INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

DEPARTAMENTO DE ENGENHARIA ELECTRÓNICA E TELECOMUNICAÇÕES E DE COMPUTADORES (DEETC)

Engenharia Informática e de Computadores

PROCURA DISTRIBUÍDA DE SOLUÇÕES COM RESTRIÇÕES LOCAIS E GLOBAIS

Bruno Miguel da Silva Pereira

DISSERTAÇÃO PARA OBTENÇÃO DO GRAU DE MESTRE
EM ENGENHARIA INFORMÁTICA E DE COMPUTADORES

Orientador:
Doutor Paulo Manuel Trigo Cândido da Silva

Júri:
Coordenador do Mestrado
Doutor Luís Alberto dos Santos Antunes
Doutor Paulo Manuel Trigo Cândido da Silva

2008

"Anyone who thinks the sky is the limit, has limited imagination."

Unknown Author

Resumo

Esta dissertação aborda a procura distribuída de soluções baseando-se em cenários cuja informação não esteja totalmente disponível, por não ser possível ou mesmo porque não é desejável.

Os dois tipos de problemas abordados neste trabalho são: i) o problema de atribuição de tarefas, e ii) o problema de agendamento de eventos. O problema de atribuição de tarefas é ilustrado num cenário de catástrofe (atribuição de incêndios a bombeiros) cujos agentes tentam encontrar a melhor solução global de acordo com as capacidades de cada um. O problema do agendamento de eventos com múltiplos recursos, ocorre numa organização que pretende maximizar o valor do tempo dos seus empregados, enquanto preserva o valor individual (privacidade) atribuído ao evento (valor da importância relativa do evento). Estes problemas são explorados para confrontar os dois tipos de abordagem na sua resolução: centralizada e distribuída.

Os problemas são formulados para resolução de maneira distribuída e centralizada, de modo a evidenciar as suas características e as situações em que fará mais sentido a utilização de cada abordagem. O desempenho a nível de tempo de execução e consumo de memória, bem como o conceito de privacidade são os pontos considerados no comparativo das abordagens centralizada e distribuída.

Para analisar o problema de atribuição de tarefas e o problema de agendamento de eventos, é proposto um modelo que integra dois tipos de formulação de problemas distribuídos, e que utiliza um algoritmo distribuído para a resolução dos mesmos.

Abstract

This dissertation addresses problem solving in a distributed manner, focusing on scenarios in which information is not fully available, either because it is not possible or because it is not desirable.

The two types of problems in this work are: i) the problem of task allocation, and ii) the problem of event scheduling. The problem of task allocation is illustrated in a scenario of catastrophe (allocation of fires to firemen) where agents are trying to find the best overall solution in accordance with their capacities. The problem of event scheduling with multiple resources occurs in an organization that wants to maximize the value of their employees time, while preserving the individual value (privacy) attributed to the event (value of the relative importance of the event). These problems are explored to confront the two types of resolution approaches: centralised and distributed.

The problems are formulated for resolution in a distributed and centralised manner, to evidence the characteristics and the situations in which each approach is more recommended. The performance, execution time and memory consumption, as well as the concept of privacy are points regarded in the comparison between centralised and distributed approaches.

To analyze the problem of task allocation and the problem of event scheduling, a model is proposed that includes two types of formulation for distributed problems, and that uses a distributed algorithm for their resolution.

Agradecimentos

Este espaço é dedicado àqueles que deram a sua contribuição para que esta dissertação fosse realizada. A todos eles deixo aqui o meu agradecimento sincero.

Começo por agradecer ao professor Paulo Trigo, a forma como orientou o meu trabalho. As notas dominantes da sua orientação foram a utilidade das suas recomendações e a disponibilidade que sempre demonstrou. Estou-lhe muito grato pelo encaminhamento e opiniões dadas, pois também contribuiram muito para o meu desenvolvimento pessoal. Aproveito também para agradecer a oportunidade única que me concedeu de assistir à conferência *International Conference on Autonomous Agents and Multiagent Systems* onde pude estar em contacto com a comunidade científica e aproveitar para receber novas informações que também me ajudaram na realização deste trabalho.

Em segundo lugar, agradeço ao professor Luis Morgado, uma pessoa que muito admiro pela sua dedicação e sabedoria. As aulas de complementos de inteligência artificial foram sem dúvida uma grande mais valia e muito me ajudaram. A sua disponibilidade para esclarecer as dúvidas, e as opiniões dadas foram também muito importantes para mim, mesmo a nível pessoal.

Aos meus colegas João Ferreira, Tiago Garcia e Paulo Marques muito lhes tenho a agradecer pelo companheirismo, pelas palavras de incentivo e pela constante partilha de ideias.

Quero também agradecer à minha família, principalmente aos meus pais, José Carlos de Sousa Pereira Lopes e Isabel Maria Lopes da Silva Pereira, e à minha irmã, Filipa Alexandra da Silva Pereira, pois foram eles que sempre me acompanharam ao longo de toda a vida e são o meu suporte em todos os momentos.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Abreviaturas

IA	Inteligência Artificial
DCOP	Problema de optimização de restrições distribuídas
COP	Problema de optimização de restrições
CSP	Problema de satisfação de restrições
ADOPT	Algoritmo assíncrono de optimização de restrições distribuídas
BnB-ADOP	ADOPT com ramificação e corte
GAP	Problema generalizado de afectação
BnB	*Branch and Bound*
UB	Limite superior
LB	Limite inferior
DFST	*Depth-first search tree*
EAV	*Events as variables*
PEAV	*Private events as variables*
TSAV	*Time slots as variables*

Capítulo 1

Introdução

A evolução dos sistemas distribuídos e o aumento da importância de um conjunto de problemas com características particulares e diferentes dos problemas centralizados (em que tudo depende de um agente central), motiva o desenvolvimento de métodos para formular e resolver problemas distribuídos em que seja optimizado um desempenho global.

Neste conjunto de problemas com características específicas inserem-se aqueles:

- que são naturalmente distribuídos e cuja informação necessária para os resolver não se consegue reunir toda num só local.
- que envolvem entidades que contêm internamente informação sensível (e.g. informações de custos) que não pode estar disponível por questões de privacidade.
- com um objectivo comum mas cujos agentes também tenham objectivos individuais.

Um largo conjunto destes problemas distribuídos, pode ser formulado através de variáveis e restrições entre essas variáveis, visando optimizar um desempenho conjunto, e.g. problema das redes de sensores, distribuição de veículos para cobrir um terreno, atribuição de tarefas de resgate em ambientes de catástrofes e agendamento de eventos. É neste tipo de problemas que este trabalho incide, mais especificamente nos problemas de atribuição de tarefas e problemas de agendamento de eventos.

O problema de atribuição de tarefas é ilustrado num cenário de catástrofe (atribuição de incêndios a bombeiros) onde agentes tentam encontrar a melhor solução global de acordo com as capacidades de cada um. O problema de agendamento de eventos com múltiplos recursos, ocorre numa organização que pretende maximizar o valor do tempo dos seus empregados, enquanto preserva o valor individual (privacidade) atribuído ao

evento (valor da importância relativa do evento). Estes dois problemas enquadram-se em cenários reais relevantes, que motivam o investimento na procura distribuída de soluções encontrando formulações e algoritmos adequados para os resolver.

Neste contexto surge o "problema de optimização de restrições de modo distribuído", em inglês, *distributed constraint optimization problem* (DCOP), como uma formulação onde cada variável é considerada como um agente que controla o valor dessa mesma variável. Os agentes (representando variáveis) actuam de modo distribuído comunicando através de mensagens, e procuram em conjunto definir os seus valores de modo a optimizar uma função de custo global (Modi et al., 2006). Rapidamente o DCOP se tem revelado uma técnica importante de coordenação entre agentes, porém, dependendo do problema e do algoritmo adoptado, a comunicação entre os agentes pode gerar uma grande quantidade de mensagens trocadas.

Neste trabalho é feita a comparação da resolução dos problemas de atribuição de tarefas e agendamento de eventos de um modo centralizado e de um modo distribuído, destacando as características principais que fazem com que a pesquisa distribuída seja, ou não, preferível em relação aos métodos de pesquisa centralizada. Deste modo pretendem-se identificar em que situações fará mais sentido utilizar uma abordagem distribuída tendo em conta vários critérios: i) desempenho, ii) limitação de comunicação com períodos de quebra e falhas (intermitência de comunicação), iii) memória utilizada, e iv) privacidade. Deseja-se também verificar o comportamento de um dos algoritmos mais utilizados na resolução de DCOP, *Asynchronous Distributed Constraint Optimization with Quality Guarantees* (ADOPT), quando utilizado nos dois tipos de problemas aqui tratados, para concluir que ganhos poderá trazer e em que contextos é mais adequado.

As contribuições deste trabalho alinham-se do seguinte modo:

- Sistematizar e automatizar a formulação dos problemas de atribuição de tarefas e agendamento de eventos, para resolução de modo distribuído
- Suporte à privacidade nos problemas de agendamento de eventos, para que informação sensível não tenha que estar disponível publicamente
- Ajuste do algoritmo ADOPT para suportar os problemas tratados nesta dissertação
- Avaliação dos métodos de resolução dos problemas enunciados

O modelo (DiMES-GAP) proposto neste trabalho, integra duas formulações para os problemas (atribuição de tarefas e agendamento de eventos) e efectua a resolução dos mesmos utilizando um algoritmo distribuído, permitindo deste modo a sua exploração de maneira sistemática e automática.

O essencial do trabalho desenvolvido foi apresentado em (Pereira, 2008).

1.1 Estrutura

A dissertação está organizada em 5 capítulos e 3 anexos.

Capítulo 1. Enquadra o trabalho desenvolvido e apresenta a estrutura da dissertação.

Capítulo 2. Apresenta um percurso pessoal através do estado da arte e o contexto teórico da dissertação, introduzindo os vários conceitos de suporte ao trabalho desenvolvido.

Capítulo 3. Apresenta os comparativos das diferentes abordagens e caracteriza os dois tipos de problemas base deste trabalho. São mostrados vários resultados experimentais que suportam as ideias deste trabalho.

Capítulo 4. Apresenta o modelo utilizado nesta dissertação para automatizar a tarefa de formulação e resolução de problemas através de DCOP.

Capítulo 5. Apresenta as ilações resultantes do trabalho realizado, e deixa algumas indicações do que pode ser desenvolvido futuramente.

O conteúdo central da dissertação é complementado por três apêndices. O Apêndice A apresenta a implementação feita em JAVA do algoritmo *Branch and Bound*. O Apêndice B descreve o algoritmo ADOPT utilizado na resolução dos problemas DCOP. O Apêndice C apresenta as interfaces gráficas do modelo DiMES-GAP proposto.

Capítulo 2

Estado da Arte

A procura distribuída de soluções, particularmente a área dos problemas de optimização de restrições de modo distribuído, é uma área em expansão na qual a comunidade científica tem investido.

Este capítulo apresenta o suporte teórico através da caracterização das formulações existentes, e introduz alguns desenvolvimentos recentes.

Neste sentido, apresentam-se estratégias de procura, que formam a base teórica da resolução de problemas, e os formalismos *Constraint Satisfaction Problem* (CSP) e *Constraint Optimization Problem* (COP) que são a base do formalismo *Distributed Constraint Optimization Problem* (DCOP) (Modi et al., 2003), utilizado neste trabalho para representar problemas distribuídos. O algoritmo ADOPT é também apresentado, pois actualmente é um dos algoritmos mais utilizados para resolução de problemas DCOP, sendo também adoptado neste trabalho.

No final deste capítulo é dada uma visão das evoluções e trabalhos que a comunidade científica tem feito nesta área, alguns dois quais servem de suporte às ideias desta dissertação.

2.1 Estratégias de Procura

No domínio da inteligência artificial existem inúmeros problemas que podem ser formulados através do conceito de estados e transições entre eles. Um estado representa a situação do problema naquele momento e internamente contem os atributos e valores que o caracterizam. As transições entre estados representam as relações entre os mesmos, definindo o modo como os estados estão ligados e de que forma se pode transitar

de um estado para outro. Esta representação implica que existam formas e métodos para percorrer o espaço de estados na procura de soluções (alcançar estados objectivos a partir de estados iniciais), podendo ser utilizados vários critérios nesta procura.

Por exemplo num jogo de damas um estado poderia representar uma posição no tabuleiro e a peça associada a essa mesma posição, e as transições de estado definem que posições (estados) são atingíveis no tabuleiro a partir dessa posição, obedecendo às regras de movimentação do jogo.

Este conceito de estados e transições entre eles pode ser traduzido num grafo em que os estados se representam por vértices e as transições pelas arestas. A essência da procura reside na escolha de uma opção, deixando as restantes para uma análise posterior. Sempre que existem várias opções é necessário escolher, e é o critério para esta escolha que define a estratégia de procura.

Essencialmente existem dois tipos de procura:

- procura cega ou não informada: apenas distingue um estado objectivo de outro que não o seja, em que a expansão dos estados não considera o custo do caminho já feito e não estima o custo do caminho que ainda falta percorrer.
- procura guiada ou informada: considera o custo do caminho percorrido e estima o custo do caminho que ainda falta percorrer para encontrar o objectivo.

O algoritmo 1 apresenta o pseudo-código para efectuar uma procura no espaço de estados. No seguimento, são apresentadas duas estratégias de procura cega, a procura em profundidade e procura em largura, sendo depois apresentada uma estratégia de procura informada (pesquisa com ramificação e corte) que utiliza o custo do caminho já percorrido para guiar a procura. Estas procuras partem de um estado inicial, e terminam ao chegar a um estado final ou quando exploram todo o espaço de estados sem encontrar qualquer estado final.

