooper and M.J. Swain. *Constraint-Based Reasoning*, chapter Arc Consistency:parallelism and domain depenedence. MIT/Elsevier, 1994.

[Dec90] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1990.

[Der04] Pierre Deransart. Main results of the OADymPPaC project. In Bart Demoen and Vladimir Lifschitz, editors, *Proceedings of the Twentieth International Conference on Logic Programming*, volume 3132 of *Lecture Notes in Computer Science*, St. Malo, France, September 2004. Springer-Verlag.

[DGN98] Denys Duchier, Claire Gardent, and Joachim Niehren. *Concurrent Constraint Programming in Oz for Natural Language Processing.* Programming Systems Lab, Universität des Saarlandes, Germany, 1998. Available at http://www.ps.uni-sb.de/Papers.

[Dia95] Daniel Diaz. *Étude de la compilation des Langages Logiques de Programmation par Contraintes sur les Domaines Finis: le Système CLP(FD).* PhD thesis, Université d'Orléans, 1995.

[DSHB03] Frej Drejhammar, Christian Schulte, Seif Haridi, and Per Brand. Flow Java: Declarative concurrency for Java. In Catuscia Palamidessi, editor, *Proceedings of the Nineteenth International Conference on Logic Programming*, volume 2916 of *Lecture Notes in Computer Science*, pages 346–360, Mumbai, India, December 2003. Springer-Verlag.

[F.86] Glover F. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, pages 533–549, 1986.

[FA99] Lígia Ferreira and Salvador Abreu. A Constaint Logic Programming Framework in Java. In *Proceedings of the Workshop on Parallelism and Implementation Technologies, 1999.* New Mexico State University, 1999.

[FA00a] Lgia Ferreira and Salvador Abreu. Design for AJACS, yet another Java Constaint Programming framework. In *Proceedings of AGP00: The 2000 Joint Conference on Declarative Programming*, December 2000.

[FA00b] Lígia Ferreira and Salvador Pinto Abreu. A Constaint Logic Programming Framework in Java. *Elsevier Electronic Notes in Theoretical Computer Science*, 30(issue 3), 2000. http://www.elsevier.nl/gejng/31/29/23/55/27/show/Products/notes/index.http.

[FA01] Ligia Ferreira and Salvador Pinto Abreu. Design for AJACS, yet another Java constraint programming framework. In Agostino Dovier, Maria Chiara Meo, and Andrea Omicini, editors, *Declarative Programming – Selected Papers from AGP'00*, volume 48 of *Electronic Notes in Theoretical Computer Science*, pages 167–178. Elsevier Science B. V., 2001.

[Frü98] Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, Vol 37(1-3), 1998.

[GL95] F. Glover and M. Laguna. Tabu search. In C.R. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*, pages 70–150. McGraw-Hill, 1995.

[Hen97] Martin Henz. *Objects for Concurrent Constraint Programming.* Internationale Series in Engineering and Computer Science. Kluwer Academic Publishers, Boston, MA, USA, 1997.

[Hen99] Pascal Van Hentenryck. *The OPL, Optimization Programming Language.* The MIT Press, 1999.

[HJ80] M. Haralick and J.Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[HL88] C.C. Han and C-H. Lee. Comments on Mohr and Henderson's path consistency algorithm. *Artificial Intelligence*, 36:125–130, 1988.

[HP96] John L. Hennsey and David A. Patterson. *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., second edition edition, 1996.

[JG96] G. Steele Jr. J. Gosling, W. Joy. *The Java Language Specification.* Addison-Wesley, Reading, Massachusetts, 1996.

[JJY] P. Stuckey J Jaffar, S. Michaylov and R. Yap. An abstract machine for CLP(R). In *Proceedings of the* ACM SIGPLAN *Conference on Programming Language DEsign and Implementation*, pages 128–139.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings Fourtheenth Annual ACM Symposium on Principles of Programming Languages*, pages 111–119. "ACM", 1987.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19(20):503–581, 1994.

[KvB97] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89:365–387, 1997.

[Mac77] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[MDB88] H. Simonis A. Aggoun T. Graf M. Dincbas, P. Van Hentenryck and F. Berthier. The constraint logic programming CHIP. In *Proceedings of the Internacional Conference on Fifth Generation Computer Systems*, Tokyo, Japan,1988.

[MH86] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233, 1986.

[MHP99] D. Cabeza M. Garcia de la Banda P. Lopez M. Hermenegildo, F. Bueno and G. Puebla. The CIAO multi-dialect compiler and system. In *In Parallelism and Implementation of Logic and Constraint Logic Programming.*, 1999.

[MJPL92] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3):161–205, 1992.

[MMH98] M. Macbeth, K. McGuigan, and P. Hatcher. Executing java threads in parallel in a distributed-memory environment. In *In Proceedings of CASCON'98*, pages 40–54. IBM Canada and the National Research Council of Canada, 1998.

[MT95] K. McAloon and C. Tretkoff. *2LP: Linear Programming and Logic Programming.* Principles and Practice of Constraint Progamming. The MIT Press, 1995.

[NFZY98] Sousuke Kaneko Neng-Fa Zhou and Kouji Yamauchi. Dj: A java-based constraint language and system. In *In Proceedings ofthe Annual JSSST conference*, 1998.

[PL95] Jean-Francois Puget and Michel Leconte. Beyond the Glass Box: Constraints as Objects. In *Logic Programming, Proceedings of the 1995 International Symposium*, pages 513–527. MIT Press, 1995. ISBN 0-262-62099-5.

[PVHD95] V. A. Saraswat P. Van Hentenryck and Y. Deville. *Constraint Programming: Basics and Trends*, chapter Constraint Processing in cc(fd). LNCS 910. Springer Verlag, 1995.

[RDP90] F. Rossi, V. Dahr, and C. Petrie. On the equivalence of constraint satisfaction problems. In *Proceedings. European Conference on Artificial Intelligence, ECAI90*, Stockholm, August 1990. Also: MCC Technical Report ACT-AI-222-89.

[Sar89] Vijay A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, January 1989. Available as Technical Report CMU-CS-89-108.

[SAS02] Matthias Saft Slim Abdennadher, Ekkerhard Krämer and Matthias Schmauss. Jack: A java constraint kit. *Elsevier Electronic Notes in Theoretical Computer Science*, 64, 2002.

[SM92] Levesque H. Selman, B. and D. Mitchell. A new method for solving hard satisfiability problems. In *In Proceedings of the Tenth National Conferenceon Artificial Intelligence*, pages 440–446, 1992.

[TKB01] Luc Bougé Thilo Kielmann, Philip Hatcher and Henri E. Bal. Enabling java for high-performance computing: Exploiting distributed shared memory and remote method invocation. *Communications of the ACM*, 44(10 - Special issue on Java for High Performance Computing):110–117, 2001.

[TWF97] M. Torrens, R. Weigel, and B. Faltings. Java constraint library: Bringing constraints technology on the internet using the java language. *In Constraints and Agents: Papers from the 1997 AAAI Workshop, 21–25. Menlo Park, California, USA: AAAI Press.*, 1997.

[vHDT92] P. van Hentenryck, Y. Deville, and C-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[Vod88] P. Voda. Types of trilogy. In *Proceedings of fifth International Conference on Logic Programming*, 1988.