Algoritmo 1 Algoritmo genérico de procura

```
1:  procedure PROCURA(v_i, v_o, EstratégiaProcura)
2:      FILA ← CriarNovaFila()
3:      FILA ← FILA ⊕ v_i
4:      while FILA ≠ ∅ do
5:          e ← pop(FILA)
6:          if e == v_o then                    ▷ Verificar objectivo satisfeito
7:              return e
8:          end if
9:          e' ← expandir(e)
10:         FILA ← EstratégiaProcura(e', FILA)
11:     end while
12: end procedure
```

Notas: no algoritmo 1, v_i representa o estado inicial, v_o o estado objectivo (final) e e o estado corrente. O conjunto de estados resultantes da expansão de um estado ($expandir(e)$) é representado por e'. É importante referir que o operador $\oplus$ transforma o estado num nó da árvore de pesquisa e adiciona-o à fila.

Procura em largura. A pesquisa em largura (*breadth-first search*) tem a característica de expandir e visitar os nós por níveis de profundidade. Começa a pesquisar pelo nó inicial v_i, expandindo todos os nós descendentes, que formam a segunda camada. Antes que os nós da segunda camada comecem a ser pesquisados, todos os nós do nível de profundidade anterior tiveram que ser visitados. Deste modo, até que seja visitado o nó objectivo v_o, todos os nós de profundidade d são expandidos antes dos de profundidade $d+1$. A figura 2.1 ilustra o modo como os nós vão sendo expandidos, formando a árvore de pesquisa.

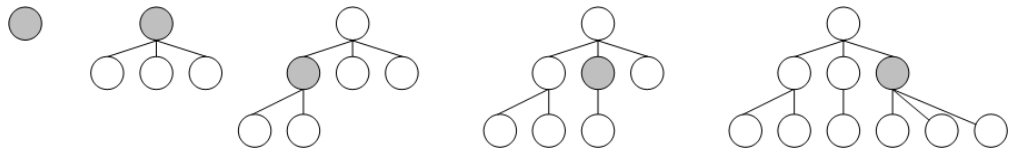

FIGURA 2.1: Exemplo do modo de expansão na procura em largura

O algoritmo 2 apresenta o pseudo-código da estratégia a utilizar no algoritmo 1 para que seja feita uma procura em largura. Para fazer a procura por camadas é utilizada uma lista em que os nós ao serem expandidos vão sendo colocados no final da mesma. Deste modo temos uma estrutura FIFO (*first in first out*) que visita primeiro todos os nós de profundidade d é só depois os de profundidade $d+1$.

Algoritmo 2 Estratégia de procura em largura

```
1: procedure EstratégiaProcura(e, FILA)
2:     return FILA ⊕ e
3: end procedure
```

A estratégia de procura em largura percorre sistematicamente cada um dos níveis da árvore de pesquisa, pelo que caso exista uma solução, esta é encontrada; o algoritmo é completo. Se existir mais do que uma solução, é garantido que neste tipo de procura a solução óptima é encontrada (se as transições entre estados tiverem custo fixo), pois é apresentada a solução com menor caminho desde a raiz; o algoritmo é óptimo. Esta procura, pode no entanto, exibir uma grande complexidade espacial e temporal devido ao factor de ramificação (expansão dos caminhos alternativos) do algoritmo, que se acentua com o aumento da dimensão do espaço de estados.

Procura em profundidade. Na procura em profundidade (*depth-first search*) os nós são visitados em sequência de profundidade. A procura em profundidade visita primeiro os nós de maior profundidade existentes da fila. Esta estratégia de procura é ilustrada na figura 2.2.

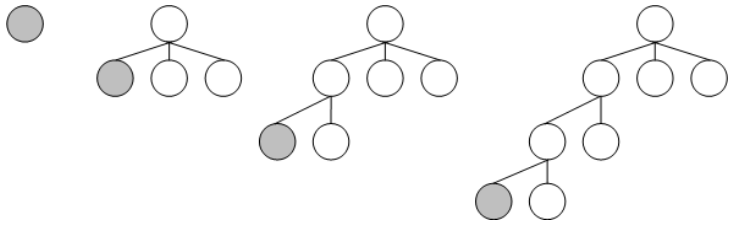

Figura 2.2: Exemplo do modo de expansão na procura em profundidade

Neste modo, a pesquisa progride para o maior nível de profundidade da árvore de procura, em que os nós não têm sucessores. Esta estratégia pode ser implementada usando uma fila LIFO (*last in first out*). O algoritmo 3 apresenta o pseudo-código da estratégia utilizada para efectuar uma procura em profundidade.

Algoritmo 3 Estratégia de procura em profundidade

```
1: procedure EstratégiaProcura(e, FILA)
2:     return e ⊕ FILA
3: end procedure
```

Esta estratégia de procura, em comparação com a procura em largura, é mais poupada no consumo de recursos ao nível da memória, porque em vez de ter que manter todos os nós em memória, necessita apenas de manter os nós da fronteira mais o caminho percorrido desde a raiz. Porém, esta procura não assegura que a solução óptima seja encontrada,

visto que pode ser dada um solução de profundidade maior que a óptima; o algoritmo não é óptimo. Pode também entrar em ciclos e o algoritmo não ter forma de sair deles; o algoritmo não é completo. No entanto com técnicas simples, pode ser transformado num algoritmo completo, bastando para isso garantir que não existem estados repetidos no caminho até à raiz (e.g. mantendo uma lista dos estados explorados).

Procura com ramificação e corte (*Branch and Bound*). O algoritmo *Branch and Bound* (BnB) é um método adequado à resolução de problemas de optimização. A ideia geral do BnB é a mesma de uma procura em largura, no entanto nem todos os nós são expandidos, (i.e. nem sempre se geram todos os descendentes de um nó).

Este algoritmo baseia-se em dois conceitos: ramificação (*branching*) e corte (*bounding*). A ramificação divide recursivamente uma procura global (conjunto de estados) em várias procuras (subconjuntos de estados) criando uma estrutura em árvore. O corte vai descartando da procura os subconjuntos das soluções candidatas que pareçam inúteis, usando para isso os limites inferior e superior já encontrados. De um modo geral, se o limite inferior de algum nó da árvore (subconjunto de nós) A é maior que o limite superior de qualquer outro nó B, o nó A pode seguramente ser retirado da procura.

O objectivo do algoritmo é procurar uma solução que minimize a função $f(x)$, em que x varia num conjunto de soluções admissíveis ou candidatas. O pseudo-código do BnB é mostrado no algoritmo 4.

Importa também referir que o BnB (tal como a pesquisa em profundidade) efectua uma procura aplicando o conceito de retrocesso (*backtracking*). Quando o caminho explorado não apresenta uma solução, a procura retorna ao ponto de escolha anterior, seguindo depois a partir desse ponto para outra alternativa. O retrocesso vai explorando os diversos caminhos até não existirem caminhos alternativos para procurar soluções, ou até encontrar a solução. Quando se dá o caso de todos os estados terem sido explorados e de não existirem alternativas, a pesquisa falha. A figura 2.3 ilustra o modo de exploração dos nós na procura com retrocesso.

Este algoritmo só precisa de guardar o caminho de procura percorrido. Tem a desvantagem de poder visitar o mesmo estado por diversas vezes, bastando para isso um estado ser atingível através de vários outros estados.

Algoritmo 4 Algoritmo *Branch And Bound*

```
1:  melhorEstado ← null
2:  melhorCusto ← +∞
3:
4:  procedure ACTUALIZARMELHOR(estado)
5:      melhorEstado ← estado
6:      melhorCusto ← estado.obterCusto()
7:  end procedure
8:
9:  procedure PROCURAR(estado)
10:     if estado.verificarEstadoCompleto() then
11:         actualizarMelhor(solucao)
12:     else
13:         sucessores ← solucao.obterSucessores()
14:         for all sucessor ∈ sucessores do
15:             if sucessor.admissivel() AND sucessor.obterLimite() < melhorCusto
    then
16:                 Procurar(sucessor)
17:             end if
18:         end for
19:     end if
20: end procedure
```

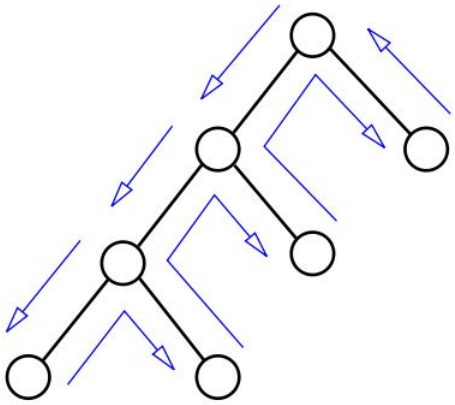

FIGURA 2.3: Procura com retrocesso

2.2 Problema de satisfação de restrições (CSP)

Um problema de satisfação de restrições (CSP) é um enquadramento geral que é utilizado para representar vários problemas na área da inteligência artificial (Yokoo, 2001). Formalmente um CSP é definido por um tuplo $< X, D, C >$ em que $X = \{X_1, X_2, ..., X_n\}$ é o conjunto das variáveis, $D = \{D_1, D_2, ..., D_n\}$ o conjunto dos domínios das variáveis, e $C = \{C_1, C_2, ..., C_m\}$ o conjunto das restrições definidas para o problema (Russell and Norvig, 2003).

Um CSP define para cada variável X_i o seu respectivo domínio D_i de valores admissíveis, tendo esse domínio a particularidade de ser finito, discreto e não vazio. Cada variável representa uma parte do problema global e o valor que assume é usado na avaliação das restrições a serem satisfeitas para o problema. Cada restrição $C_i \in C$ envolve um subconjunto de variáveis de X e define as condições que têm de ser satisfeitas na escolha dos valores a associar às variáveis desse subconjunto, para que o objectivo do CSP seja realizado (todas as restrições satisfeitas).

As restrições de um CSP podem ser de vários tipos dependendo do número de variáveis às quais a restrição é aplicada, sendo a restrição unária (aplicada a uma só variável) a mais simples (Meisels, 2008). Uma restrição binária R_{ij} efectua-se entre duas variáveis X_j e X_i, formando um subconjunto do produto cartesiano entre os seus domínios - $R_{ij} \subseteq D_j \times D_i$. As variáveis continuam a sua classificação de acordo com o número de variáveis envolvidas, implicando que uma restrição n-ária envolva as n variáveis do problema. As restrições binárias são as mais comuns , e um problema CSP representado somente com restrição deste tipo é chamado de CSP binário (Meisels, 2008).

As atribuições de valores $\{X_i = v_i, X_j = v_j, ...\}$ ao subconjunto ou à totalidade das variáveis determina o estado do problema, sendo que uma atribuição de valores às variáveis que não viole nenhuma das restrições é chamada atribuição consistente. Caso uma atribuição seja completa (inclui todas as variáveis de X) e também seja consistente é considerada solução do CSP. Quando uma atribuição não é completa pode também ser chamada de solução parcial.

Um CSP pode ter várias soluções, no entanto há casos em que não consegue ser solucionado. A figura 2.4 tem dois exemplos de CSPs e do modo como são representados. Neste exemplo, os dois CSPs têm quatro variáveis $X1, X2, X3, X4$, em que os valores (domínio) aplicados a essas mesmas variáveis podem ser: r, g, b. As restrições são representadas por linhas e como pode ser observado na figura 2.4 todas as restrições são binárias, e neste caso são restrições de desigualdade, $\{\{x1 \neq x2\}, \{x1 \neq x3\}, \{x2 \neq x4\}, \{x3 \neq x4\}\}$. Com a definição destas restrições pretende-se que as variáveis implicadas nas restrições não tenham os mesmos valores atribuídos.

A atribuição $(X1 = b, X2 = g, X3 = r, X4 = b)$ representa uma solução para o CSP da esquerda, no entanto existem mais. Para o CSP da direita não existe uma atribuição que cumpra todas as restrições $\{\{x1 \neq x2\}, \{x1 \neq x3\}, \{x2 \neq x4\}, \{x3 \neq x4\}, \{x1 \neq x4\}, \{x2 \neq x3\}\}$, logo é um CSP sem solução.

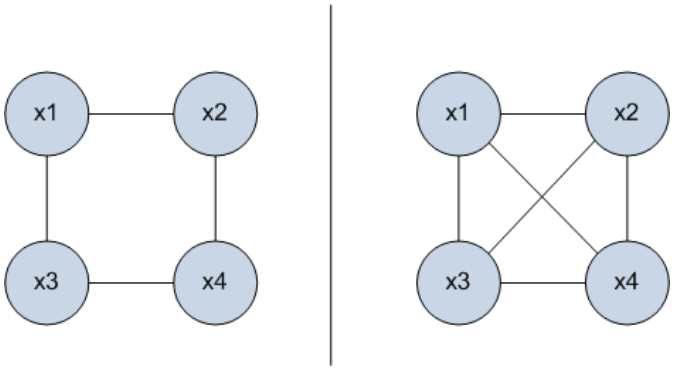

FIGURA 2.4: Exemplo de dois problemas CSP

2.3 Problema de optimização de restrições (COP)

Um problema de optimização de restrições (COP) evolui a partir dos CSP. Enquanto um CSP apresenta várias soluções que podem ser satisfatórias, um COP tem o objectivo de encontrar não só uma solução, mas sim uma solução óptima. A procura da melhor solução torna-se essencial nos casos em que a resolução satisfazendo todas as restrições não é um caso possível (e.g. CSP direito da figura 2.4), o que neste cenário é desejado encontrar a melhor solução possível. A melhor solução possível, é encontrada através da maximização ou minimização das funções de custo atribuidas ás restrições, que definem a qualidade da solução.

Formalmente, um COP é definido por um tuplo $< \mathcal{X}, \mathcal{D}, \mathcal{R} >$. $\mathcal{X}$ é um conjunto finito de variáveis $X_1, X_2, ..., X_m$ (Meisels, 2008). $\mathcal{D}$ é um conjunto de domínios $D_1, D_2, ..., D_m$. Cada domínio D_i contém um conjunto finito de valores que podem ser atribuídos à variável X_i. Para a definição completa do tuplo falta o $\mathcal{R}$, que representa um conjunto de relações entre as variáveis (restrições). Estas restrições, à semelhança das restrições dos CSP podem ser de diversos tipos consoante o número de variáveis implicadas, sendo o caso das restrições binárias o mais utilizado (restrição entre duas variáveis). A grande diferença em relação aos CSP prende-se com o facto de que neste caso, as restrições entre as variáveis, representam uma função de custo não negativa para a combinação de todos os valores de domínio das variáveis implicadas. Um COP como já foi referido, tem aspectos iguais a um CSP, no entanto a grande diferença reside na introdução da função f que mapeia a solução dos diversos tuplos S para um valor numérico (Tsang, 1993):

$$f : S \rightarrow \text{valor numérico}$$

O objectivo num problema COP é optimizar (maximizar ou minimizar) uma função de custo global F que é a agregação de todas as funções de custo f. Estas funções de

custo num CSP são definidas como funções booleanas, que somente adquirem o valor de satisfeitas/não satisfeitas, e que neste caso adquirem um valor numérico que traduz a qualidade da atribuição.

Na figura 2.5 é apresentado um exemplo de um COP, que como se trata de um caso exemplificativo, para simplificar é assumido que o domínio $D_i = \{0, 1\}$ é igual para todas as variáveis X_1, X_2, X_3, X_4. A função de custo f de forma a também simplificar, é partilhada por todas as restrições. O objectivo é minimizar a função de custo global F que é a agregação de todas as funções de custo f, ficando neste caso a função F definida como $F = f(x_1, x_3) + f(x_1, x_2) + f(x_2, x_4)$. A solução deste COP seria $\{\{x_1 = 0\}, \{x_2 = 0\}, \{x_3 = 0\}, \{x_4 = 0\}\}$, pois é a atribuição que minimiza F.

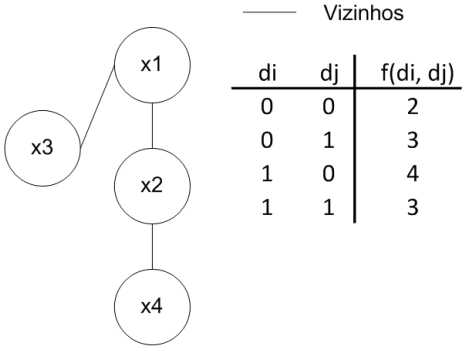

FIGURA 2.5: Exemplo de um COP

2.4 Problema de optimização de restrições distribuídas (DCOP)

Em muitas aplicações reais, partes dos problemas são conhecidos e controlados por diversas partes envolvidas (agentes), e não existe nenhuma autoridade central que tenha acesso a todo o problema. É muito comum que estes problemas sejam resolvidos por algoritmos centralizados que procuram a solução num só local, onde todas as partes do problema foram colocadas. Contudo, este tipo de abordagens muitas vezes não é possível ou desejável em determinadas circunstâncias.

O *Distributed Constraint Optimization Problem* (DCOP) é um COP numa versão distribuída, onde as variáveis são divididas por agentes $A_1, A_2, ..., A_n$ (Meisels, 2008). Formalizando, o DCOP consiste em n variáveis que formam o conjunto $V = \{x_1, x_2, ..., x_n\}$, cada uma associada a um agente, e os valores que podem ter estão contidos num domínio discreto e finito $D_1, D_2, ..., D_n$ respectivamente (Modi et al., 2003). Somente o agente designado para uma determinada variável tem controlo sobre o seu valor e conhece o

seu domínio. O objectivo é escolher os valores para as variáveis, que dada uma função objectivo, esta seja maximizada ou minimizada. A função objectivo é definida como o somatório de um conjunto de funções de custo f, em que a função de custo para um par de variáveis x_i, x_j é definida (Modi et al., 2003) como:

$$f_{ij} : D_i \times D_j \rightarrow N \tag{2.1}$$

O objectivo é encontrar uma atribuição $\mathcal{A}^*$ de valores para as variáveis, tal que o custo agregado de F é minimizado. Formalmente quere-se encontrar $\mathcal{A}(= \mathcal{A}^*)$ de modo a que $F(\mathcal{A})$ seja minimizado e onde a função objectivo F é definida por:

$$F(\mathcal{A}) = \sum_{x_i, x_j \in V} f_{ij}(d_i, d_j), \text{ onde } x_i \leftarrow d_i, x_j \leftarrow d_j \text{ em } \mathcal{A} \tag{2.2}$$

A figura 2.6 mostra um exemplo de um problema DCOP com quatro agentes, cada um responsável por uma variável de domínio {0,1}. Dois agentes x_i, x_j são vizinhos se tiverem uma restrição entre eles. Na figura 2.6, x_1 e x_3 são vizinhos mas x_1 e x_4 não. Neste exemplo, a função global F quando todas as variáveis assumem o valor zero é dada por $F(\{(x_1, 0), (x_2, 0), (x_3, 0), (x_4, 0)\}) = 4$ e no caso em que todas as variáveis assumem o valor um, fica, $F(\{(x_1, 1), (x_2, 1), (x_3, 1), (x_4, 1)\}) = 0$. A atribuição $\mathcal{A}^* = \{(x_1, 1), (x_2, 1), (x_3, 1), (x_4, 1)\}$ representa a solução, pois minimiza a função global F.

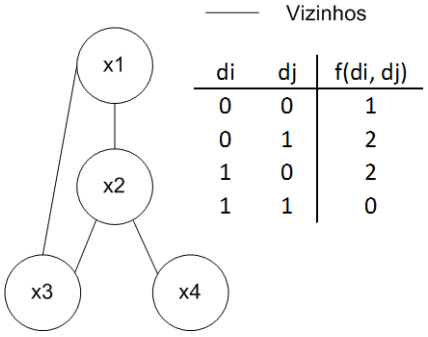

FIGURA 2.6: Exemplo de um DCOP (Modi et al., 2003)

2.5 Algoritmo assíncrono de optimização de restrições distribuídas (ADOPT)

O algoritmo ADOPT é dos algoritmos mais utilizados na resolução de DCOP, e foi o primeiro algoritmo a reunir as três seguintes características : distribuído, assíncrono e completo (Meisels, 2008, Modi et al., 2006). A característica assíncrona leva a ter

uma expectativa de conseguir maior performance (concorrência) e é um dos pontos mais importantes do ADOPT (Meisels, 2008).

O ADOPT para ser utilizado necessita de organizar os agentes numa estrutura hierárquica, em que cada agente têm um único pai mas este pode ter vários filhos, definindo deste modo a prioridade dos vários agentes. A organização dos agentes é apresentada no parágrafo seguinte.

2.5.1 Organização hierárquica dos agentes

Em inglês *depth-first search tree* (DFST), é uma estrutura em árvore de dispersão de um grafo. A partir de um qualquer grafo $G = (V, A)$, em que V é um conjunto de vértices, A um conjunto de arestas, uma DFST consiste na organização de um novo grafo conexo G' construído a partir de um subconjunto de arestas $T \subset A$ de maneira a não existirem ciclos (Kleinberg and Tardos, 2005). Numa DFST temos também $G' = (V, T)$ mas organizado de forma a que G' não tenha ciclos, ou seja, a diferença reside no modo de determinar que arestas compõem T.

Na figura 2.7 é ilustrado um grafo cuja DFST gerada está presente na figura 2.8. Dada uma DFST $G' = (V, T)$, podem definir-se as seguintes relações entre os vértices de V tendo como base as figuras 2.7 e 2.8 :

- pai: um vértice x_i é pai de outro(s) vértice(s) x_j se o mesmo está um nível acima e existe uma aresta entre ambos. e.g. x_3 é pai de x_5, x_2 é pai de x_3 e x_1.
- filho: um vértice x_j é filho de outro vértice x_i se o mesmo está um nível abaixo e existe uma aresta entre x_i e x_j.
- pseudo-pai: um vértice x_i é pseudo-pai de x_j se x_i está localizado em qualquer dos níveis acima de x_j e existe uma aresta que liga x_i e x_j em G, mas esta aresta não faz parte de G'. e.g. x_4 é pseudo-filho de x_2.
- pseudo-filho: um vértice x_j é pseudo-pai de x_i se x_i está localizado em qualquer dos níveis abaixo de x_i e existe uma aresta que liga x_j e x_i em G, mas esta aresta não faz parte de G'. e.g. x_2 é pseudo-pai de x_4.
- ascendentes: vértices ascendentes de um vértice x_i são todos aqueles que estão localizados em qualquer nível acima de x_i, no mesmo ramo da árvore. Os nós ascendentes formam um caminho de x_i até à raiz da árvore. e.g. x_3 e x_2 são ascendentes de x_5.

- descendentes: vértices descendentes de um vértice x_i são todos aqueles que estão localizados em qualquer nível inferior de x_i. Os nós descendentes formam um caminho de x_i até aos nós folhas. e.g. x_5 etem como descendente x_4, no caso de x_2 todos os restantes vértices são seus descendentes.

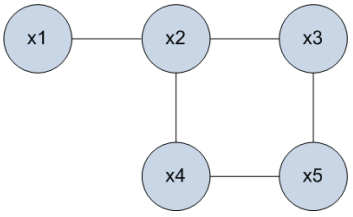

FIGURA 2.7: Grafo

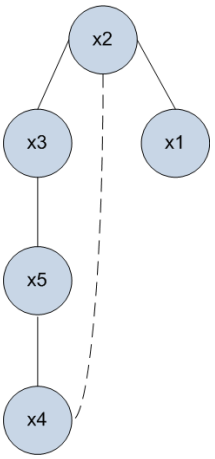

FIGURA 2.8: Estrutura de organização hierárquica

A figura 2.10 mostra uma DFST formada a partir do grafo de restrições presente na figura 2.6, em que x_1 é o nó raiz, x_1 é pai de x_2 e x_2 é pai de x_3 e x_4. A comunicação entre estes mesmos agentes é feita localmente (somente entre agentes vizinhos) seguindo também a estrutura definida na DFST.

2.5.2 Características gerais.

Este algoritmo baseia-se em limites (*lower bound* e *upper bound*) para estabelecer as estimativas de custo e direccionar o processo de procura (convergência). Estes limites são processados com informação local para cada valor do domínio. No caso da reconstrução de soluções é utilizada a noção de *threshold* (limiar), que representa o grau de aceitação de uma solução. A detecção de término do algoritmo é implícita e baseada no intervalo *lower bound*/*upper bound* que quando é zero indica a finalização.

O ADOPT é um algoritmo complexo com uma grande quantidade de pseudo-código, em vez de ser explicado o código (cf. Apêndice B), nos próximos parágrafos vão ser focadas as ideias essenciais (Meisels, 2008).

2.5.3 Mensagens envolvidas no ADOPT

Antes de focar diversos pontos do ADOPT, importa identificar os tipos de mensagens envolvidas na comunicação entre os vários agentes:

- Mensagens VALUE : este tipo de mensagens são usadas para os agentes enviarem o valor das variáveis para os vizinhos (com quem têm restrições) - um agente x_i envia mensagens VALUE para os vizinhos que se encontram num nível inferior da DFST, e somente recebem estas mensagens de vizinhos que se encontrem num nível superior na DFST.
- Mensagens THRESHOLD : estas mensagens são enviadas somente de agentes pai para agentes filhos, com um valor que representa o limiar para retroceder (*backtrack threshold*), que inicialmente é zero. As mensagens de THRESHOLD são propagadas pela DFST a baixo para reduzir redundância nas pesquisas.
- Mensagens COST : são enviadas dos agentes filhos para os agentes pais. Uma mensagem de COST enviada por x_i para os seus pais contem o custo calculado em x_i mais o custo reportado pelos filhos de x_i. x_i é informado dos custos calculados pelos agentes pertencentes à sub-árvore da qual x_i é o nó raiz. Neste tipo de mensagens vão os campos: contexto (*context*), limite inferior (*lower bound - lb*) e limite superior (*upper bound - ub*).

Para uma melhor percepção do modo de comunicação e do sentido das mensagens trocadas entre os vários agentes é apresentada a figura 2.10.

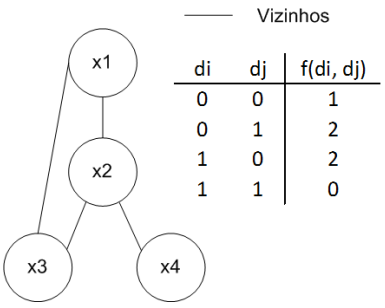

FIGURA 2.9: Grafo de restrições (Modi et al., 2003)

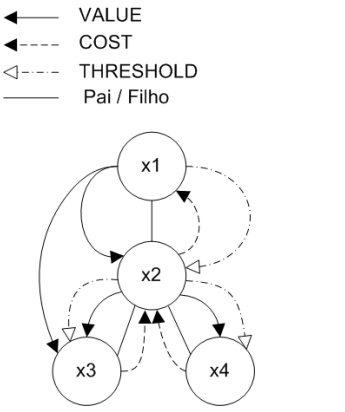

FIGURA 2.10: Mensagens envolvidas no ADOPT (Modi et al., 2003)

2.5.4 Limites no ADOPT (*lower bound* / *upper bound*).

Uma das principais bases de funcionamento do ADOPT é o cálculo dos limites superior (*upper bound - UB*) e inferior (*lower bound - LB*) por cada agente. Para que o ADOPT possa ser executado, como já foi referido, existe a necessidade de organizar os agentes numa estrutura hierárquica (DFST). Cada agente tem a responsabilidade de calcular os limites para o seu sub-problema, informando à posteriori o seu pai dos limites calculados.

Deste modo o problema global é dividido em diversos sub-problemas e as responsabilidades são repartidas pelos vários agentes. A organização dos agentes em árvore define também a prioridade dos mesmos, sendo que um agente num nível superior da árvore tem mais prioridade que um agente presente num nível inferior da mesma. Em seguida são apresentados vários pontos com as definições mais importantes para perceber o cálculo dos limites:

- Um contexto (*context*) é definido como uma solução parcial, e é representado na forma $\{(x_i, d_i), (x_j, d_j), ...\}$. Um contexto não pode conter a mesma variável mais que uma vez e dois contextos são compatíveis se todos os valores atribuídos às variáveis forem iguais. Um contexto corrente (*current context*) é um contexto local a um determinado agente, em que nesse contexto estão as atribuições dos agentes vizinhos. Na figura 2.10, como podemos observar, o *CurrrentContext* de x_3 seria $\{(x_1, 0), (x_2, 0)\}$. No caso de x_1, como é o agente raiz, o seu *CurrrentContext* é sempre vazio (não recebe mensagens VALUE).

- O custo local $\delta(d_i)$ é o custo calculado de escolher o valor $d_i \in D_i$ para o agente x_i, mais o somatório de todos os custos das restrições entre x_i e os vizinhos de maior nível da árvore. Formalmente, o custo local é dado por:

$$\delta(d_i) = \sum_{(x_j,d_j) \in CurrentContext} f_{ij}(d_i, d_j) \tag{2.3}$$

  Por exemplo, na figura 2.10, supomos que x_3 recebe mensagens com o valor de x_1 e x_2 a 0. Deste modo fica no *CurrentContext* de x_3 $\{(x_1, 0), (x_2, 0)\}$, se x_3 escolher o valor 0 para si, o valor de custo local é:

$$\delta(0) = f_{1,3}(0, 0) + f_{2,3}(0, 0) = 1 + 1 = 2$$

  em que $f_{1,3}(0, 0)$ é a função de custo aplicada à restrição de x_3 com x_1, e $f_{2,3}(0, 0)$ é a função de custo aplicada à restrição de x_3 com x_2.

- $LB(d)$ é o limite inferior (*lower bound*) para a sub-árvore em que x_i é o agente raiz. O cálculo de $LB(d)$ tem em conta o *CurrentContext* e o valor d escolhido para o agente x_i. A definição de $LB(d)$ é:

$$\forall d \in D_i, LB(d) = \delta(d) + \sum_{x_l \in Children} lb(d, x_l) \tag{2.4}$$

- $UB(d)$ é o limite superior (*upper bound*) para a sub-árvore em que x_i é o nó raiz. O cálculo de $UB(d)$ tem em conta o *CurrentContext* e o valor d escolhido para o

agente x_i. A definição de $UB(d)$

$$\forall d \in D_i, UB(d) = \delta(d) + \sum_{x_l \in Children} ub(d, x_l) \tag{2.5}$$

- LB é o limite inferior para a variável x_i e representa o custo mínimo da solução para a sub-árvore em que x_i é o agente raiz, seja qual for o valor escolhido para x_i. Isto significa que qualquer solução para um problema com o limite inferior LB, tem um custo de pelo menos LB. Formalmente LB é definido por:

$$LB = min_{d \in D_i} LB(d) \tag{2.6}$$

- UB é o limite inferior para a variável x_i e representa o custo máximo da solução para a sub-árvore em que x_i é o agente raiz, seja qual for o valor escolhido para x_i. Para um problema com um limite superior de UB, a solução para o mesmo terá um custo máximo de UB que é expresso por:

$$UB = min_{d \in D_i} UB(d) \tag{2.7}$$

No ADOPT, cada agente através de mensagens VALUE recebe os valores das atribuições feitas dos agentes de maior prioridade, e com isto, é responsável por calcular os limites (LB e UB) para o sub-problema em que este é o nó raiz. Os valores dos limites são continuamente refinados ao longo do tempo e são reportados ao agente pai através de mensagens COST. Como o algoritmo é assíncrono, o importante é que o intervalo dos limites seja admissível e mesmo que não exista informação, um $LB = 0$ e um $UB = \infty$ é sempre um intervalo válido. Deste modo os limites vão sendo refinados através das mensagens reportadas, e por sua vez a informação passada aos agentes pai, vai sendo mais precisa. O processo de pesquisa (convergência da solução) é direccionado pelos valores destes limites (LB e UB) que à medida que a pesquisa evolui, tenta-se que o intervalo entre LB e UB vá diminuindo. Quando chega ao ponto em que os valores de LB e UB se encontram (o intervalo entre LB e UB é zero), foi encontrada uma atribuição que forma a solução óptima para o problema (sub-problema).

Cálculo dos limites (LB e UB). Cada agente para suportar os diversos cálculos, mantem internamente para $d \in D_i$ e para todos os $x_l \in Children$ os seguintes campos:

- $context(d, x_l)$ é o contexto corrente que x_l enviou da última vez.
- $lb(d, x_l)$ é o limite remetido por x_l para o sub-problema em que x_l é o nó raiz. Este limite é calculado a partir do *CurrentContext* de x_l.

- $ub(d, xl)$ é a mesma definição do ponto anterior mas para o limite superior.
- $t(d, xl)$ é o limiar (*threshold*) associado ao filho x_l.

Olhando para a figura B.1, na linha 15, podemos ver que ao ser recebida uma mensagem de VALUE, o *CurrentContext* é actualizado, e também é verificado se existem incompatibilidades entre o *CurrentContex* e o $context(x, x_l)$. Caso exista alguma incompatibilidade (fig. B.1, linha 17) as estruturas de suporte dos limites (UB e LB), do limiar (*threshold*) e contexto (*context*) são reinicializadas.

Quando um determinado agente x_l recebe de um agente filho uma mensagem de COST, esta transporta os valores dos limites (LB e UB) e o respectivo contexto (*context*) em que x_l se baseou para o seu cálculo. Para que a informação seja sempre compatível (fig. B.1, linha 28) é verificado se o contexto recebido do agente x_l é compatível com o contexto corrente (*CurrentContext*) do agente actual (x_i). A informação que não é compatível é reiniciada a zero, e a informação $(lb(d, x_l), ub(d, x_l), t(d, x_l))$ compatível é guardada e usada nos cálculos de $UB(d)$ e $LB(d)$ para $d \in D_i$, podendo depois calcular-se LB e UB através destes.

Atribuição de valores às variáveis. Inicialmente os agentes definem o valor para as variáveis de maneira concorrente, em que cada um escolhe o valor de uma maneira "gananciosa" (escolhe naquele momento o melhor valor de acordo com o custo) e envia através de mensagens VALUE para os vizinhos de menor prioridade (agentes com quem tem restrições). Como o algoritmo é assíncrono existem muitos modos de evolução a partir do ponto em que cada agente avisa os seus vizinhos da escolha do valor. Em (Modi et al., 2006, cap. 4.2) é apresentado um exemplo de execução dos vários passos do algoritmo. Não sendo tão exaustivo, cada agente escolhe o valor para a sua variável e informa a sua escolha através de mensagens VALUE. Após isto, os agentes calculam os limites (LB e UB) e comunicam-nos constantemente aos seus pais (através de mensagens COST), sem esperar primeiro que todo o espaço esteja explorado e os limites sejam calculados de forma mais precisa. Este comportamento leva a que o problema seja sub-dividido e a responsabilidade dos sub-problemas vá passando para os nós de menor prioridade, cujo objectivo é arranjar a atribuição com menor custo (com limite inferior menor). Os diversos nós vão calculando os limites e informam em cadeia os nós pela DFST acima. Ao serem explorados os sub-problemas, os limites inferiores vão crescendo até ao ponto em que se efectua uma comutação no valor da variável, porque essa mudança corresponde a um valor mais promissor (com um menor limite inferior). Após troca do valor numa variável são enviadas mensagens a informar do sucedido, sendo a informação propagada pela árvore de pesquisa abaixo.

Mecanismo de *threshold* (limiares). Como forma de optimizar a revisitação de espaços, o ADOPT utiliza o mecanismo de *threshold* (Modi et al., 2003). Este campo é iniciado com zero e representa o valor actual do limite inferior (LB) ou o limite inferior (LB) anteriormente conhecido do sub-problema em que o agente corrente é a raiz.

O valor do *threshold* de x_q é armazenado no seu pai x_p e pode ser actualizado internamente em x_q ou através de mensagens de THRESHOLD recebidas de x_p.

Como um agente só pode guardar informação consistente com o contexto corrente (*CurrentContext*), no caso de um agente com mais prioridade (e.g. o seu pai) mudar o valor da variável, e em seguida decidir voltar ao valor anterior, é perdido o limite inferior (LB) previamente calculado (Meisels, 2008). No entanto o último valor que o agente filho x_q reportou e ficou guardado em x_p, é útil para recomeçar os cálculos, aumentando a eficiência da pesquisa e evitando ter que reiniciar as estruturas utilizadas para guardar os valores dos limites e *thresholds*.

O pseudo-código que suporta este mecanismo de *thresholds* é apresentado na figura B.3, e um exemplo do seu funcionamento é dado em (Modi et al., 2006, cap. 4.3).

2.5.5 Aspectos da implementação ADOPT

A implementação do algoritmo ADOPT, usada neste trabalho para resolver problemas DCOP, baseou-se na proposta por Modi (Modi et al., 2006). No entanto, esta implementação não contempla alguns aspectos definidos na formulação DCOP, como as funções de custo gerarem valores naturais, para os pares de valores das restrições. As funções de custo definidas no DCOP são apresentadas em 2.1, no entanto a implementação ADOPT segue a definição seguinte:

$$f_{ij} : D_i \times D_j \rightarrow \{0, 1\} \tag{2.8}$$

A implementação ADOPT com esta característica limita a definição das funções de custo, permitindo somente especificar pares de valores admissíveis (utilizando o valor 0) ou não admissíveis (utilizando o valor 1).

As funções de custo caracterizadas deste modo, afectam o cálculo do custo local:

$$\delta(d_i) = \sum_{(x_j,d_j)\in CurrentContext} f_{ij}(d_i, d_j) \tag{2.9}$$

que é utilizado no cálculo dos limites. Para que esta implementação possa ser usada na resolução dos problemas formulados nesta dissertação, e para que cumpra a especificação DCOP, houve necessidade de alterar a forma de avaliação das funções de custo. Este aspecto é explicado em 4.4.2.

2.6 Outros trabalhos na área

BnB-ADOPT. A investigação do DCOP tem-se centrado na melhoria dos algoritmos existentes e no desenvolvimento de novos, comparando-os com os já existentes. O algoritmo ADOPT é actualmente o algoritmo que mais tem sido tomado como base de utilização e comparação. Os esforços de modificação e refinação, têm originado várias derivações, como é o caso do algoritmo *Branch-and-Bound DCOP Algorithm* (BnB-ADOPT) (Yeoh et al., 2008). Este algoritmo baseia-se na estrutura e no sistema de comunicação do ADOPT, mas adopta uma estratégia de procura *depth-first branch-and-bound* em vez da utilizada no ADOPT (*best-first*).

Privacidade e pré-processamento. Estes algoritmos utilizados no DCOP seguem uma abordagem multiagente tendo em conta a sua execução num só domínio público, sem preocupações em relação à privacidade o que conduz também a algum investimento no estudo deste aspecto da privacidade (Greenstadt et al., 2007). Outro tema de destaque na investigação é o uso de técnicas de pré-processamento para tentar aumentar a velocidade de execução desses mesmos algoritmos, nomeadamente do ADOPT (Ali et al., 2005).

DiMES. A complexidade associada ao conjunto de problemas aqui referidos torna muitas vezes difícil a sua adaptação a problemas complexos reais. Devido a este aspecto começam a surgir algumas optimizações e *frameworks* que facilitam a formulação e resolução de problemas mais adaptados à realidade. Neste contexto, para formalizar uma classe de problemas referentes ao agendamento de eventos com múltiplos recursos, surge a formulação *Distributed Multi-Event Scheduling* (DiMES) (Maheswaran et al., 2004).

DCOPolis. Avaliar e comparar os algoritmos utilizados no DCOP torna-se por vezes complicado pelo facto de estarem implementados em simulação, e não há na literatura registos de comparações e aplicação dos algoritmos em ambientes distribuídos reais (Sultanik et al., 2007). Os algoritmos estão implementados em simuladores individuais que têm diferenças entre eles e que adicionam efeitos indesejáveis às comparações devido às diferenças de implementação. Idealmente, deveria existir uma *framework* com uma única

implementação dos algoritmos utilizados no DCOP. Essa *framework* deveria permitir a execução dos algoritmos em modo de simulação ou num modo fisicamente distribuído. A plataforma DCOPolis (Sultanik et al., 2007) tem como objectivo suprimir estes vários aspectos identificados e funcionar como bancada de desenvolvimento para DCOP.

Capítulo 3

Abordagem centralizada e distribuída

Este capítulo enquadra dois tipos de problemas, i) atribuição de tarefas, e ii) escalonamento de eventos. Estes tipos de problemas são formulados usando, respectivamente, a formulação *Generalized Assignment Problem* (GAP) (secção 3.2), e a formulação *Distributed Multi-Event Scheduling* (DiMES) (secção 3.5.1). Ambos os problemas podem ser vistos como problemas COP utilizando uma abordagem centralizada, ou problemas DCOP utilizando uma abordagem distribuída.

As abordagens centralizada e distribuída são analisadas de modo a adequar as abordagens às características dos problemas, enquadrando-os no contexto de dois cenários.

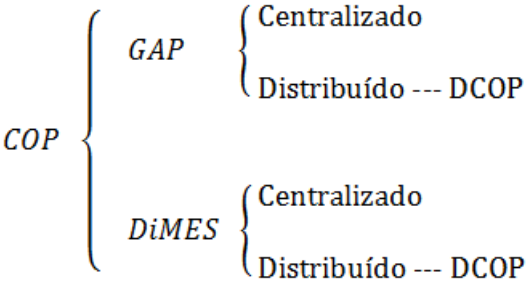

FIGURA 3.1: Diagrama dos problemas e formulações

3.1 COP e DCOP

Em geral, quando se caracterizam abordagens centralizadas e distribuídas para resolver problemas, os factores que se têm em conta são o desempenho a nível de consumos de

memória e do tempo de processamento. Existe no entanto, um outro factor que pode fazer toda a diferença quando há necessidade de escolher qual vai ser a abordagem utilizada. A privacidade pode muitas vezes ser preferida em detrimento de outra característica como o desempenho. Este conceito de privacidade só consegue ser alcançado utilizando DCOP, mas é necessário verificar as implicações que se tem ao nível do desempenho (memória e tempo de execução). É precisamente neste ponto que nos focaremos, verificando de acordo com a natureza, formulação e características dos problemas, qual a aproximação mais adequada.

Em seguida apresenta-se a formulação GAP, utilizada na representação do problema de atribuição de tarefas. Esta formulação é válida para qualquer problema de atribuição de tarefas, no entanto será aqui ilustrada a sua aplicação recorrendo a um cenário simplificado do *RoboCup Rescue* (secção 3.3.2).

3.2 Problema generalizado de afectação - GAP

Um problema generalizado de afectação (GAP), em inglês *Generalized Assignment Problem*, enquadra de forma genérica, os problemas de alocação de tarefas entre agentes (Shmoys and Tardos, 1993, Scerri et al., 2005). Num problema de afectação de tarefas pretende-se determinar que agentes devem realizar um conjunto de tarefas, de modo a que a afectação feita tenha a maior qualidade possível. Esta qualidade é definida de acordo com as capacidades que os agentes têm para realizar diversas tarefas, tentando atribuir tarefas a agentes com maior capacidade para as realizar.

Para formular um problema GAP começamos por definir um conjunto $\theta = \{\theta_1, .., \theta_m\}$ de tarefas a atribuir a um conjunto $E = \{e_1, .., e_n\}$ de agentes. Cada agente $e_i \in E$ tem determinadas capacidades para executar cada uma das tarefas $\theta_j \in \theta$ definidas por $Cap(e_i, \theta_j) \rightarrow [0, 1]$. Os agentes têm também uma capacidade global definida por $e_i.res$, sendo que e_i gasta $Res(e_i, \theta_j)$ da sua capacidade global na execução da tarefa θ_j. Convencionalmente é definida uma matriz de afectação A onde a_{ij} é:

$$a_{ij} = \begin{cases} 1, & \text{se } e_i \text{ realiza } \theta_j, \text{ e} \\ 0, & \text{caso contrário.} \end{cases} \tag{3.1}$$

O objectivo é encontrar a afectação A no espaço, A', de todas as possíveis matrizes de afectação, de forma a maximizar a utilidade da afectação das tarefas para o conjunto de agentes; assim A é definido por:

$$A = \arg\max_{A'} \sum_{e_i \in E} \sum_{\theta_j \in \theta} Cap(e_i, \theta_j) \times a_{ij} \tag{3.2}$$

As capacidades globais dos agentes devem no entanto ser respeitadas, (i.e. não se podem atribuir mais tarefas do que as que o agente tem possibilidade de realizar). Deste modo temos:

$$\forall e_i \in E, \sum_{\theta_j \in \theta} Res(e_i, \theta_j) \times a_{ij} \leq e_i.res \tag{3.3}$$

em que cada tarefa só pode ser atribuída a um agente:

$$\forall \theta_j \in \theta, \sum_{e_i \in E} a_{ij} \leq 1 \tag{3.4}$$

3.3 Formulação GAP como DCOP

3.3.1 Restrições entre agentes

Da formulação GAP (sec. 3.2), passamos para o mapeamento DCOP (restrições entre agentes) da seguinte forma (dos Santos, 2007):

- cada agente $a_i \in A$ tem correspondência a uma variável $x_i \in V$ no DCOP.
- para construir o domínio D_i de cada variável, começamos por definir um domínio global D formado por todas as possíveis atribuições de tarefas a agentes (todas as tarefas de θ) (secção 3.2). O domínio D_i para o agente x_i é formado pelos elementos $d \in D$ para os quais o agente x_i possua recursos para efectuar todas as tarefas θ_k especificadas em d. Formalmente, $\forall d \in D_i, \sum_{\theta_k \in d} Res(x_i, \theta_k) \leq x_i.res$ (cf. secção 3.2).
- as funções de custo f_{ij} correspondem à minimização da função "soma das capacidades", que é calculada para duas variáveis x_i e x_j, de acordo com os valores dos domínios d_i e d_j. A definição GAP tem como objectivo encontrar uma solução que maximize a soma das capacidades, mas o DCOP está formulado de modo a minimizar custos. Deste modo, atendemos a que para qualquer função $g(x)$:

$$\text{maximizar } g(x) = -\text{ minimizar } -g(x)$$

Na tabela 3.9 temos a função "soma das capacidades" $sc(x)$ a maximizar , o que implica que a função de custo a minimizar, deverá ser $f(x) = -sc(x)$. Há no entanto outro aspecto a considerar: o DCOP só admite funções custo em $\mathbb{N}$, portanto não se pode fazer simplesmente $-sc(x)$, pois passaríamos a ter funções com números negativos. Assim, o que há a fazer é encontrar o maior valor da "soma das capacidades" em que não existe sobreposição de tarefas, e considerar esse o "zero". O valor máximo da soma das capacidades é dado por:

$$M_{ij} = \max(\sum_{\theta_k \in d_i} Cap(x_i, \theta_k) \times a_{x_i,\theta_k} + \sum_{\theta_k \in d_j} Cap(x_j, \theta_k) \times a_{x_j,\theta_k}) \text{ se } a_{x_i,\theta_k} \neq a_{x_j,\theta_k} \tag{3.5}$$

No caso da tabela 3.9 o maior valor da soma das capacidades é 4. Logo, deviamos ter -4, mas como não podemos ter negativos, o valor -4 será considerado 0 (zero). Formalizando, as funções de custo entre duas variáveis x_i e x_j definem-se por:

$$f_{ij} = \begin{cases} M_{ij} - \left(\begin{array}{l} \sum_{\theta_k \in d_i} Cap(x_i, \theta_k) \times a_{x_i,\theta_k} + \\ \sum_{\theta_k \in d_j} Cap(x_j, \theta_k) \times a_{x_j,\theta_k} \end{array} \right) & \text{se } a_{x_i,\theta_k} \neq a_{x_j,\theta_k} \\ \infty & \text{caso contrário.} \end{cases} \tag{3.6}$$

Nesta função, o primeiro termo ($a_{x_i,\theta_k} \neq a_{x_j,\theta_k}$) define uma condição que resulta da especificação GAP e que impossibilita a existência de sobreposição de tarefas por dois agentes.

- a função objectivo $\mathcal{F}(\mathcal{A})$ corresponde ao somatório das funções de custo para cada par de agentes x_i, x_j, dada a atribuição $\mathcal{A}$. Uma solução corresponde a uma atribuição de valores às variáveis tal que a função objectivo tenha o valor mínimo. Formalmente:

$$\mathcal{F}(\mathcal{A}) = \sum_{x_i, x_j \in V} f_{x_i,x_j} \tag{3.7}$$

3.3.2 Capacidades e recursos dos agentes

Para ilustrar este mapeamento, irá ser utilizado o *RoboCup Rescue*, utilizando somente agentes do tipo brigadas de bombeiros atribuindo-lhes a tarefas de combate aos incêndios. O parágrafo seguinte descreve de um modo geral o cenário *RoboCup Rescue*.

Cenário *RoboCup Rescue*. O simulador *RoboCup Rescue* (Kitano et al., 1999) apresenta um cenário de catástrofe após um terramoto, em que paramédicos e equipas de resgate têm a difícil tarefa de intervir devido a rupturas e bloqueamento de ruas, edifícios desmoronados, pessoas feridas e existência de focos de incêndio. Neste contexto existe também a dificuldade de conseguir informação precisa para saber onde e como actuar. O objectivo é que os diversos agentes (forças policiais, paramédicos e brigadas de bombeiros) estejam coordenados e tenham um plano conjunto que apresente a melhor solução para controlar os incêndios, resgatar os feridos e desbloquear as ruas. Neste cenário cada agente tem percepção limitada do que o rodeia e a comunicação entre os vários agentes é muitas vezes limitada e sujeita a falhas. Além do ambiente hostil é acrescentada a dificuldade destes cenários serem dinâmicos, podendo existir novos incêndios, novos colapsos de edifícios que provocam obstrução de ruas, e agravamento dos ferimentos dos refugiados. O planeamento e coordenação dos agentes deve responder da melhor forma às alterações de cenário para tentar minimizar os danos globais. Para este trabalho irá utilizar-se um cenário simplificado do referido anteriormente, em que se pretender atribuir às brigadas de bombeiros as tarefas de apagar os focos de incêndio.

FIGURA 3.2: Cenário *Robocup Rescue*

Após apresentar o cenário completo do *RoboCup Rescue*, para este trabalho vai ser utilizado o cenário simplificado ilustrado na figura 3.3, em que os incêndios formam o conjunto $\theta = \{\theta_1, \theta_2, \theta_3\}$ de tarefas. Neste caso temos três agentes que representam as brigadas de bombeiros e formam o conjunto $A = \{a_1, a_2, a_3\}$.

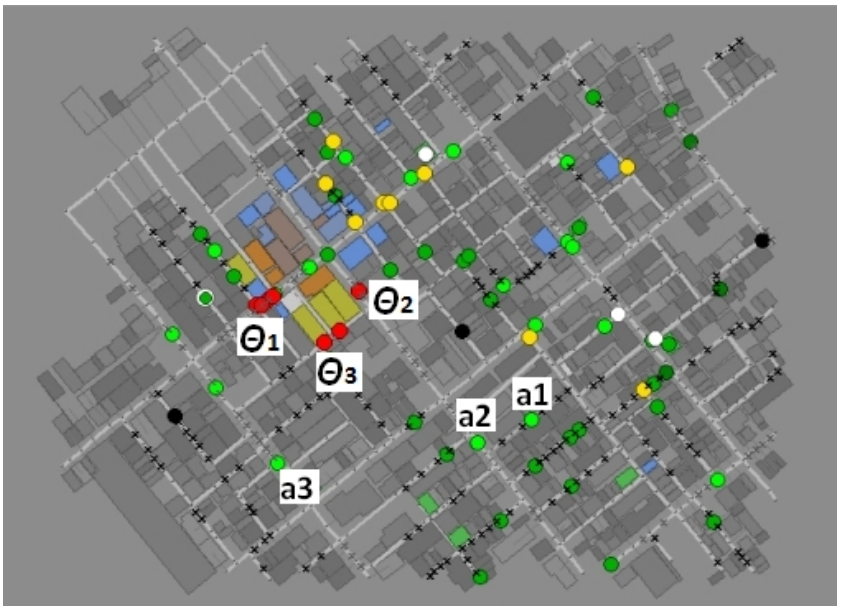

FIGURA 3.3: Cenário *Robocup Rescue* simplificado

***RobCup Rescue* como GAP.** A formulação GAP tem três fases:

- A primeira fase consiste em definir a matriz que representa as capacidades globais dos vários agentes. Neste cenário os agentes representam brigadas de bombeiros, pelo que a sua capacidade representa uma métrica da quantidade de água existente nos tanques. A matriz de capacidades globais considerada para este cenário está apresentada na tabela 3.1.

- A segunda fase é a da definição das matrizes de capacidades e custos de cada agente por tarefa. No caso da definição da matriz de capacidades do agente por tarefa, são considerados vários aspectos que influenciam o desempenho do agente. A distância a que a brigada se encontra do incêndio é uma métrica para cálculo dos valores da matriz de capacidades dos agentes por tarefa; esta matriz de capacidades dos agentes por tarefa é apresentada na tabela 3.2.

- A terceira fase é a da definição da matriz de consumos dos agentes por tarefa. Esta é a matriz representa os recursos consumidos por cada agente na concretização de uma tarefa. Um referencial utilizado para definir os valores desta matriz é estimativa da quantidade de água gasta em cada incêndio; a matriz de consumos dos agentes por tarefa é apresentada na tabela 3.3.

Matriz recursos	
Agente	Capacidades
a1	3
a2	2
a3	4

TABELA 3.1: Matriz global de recursos

Matriz de capacidades - Cap			
Agente	Tarefa		
	t1	t2	t3
a1	2	1	0
a2	1	1	1
a3	2	0	0

TABELA 3.2: Matriz de capacidades dos agentes

Matriz de consumos - Res			
Agente	Tarefa		
	t1	t2	t3
a1	1	2	1
a2	1	0	1
a3	1	3	1

TABELA 3.3: Matriz de consumos dos agentes

Após a descrição do mapeamento, podemos começar por definir o conjunto de variáveis DCOP, sabendo que cada agente GAP corresponde a uma variável no DCOP. Define-se um conjunto DCOP $V = \{a_1, a_2, a_3\}$. Para manter a coerência da notação utilizada na secção 3.3.2, na especificação das variáveis em DCOP vai ser utilizado a_i em vez de x_i.

Depois de definido o conjunto das variáveis, passamos à atribuição do seu domínio. A tabela 3.4 mostra a domínio global D. Nesta tabela, os valores a "1" indicam tarefa alocada e a "0" não alocada. Por exemplo, a primeira linha da tabela contém todas as três tarefas atribuídas.

t1	t2	t3
1	1	1
1	1	0
1	0	1
1	0	0
0	1	1
0	1	0
0	0	1
0	0	0

TABELA 3.4: Domínio global D

A partir do domínio global, podemos especificar os domínios para cada um dos agentes, verificando quais as atribuições presentes em D que os agentes conseguem satisfazer, tendo em conta os recursos consumidos para realizar cada tarefa sem exceder as suas capacidades globais. Para auxiliar, podemos definir uma tabela como a seguinte:

Domínio			a1 (recursos = 3)		a2 (recursos = 2)		a3 (recursos = 4)	
t1	t2	t3	consumo	possível?	consumo	possível?	consumo	possível?
1	1	1	4	não	2	sim	5	não
1	1	0	3	sim	1	sim	4	sim
1	0	1	2	sim	2	sim	2	sim
1	0	0	1	sim	1	sim	1	sim
0	1	1	3	sim	1	sim	4	sim
0	1	0	2	sim	0	sim	3	sim
0	0	1	1	sim	1	sim	1	sim
0	0	0	0	sim	0	sim	0	sim

TABELA 3.5: Tabela de recursos consumidos por agente para $d \in D$

Após a verificação da tabela podemos definir os domínios para os vários agentes, que resulta dos $d \in D$ cujos agentes podem satisfazer (e.g. etiqueta "sim" na tabela 3.5). Como resultado dos vários domínios temos:

t1	t2	t3
1	1	0
1	0	1
1	0	0
0	1	1
0	1	0
0	0	1
0	0	0

TABELA 3.6: Domínio do agente a_1

t1	t2	t3
1	1	1
1	1	0
1	0	1
1	0	0
0	1	1
0	1	0
0	0	1
0	0	0

TABELA 3.7: Domínio do agente a_2

t1	t2	t3
1	1	0
1	0	1
1	0	0
0	1	1
0	1	0
0	0	1
0	0	0

TABELA 3.8: Domínio do agente a_3

Depois de terem sido definidos os vários domínios dos agentes, a última fase do mapeamento para DCOP consiste no cálculo das funções de custo. Como já foi referido, as funções de custo são calculadas para cada par de agentes a_i, a_j, e correspondem à minimização da soma das capacidades dos dois agentes na realização das tarefas (combinação das tarefas de d_i e d_j). Desta forma, pode ser construída uma tabela de suporte destes cálculos. A tabela 3.9 apesar de incompleta, exemplifica o modo de cálculo da função de custo f_{a_1,a_2}. A tabela vai ser formada pela combinação dos domínios d_1 e d_2, que de acordo com a alocação das tarefas definidas em cada um, define uma capacidade. A capacidade total é a soma das capacidades individuais e é este valor que vai ser usado na função de custo f_{a_1,a_2} caso não exista sobreposição de tarefas (condição proveniente da formulação GAP). No caso de existir sobreposição de tarefas nos agentes, a função de custo f_{a_i,a_j} vai ter o valor ∞. Para os outros casos, toma-se como referência o valor máximo da soma das capacidades quando não há sobreposição, e a este valor é subtraído cada valor da soma. Exemplificando, para a linha sete da "função de custo" da tabela 3.9, a função f_{a_1,a_2} vai ter o valor 0 porque é a subtracção do valor 4 (valor máximo da

função "soma das capacidades" quando não existe sobreposição de tarefas) pelo valor 4 da soma das capacidades deste caso particular.

Agente 1				Agente 2						
Domínio			capacidades	Domínio			capacidades	soma das capacidades	sobreposição de tarefas?	função custo
t1	t2	t3		t1	t2	t3				
1	1	0	3	1	1	1	3	6	sim	∞
				1	1	0	2	5	sim	∞
				1	0	1	2	5	sim	∞
				1	0	0	1	4	sim	∞
				0	1	1	2	5	sim	∞
				0	1	0	1	4	sim	∞
				0	0	1	1	4	não	0
				0	0	0	0	3	não	1
1	0	1	2	1	1	1	3	5	sim	∞
				1	1	0	2	4	sim	∞
				1	0	1	2	4	sim	∞
				1	0	0	1	3	sim	∞
				0	1	1	2	4	sim	∞
				0	1	0	1	3	não	1
				0	0	1	1	3	sim	∞
				0	0	0	0	2	não	2

TABELA 3.9: Exemplo da Função de custo f_{a_1,a_2}

Tendo como base este exemplo, há necessidade de efectuar o mesmo processo para os outros pares de agentes para formar f_{a_1,a_3} e f_{a_2,a_3}. Quando concluídos os cálculos para cada uma das funções de custo, temos:

$$\mathcal{F}(\mathcal{A}) = f_{a_1,a_2} + f_{a_1,a_3} + f_{a_2,a_3}$$

e o problema está representado em DCOP.

Na próxima secção é utilizado este mapeamento para gerar problemas DCOP a partir de GAP. São utilizadas três modos de resolução de problemas GAP (dois centralizados e um distribuído) para confrontar as características de uma resolução centralizada e distribuída.

3.4 Resolução do problema GAP

3.4.1 ADOPT

Em seguida vão ser analisados os resultados de testes ao desempenho a nível de consumo de memória e tempo de execução de problemas GAP como DCOP, utilizando o algoritmo ADOPT.

Na definição destes problemas GAP começou-se por definir problemas com três agentes e três tarefas e foi-se sucessivamente aumentando a complexidade (em número de agentes e tarefas) para poder testar a evolução do comportamento.

Os testes seguintes foram efectuados recorrendo a um computador com processador intel centrino Core 2 Duo (T7500 - 2,2 Ghz) com 2GB de memória. Os testes executados nesta secção têm em conta o desempenho ao nível do tempo de execução e ao nível dos consumos de memória. No parágrafo seguinte é explicado como é feita a estimativa do consumo de memória por parte dos agentes no ADOPT.

Consumo de memória No ADOPT, a memória utilizada por agente está directamente relacionada com a quantidade das variáveis e cardinalidade dos respectivos domínios. Para efeitos de análise, é feita uma aproximação do valor máximo de memória (em *bytes*) que um agente precisa no ADOPT, considerando três aspectos:

A memória necessária para guardar o contexto corrente do agente.

B memória necessária para guardar o limiar e os limites (t e lb e ub)

C memória necessária para guardar o contexto de cada filho (*context*)

em que:

$$\text{MemóriaPiorCaso} = A + B + C \tag{3.8}$$

Cada agente precisa de ter uma estrutura para guardar o contexto corrente (*Current-Context*). Sabendo que o contexto corrente é construído por um par identificador do agente, e o seu valor, assumindo dois valores inteiros a quatro *bytes* para este suporte, e que cada agente pode ter um máximo de $n-1$ agentes no seu contexto, iremos ter:

$$A = (n-1) \times \underbrace{(2 \times 4)}_{\text{2 variáveis} \times 4\ bytes} \tag{3.9}$$

Além do contexto corrente, cada agente (x_i) tem que armazenar $lb(d_i, x_l)$, $ub(d_i, x_l)$, $t(d_i, x_l)$ (cf. secção 2.5) para cada valor de domínio $d_i \in D_i$ (domínio do agente x_i) e filho x_l, o que adiciona ao cálculo de memória anterior, o seguinte:

$$B = \underbrace{3}_{lb,ub,t} \times (n-1) \times \dim(D_i) \times \underbrace{(2 \times 4)}_{\text{2 variáveis} \times 4\ bytes} \tag{3.10}$$

Nota: Para efeitos de estimativa, assume-se que um agente pode ser pai de todos os outros, daí o uso do factor $(n-1)$ na expressão.

Para ficar completa a aproximação de memória máxima necessária por agente, tem que ainda ser adicionado o $context(d_i, x_l)$ para cada filho e para cada valor de domínio. Este campo pode definir-se através da parcela 3.9 multiplicada pelo número de valores do domínio, em semelhança com o que foi feito na parcela anterior. Deste modo, podemos estimar a memória máxima consumida para guardar o contexto dos agentes filhos:

$$C = A \times \dim(D_i) \tag{3.11}$$

Resultados experimentais Na tabela 3.10 são apresentados os valores estimados do consumo de memória por agente e os tempos de execução, resultantes da experimentação. Nesta tabela são apresentados para o mesmo problema dois tipos de execução:

- a de todos os agentes num fio de execução (uma *thread*) que obriga a execução síncrona do algoritmo
- a execução de um agente por fio de execução (n *threads*) que simula o paralelismo de execução

ADOPT				
Número de Tarefas	Tempo total de execução (segundos)		Máximo de memória utilizada por agente (bytes)	Memória utlizada quanto todos os agentes estão na mesma máquina (bytes)
	Um agente por thread	Todos os agentes numa thread		
3	0,17	0,06	208	624
4	1,07	0,44	408	1.632
5	6,03	4,67	672	3.360
6	134,40	115,20	1.000	6.000
7	8.208,00	7.218,00	1.392	9.744

TABELA 3.10: Resultados da resolução dos problemas GAP com ADOPT

A nível do tempo necessário para resolução, podemos verificar que a partir do problema GAP com seis tarefas e seis agentes, existe um grande aumento de tempo de execução quando comparado com o problema GAP definido com cinco tarefas.

Ao nível de consumo de memória, se o problema GAP for resolvido de maneira fisicamente distribuída, a memória necessária para todo o problema é dividida por todos os agentes, no entanto se o algoritmo correr com todos os agentes na mesma máquina, a memória

necessária é a multiplicação do número total de agentes pela memória consumida por agente.

Em seguida é mostrado um gráfico com a evolução dos tempos resultantes dos testes, um com os valores respeitantes à execução de todos os agentes num só fio de execução (uma *thread*), e o outro com a execução dos agentes em fios de execução distintos (*n threads*).

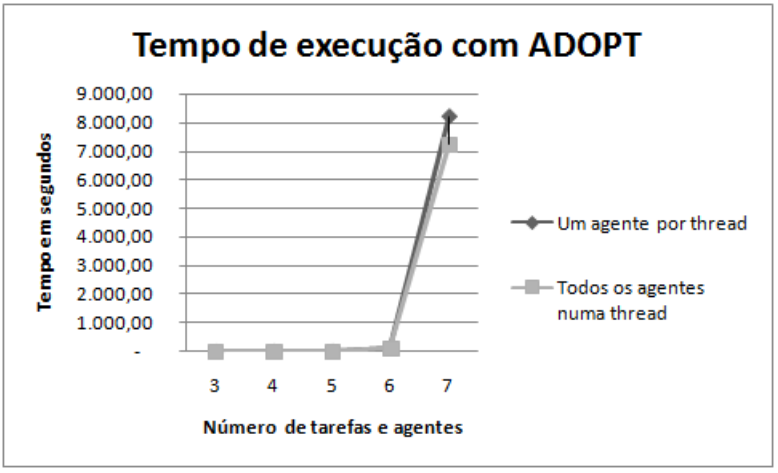

FIGURA 3.4: Gráfico com os tempos de execução do ADOPT

Conclusões. Após a análise dos resultados dos testes, podemos concluir que neste caso concreto de problemas GAP mapeados para DCOP, conforme aqui foram apresentados, são levantadas duas questões críticas quanto à complexidade do DCOP gerado: a dimensão dos domínios e a densidade do grafo de restrições.

A dimensão dos domínios cresce exponencialmente com o número de tarefas a alocar. No pior caso, onde os agentes tenham recursos para poder alocar todas as tarefas simultaneamente, a dimensão do domínio de cada variável será 2^{θ}, em que θ representa o número de tarefas. Para casos mais simples como ter três tarefas a alocar, vamos ter um domínio com tamanho máximo de $2^3 = 8$ valores. No entanto, se tivermos problemas maiores, como por exemplo quinze tarefas a serem alocadas, implica domínios na ordem dos $2^{15} = 32768$, o que numa perspectiva real é uma situação normal.

As funções de custo também crescem exponencialmente com o número de agentes, pois ao especificar restrições entre cada par de agentes a_i, a_j, os grafos de restrições vão ter uma grande densidade e vão aumentando o tempo necessário para a resolução.

3.4.2 Pesquisa com cortes

A resolução de problemas de atribuição de tarefas de uma maneira descontrolada, pesquisando todas as soluções possíveis, mesmo em problemas simples, torna-se incomportável tanto a nível de consumo de memória (necessita conter todos os nós em memória), como em tempo de execução.

Formulação do problema GAP para pesquisa com cortes. A primeira abordagem ao problema, para que o algoritmo BnB possa funcionar, é formular o problema de modo a poder ser expandido como árvore de pesquisa, existindo dependência entre estados de vários sub-espaços.

A primeira ideia é desenvolver um processo que ajude na escolha de quais os estados a expandir, e que possivelmente levem a melhores soluções. Cada estado é caracterizado por um vector que tem as tarefas e os respectivos agentes atribuídos, e por um campo que indica a profundidade associada. Assim podemos expandir os estados tendo em conta as tarefas, começando por verificar qual o melhor agente (com mais capacidade) a atribuir à primeira tarefa, e seguindo assim sucessivamente pelas outras tarefas. Esta forma de expansão adiciona uma dependência entre os estados a expandir, podendo assim tirar vantagem de não continuar a explorar uma solução que se verifique ser pouco favorável. Ter esta dependência, é importante porque evita a expansão de ramos desnecessários. Esta operação usa o custo acumulado do caminho percorrido desde o estado inicial até ao estado actual, para saber se irá ou não expandi-lo.

Implementação do algoritmo de pesquisa com cortes. O BnB introduz as noções de estado possível e de limites de solução, como forma de reduzir o número de nós que precisam ser expandidos. Neste caso concreto do problema de atribuição de tarefas, um estado é possível se a atribuição (tarefa a agente) feita nesse estado não excede a capacidade do agente.

De forma a verificar se um determinado caminho tem um custo dentro dos limites de solução, é necessário implementar a função *getBound*, que devolve o custo de uma solução (parcial ou final), e tem como objectivo auxiliar na decisão de não expandir um estado, cujo custo vai ser decididamente pior ou igual ao melhor encontrado até ao momento.

A actualização do melhor estado encontrado só pode ser feita quando um estado é completo e ainda quando esse estado tem um custo associado menor que o anterior. Um estado é completo se todas as decisões já tiverem sido tomadas. Neste contexto verifica-se quando a profundidade do estado é igual ao número de tarefas do problema. Esta

noção de estado completo é suportado no algoritmo através da função *isComplete*, que como foi referido antes, vai ser a condição para actualizar o melhor estado e respectivo valor de custo alcançado até ao momento.

Antes de passar à exemplificação da formulação do problema e funcionamento do algoritmo, importa também identificar o conceito de estado admissível, (função *isFeasible*), e que tem o mesmo objectivo que a utilização dos limites, ou seja, excluir da pesquisa ramos desnecessários. A sua aplicação evita que estados que até possam ter um custo dentro dos limites, sejam pesquisados, quando num determinado contexto não são considerados praticáveis. Por exemplo, um estado é considerado não admissível se a atribuição de tarefas aos agentes excede a capacidade local de algum deles.

Exemplo. Para melhor compreender o funcionamento do algoritmo BnB e a sua adaptação neste contexto de optimização de tarefas, em seguida é apresentado um exemplo. O problema a resolver é caracterizado pelas seguintes matrizes, em tudo semelhantes às apresentadas nas secções anteriores.

Matriz recursos	
Agente	Recurso
a1	2
a2	2

Tabela 3.11: Matriz de recursos globais dos agents

Matriz de capacidades			
Agente	Tarefa		
	t1	t2	t3
a1	2	1	2
a2	2	2	1

Tabela 3.12: Matriz de capacidades dos agentes por tarefa

Matriz de consumos			
Agente	Tarefa		
	t1	t2	t3
a1	1	2	2
a2	1	1	2

Tabela 3.13: Matriz de consumos dos agentes por tarefa

Antes de passar a alguns detalhes, é necessário clarificar a maneira de formular o problema de atribuição de tarefas no contexto do *RoboCup Rescue* (cf. secção 3.3.2) para ser resolvido através deste algoritmo.

Ao representar o problema do *RoboCup Rescue* de forma a poder ser resolvido através do BnB, há necessidade de adicionar uma matriz auxiliar ao conjunto de matrizes definidas para os problemas GAP. Esta matriz auxiliar é utilizada para inverter a matriz de capacidades dos agentes, pois neste caso queremos reflectir o custo dos agentes em vez da utilidade, para que tenhamos um modo de cortar ramos da pesquisa ao verificar se o custo acumulado está dentro dos limites. Com isto, o processo de corte, que usa como referência a melhor solução conhecida até ao momento, impede que estados cuja solução parcial tenha um custo maior que a solução de referência, sejam expandidos, porque não vão ser encontradas soluções melhores naquele caminho. Com esta técnica, alguns ramos da árvore são descartados sem que se perca tempo de processamento ao tentar pesquisar um caminho que não será rentável logo à partida.

Como já foi referido existe necessidade de adicionar ao conjunto das matrizes anteriores, uma matriz auxiliar, que vai ser o inverso da matriz de capacidades e vai ser chamada matriz de custos. Esta matriz é apresentada na tabela 3.14.

Matriz de custo			
Agente	Tarefa		
	t1	t2	t3
a1	0	1	0
a2	0	0	1

TABELA 3.14: Matriz de custos

Depois de identificadas as matrizes que representam o problema, na figura 3.6 é mostrada a árvore totalmente expandida. A expansão de um determinado estado é suportada no algoritmo através da função *getSuccessors*, que retorna os estados alcançáveis a partir de um estado actual. A figura 3.5 apresenta um exemplo do funcionamento da função *getSucessors*.

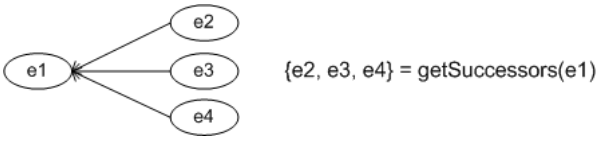

FIGURA 3.5: Aplicação do operador de expansão (*getSucessors*)

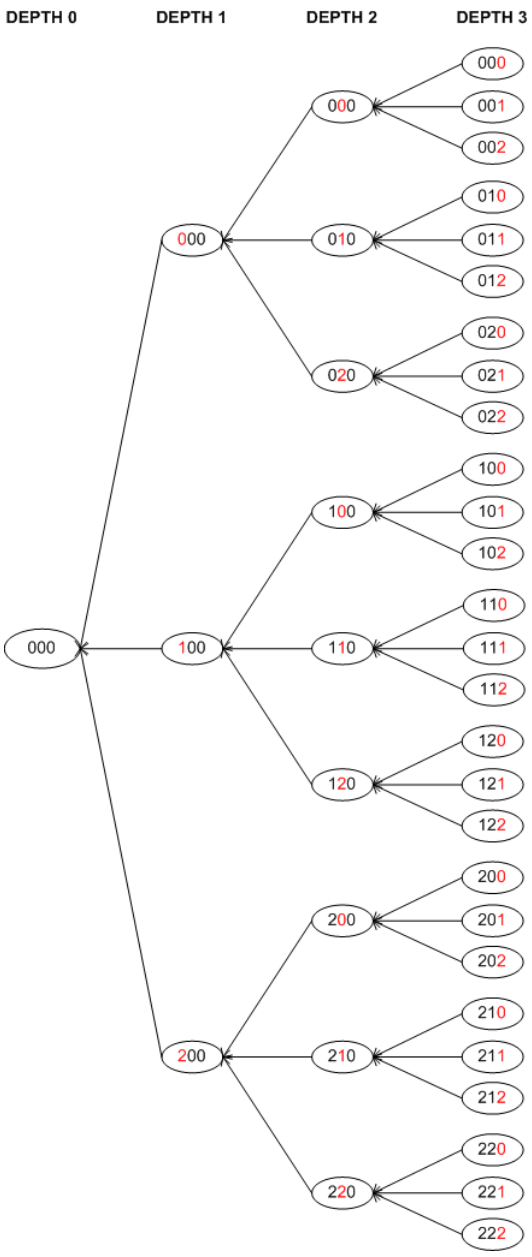

FIGURA 3.6: Modo de expansão da árvore de pesquisa

Como podemos observar, a árvore total resultante deste problema têm vinte e sete estados finais que resultam da expansão de todos os caminho. No entanto o que se pretende demonstrar é que com algumas optimizações podemos abandonar a exploração de alguns ramos, optimizando a pesquisa. Podemos observar que no total teremos trinta e nove estados que constituem todo o espaço de pesquisa, composto por vinte e sete estados finais e doze estados intermédios.

Utilizando todos os conceitos apresentados no início desta secção, e resolvendo o problema caracterizado pelas matrizes anteriores, em vez de ser necessário pesquisar todos

os estados, a pesquisa vai ser mais optimizada e vão existir ramos da árvore que não precisam ser explorados. Este comportamento é ilustrado na figura 3.7.

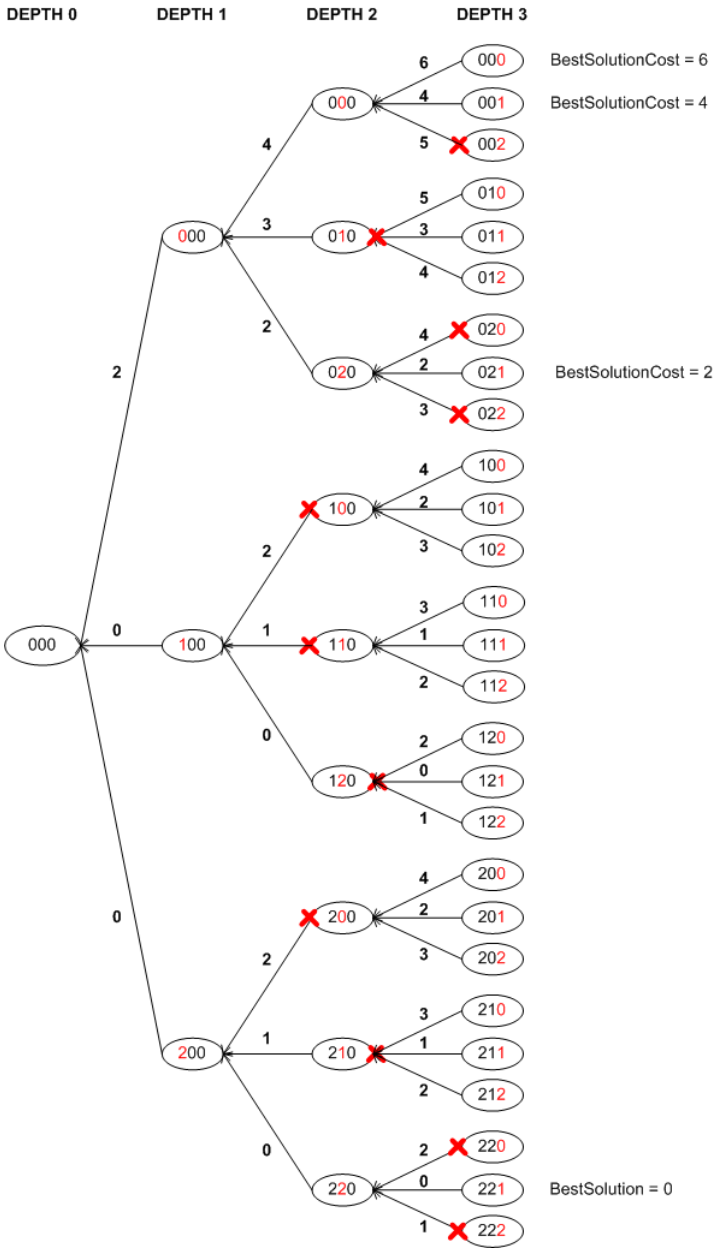

FIGURA 3.7: Resultado das optimizações

De acordo com o algoritmo 4.2, verificamos que a solução inicial começa com o custo máximo e a pesquisa é iniciada no estado (0, 0, 0), que ao ser expandido origina os estados com profundidade um. Este algoritmo, efectua sucessivamente uma pesquisa em profundidade até encontrar um estado que não permita avançar ou seja um estado final, que implica o recuo e exploração do próximo caminho (utilizando a técnica de *backtracking*).

Começando pelo estado (0,0,0), vai sendo sucessivamente feita a expansão até ser atingido o estado final (0,0,0) de profundidade três, isto porque a condição do algoritmo 4.15 vai sendo verificada. Ao atingir um estado final, a condição do algoritmo 4.4 vai ser averiguada para deduzir se o estado encontrado tem menor custo que o estado de referência. Neste exemplo, o estado final (0,0,0) vai ter um custo de seis, porque o custo de não atribuir uma tarefa é igual ao maior custo da tabela. Após explorar o estado final (0,0,0) o custo da melhor solução encontrada é actualizado e é definido um novo limite de referência para a procura e expansão dos estados.

Explorando o estado seguinte (0,0,1) vamos baixar novamente o limite para custo igual a quatro, o que implica que ao explorar o estado (0,0,2) seja efectuado um corte na pesquisa, pois continuar a pesquisa nesse ramo não iria trazer uma solução melhor.

Assim vamos continuando a percorrer os ramos da árvore e chegamos ao ramo (1,0,0) de profundidade um, cujo primeiro sucessor vai ser cortado visto que o custo não irá melhorar mais que dois e este é a melhor solução encontrada até ao momento. O segundo sucessor também não vai ser explorado mas por motivos diferentes da situação anterior, porque apesar de ser uma ramo que poderá trazer melhorias no custo da solução, trata-se de um estado que falha a exploração porque não é um estado possível.

Ao observar as matrizes, verificamos que se as tarefas um e dois fossem atribuídas ao agente um, iria existir custo superior à sua capacidade global, logo como isso não é possível o caminho deixa de continuar a ser explorado.

Influência da formulação do problema na eficiência. Este algoritmo está desenvolvido para tirar partido da formulação do problema e do modo como o operador expande os estados. Assim sendo, neste caso a sua eficiência depende fortemente do modo como se define a matriz de capacidades dos agente por tarefa e da matriz de capacidades globais dos agentes.

Para simplificação, irá assumir-se que a matriz de consumos dos agentes por tarefa, vai ser igual à matriz de capacidades dos agentes por tarefa. Assim, um agente com determinada capacidade para realizar uma tarefa, consome esse mesmo valor se a concretizar.

A matriz de consumos dos agentes por tarefa não influi na eficiência do algoritmo mas um dos mecanismo de corte utilizados neste algoritmo é não expandir estados em que os agentes não têm capacidade suficiente para realizar todas as tarefas que lhe foram atribuidas. Deste modo, se a matriz de capacidades global dos agentes tiver valores que possibilitem a atribuição de uma grande quantidade de tarefas ao agente, ou no limite todas, faz com que este mecanismo de corte não traga nenhuma optimização.

Os casos em que o problema expanda de um modo em que não encontra inicialmente uma boa referência dos limites de corte, faz com que existam mais estados que têm de ser explorados. Muitas vezes se inicialmente é encontrada um boa referência de custos, conseguem-se boas optimizações, pois muitos caminhos não têm de ser explorados.

Em seguida para exemplificar como o modo de formulação do problema altera a eficiência, é apresentado um problema idêntico ao anterior mas com os valores alterados. As tabelas seguintes representam o problema alterado.

Matriz recursos	
Agente	Recurso
a1	4
a2	8

Tabela 3.15: Matriz global de recursos alterada

Matriz de capacidades			
Agente	Tarefa		
	t1	t2	t3
a1	1	1	2
a2	3	2	3

Matriz de consumos			
Agente	Tarefa		
	t1	t2	t3
a1	1	1	2
a2	3	2	3

Tabela 3.16: Matriz de capacidades e Matriz de consumos alteradas

Neste exemplo, a alteração das matrizes influi directamente no número de estados expandidos, e essa alteração pode na pior das hipóteses implicar uma pesquisa em todos os estados. Na tabela 3.17 é apresentado um comparativo do número de estados expandidos na resolução do problema inicial e na resolução do problema alterado. Neste caso simples podemos comprovar que os mecanismos de corte do algoritmo, não conseguem ter tanta eficiência.

Matriz	Número de estados expandidos
Inicial	16
Alterada	21

Tabela 3.17: Comparativo dos estados expandidos nos dois problemas

Resultados experimentais. Após apresentar o modo de aplicação do algoritmo de pesquisa com cortes aos problemas GAP, vão ser mostrados os resultados dos testes efectuados. Para a execução dos testes são usados os mesmos problemas definidos anteriormente para a execução com o ADOPT. Antes de apresentar a tabela com os resultados obtidos, é mostrado na figura 3.18 a estrutura de memória utilizada para representar cada nó. De maneira semelhante à estrutura em memória utilizada para representar um nó na pesquisa completa,têm-se a parte genérica (representação de um nó) e a parte específica dependente do problema (representação do estado de um nó). O vector de

atribuição das tarefas a agentes é o factor que influencia o consumo de memória por nó, pois o tamanho do vector depende do número de tarefas do problema.

Objectos no Branch And Bound		
SolutionNode		
Campo	**Tipo**	**Size(bytes)**
depth	int	4
Overhead do Objecto		8
Total		12
Campos de caracterização do estado		
Campo	**Tipo**	**Size(bytes)**
tasksVector	int[N_TASKS]	12
Total		12
TOTAL		**24**

TABELA 3.18: Cálculo da memória consumida pelos objectos do algoritmo BnB

A tabela 3.19 contém os valores de consumo de memória e tempos de execução resultantes dos testes efectuados com este algoritmo. Como já foi referido, os problemas vão progressivamente aumentando de complexidade tanto em número de tarefas como em número de agentes. É importante referir que os três problemas apresentados nas últimas linhas da tabela, definidos por quinze tarefas e quinze agentes, são utilizados somente para demonstrar como os valores das matrizes que definem o problema alteram o desempenho do algoritmo.

Pesquisa com cortes						
Número de Tarefas	Número total de nós possiveis	Número total de nós expandidos	Tempo médio para execução e expansão de cada nó (ms)	Tempo total de execução (ms)	Memória utilizada por nó (bytes)	Total de memória máxima (bytes)
2	13	8	0,25	2,00	20	140
3	85	12	0,25	3,00	24	312
4	781	23	0,25	5,75	28	588
5	9.331	63	0,25	15,75	32	992
6	137.257	70	0,25	17,50	36	1.548
7	2.396.745	236	0,25	59,00	40	2.280
8	48.427.561	6.231	0,25	1.557,75	44	3.212
9	1.111.111.111	60.798	0,25	15.199,50	48	4.368
...	...	...	...	...	...	...
15	1.229.782.938.247.300.000	34.530.836	0,30	10.359.250,80	72	17.352
15	1.229.782.938.247.300.000	41.857.513	0,30	12.557.253,90	72	17.352
15	1.229.782.938.247.300.000		0,30	0,00	72	17.352
Legenda:						
	> Não foi encontrada uma solução.Teve em execução 5.248 minutos e 35 segundos e não conseguiu encontrar a solução.					

TABELA 3.19: Resultados da resolução dos problemas GAP com algoritmo *Branch and Bound*

Nota: como este problema é expandido para uma estrutura em árvore, antes de chegar aos estados finais, têm que ser explorados estados intermédios, o que aumenta o número de estados possíveis em relação a uma pesquisa completa. O número de estados possíveis depende de dois factores, o factor de ramificação da árvore (b) e a profundidade da mesma (L). Estes factores são definidos por: $b = (\text{número de agentes} + 1)$ e $L = (\text{número de tarefas} + 1)$. O número total de estados possíveis é dado por:

$$\text{NúmeroEstadosTotal} = (1 - b^L)/(1 - b) \tag{3.12}$$

Como esta pesquisa precisa guardar somente o caminho que está a explorar e os nós que proporcionam alternativas, o consumo de memória depende da profundidade da árvore e do factor de ramificação:

$$\text{Máximo memória} = ((b \times L) + 1) \times \text{MemóriaPorNó} \tag{3.13}$$

Na aplicação deste algoritmo aos problemas de atribuição de tarefas consegui-se cortar da pesquisa um grande número de caminhos. Esta relação de estados possíveis *versus* estados expandidos é ilustrado no gráfico 3.8. Podemos observar através do gráfico, que para um problema definido por nove tarefas e nove agentes, exploraram-se apenas 60.798 dos 1.111.111.111 estados possíveis do problema, o que se traduz num corte de 1.111.050.313 estados da pesquisa.

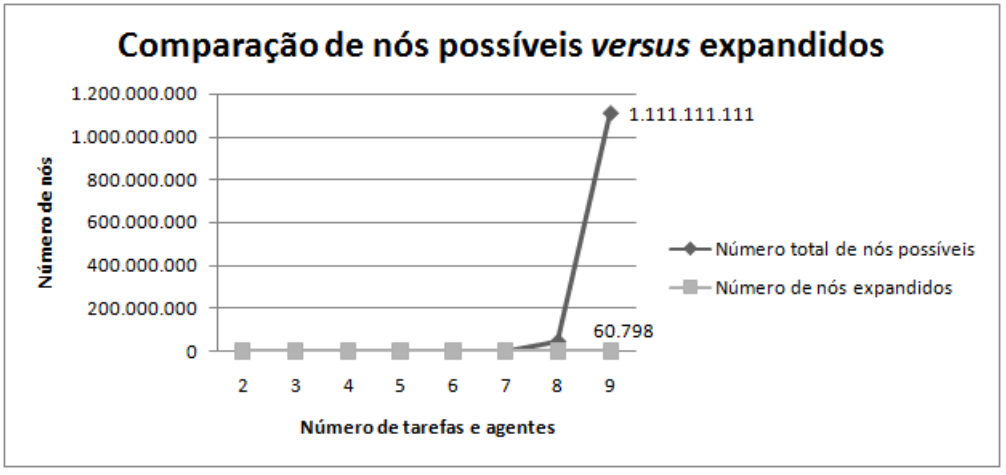

FIGURA 3.8: Gráfico com o comparativo dos nós possíveis *versus* nós expandidos

O gráfico seguinte mostra a evolução da memória de acordo com o aumento da complexidade do problema, e podemos ver que o consumo de memória neste algoritmo é

baixo. Mesmo para um problema definido por quinze tarefas e quinze agentes teríamos um consumo máximo de memória de cerca de 17 *Kbytes*.

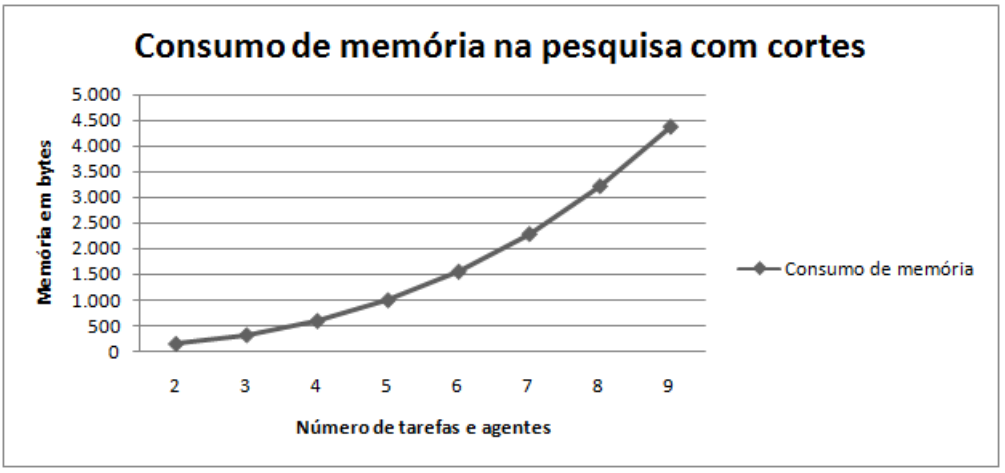

FIGURA 3.9: Gráfico com o consumo de memória na pesquisa com cortes

Fazendo a análise ao tempo de execução deste algoritmo, verificamos que também são conseguidos tempos de resolução baixos quando comparados com a execução através do ADOPT e da pesquisa completa. A evolução dos tempos de execução em relação ao aumento da complexidade dos problemas é ilustrada no gráfico 3.10.

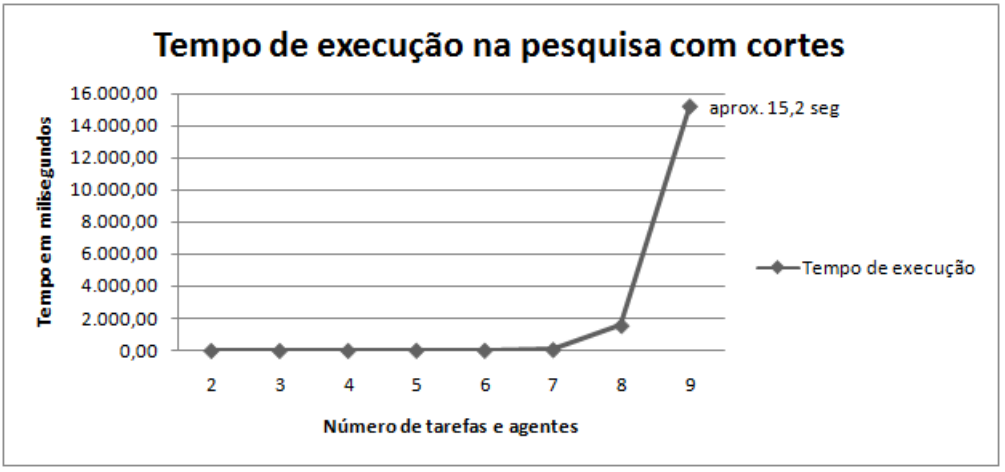

FIGURA 3.10: Gráfico com o tempo de execução na pesquisa com cortes

3.4.3 Pesquisa completa

Para ter um padrão do pior caso, resolvendo problemas GAP de forma centralizada, vamos considerar uma procura completa a todos os estados. Devido à natureza do problema de atribuição de tarefas, não se depreendeu uma heurística que guiasse a pesquisa,

pelo que resultou numa pesquisa completa.

Como se trata de uma resolução que tem por base um método genérico de pesquisa, o consumo de memória é definido pela parte genérica e pela parte particular de cada problema. A parte genérica é a representação de cada nó da árvore de pesquisa (E_Node). A parte específica é a representação de cada estado do problema (E_State). Neste caso particular cada estado do problema é definido por um vector que indica para cada tarefa o número do agente designado para a realizar, ou indica que nenhum agente foi atribuído aquela tarefa. De modo a suportar cada estado da pesquisa é definida a estrutura de memória ilustrada na figura 3.20.

Objectos na pesquisa completa		
E_Node		
Campo	**Tipo**	**Size(bytes)**
depth	int	4
f_cost	float	4
g_cost	float	4
h_cos	float	4
parent	E_Node	4
state	I_State	4
Overhead do Objecto		8
Total		32
E_State		
Campo	**Tipo**	**Size(bytes)**
Overhead do Objecto		8
tasksVector	int[N_TASKS]	12
Total		20
TOTAL		**52**

TABELA 3.20: Cálculo da memória consumida pelos objectos do algoritmo de procura completa

Identificando as partes de memória envolvidas neste contexto, verifica-se que a principal variável que influencia o consumo de memória é o vector de representação de cada estado, que vai aumentando o seu tamanho de acordo com o número de tarefas definidas para os problemas.

Resultados experimentais A tabela 3.21 apresenta os dados resultantes dos testes efectuados. Os testes foram efectuados com base nos problemas GAP definidos anteriormente, cuja complexidade vai aumentando gradualmente, incrementando o número de tarefas e agentes. A pesquisa a efectuar é completa (explora todos os estados) e mantém todos os estados em memória. O número total de estados depende do número de tarefas e agentes do problema:

$$\text{NúmeroEstadosTotal} = (\text{NúmeroAgentes} + 1)^{\text{NúmeroTarefas}}$$

Nota: como existe a possibilidade de uma tarefa não ser alocada adiciona-se um ao número de agentes.

Pesquisa completa

Número de Tarefas	Número total de nós possiveis	Número total de nós expandidos	Tempo médio para execução e expansão de cada nó (ms)	Tempo total de execução (ms)	Memória utilizada por nó (byte)	Total de memória (bytes)
2	8	8	0,545	4,4	48	384
3	63	63	0,159	10,0	52	3.276
4	624	624	0,055	34,3	56	34.944
5	7.775	7.775	0,039	303,2	60	466.500
6	117.648	117.648	0,086	10.070,7	64	7.529.472
7	2.097.151	2.097.151	1,100	2.306.866,1	68	142.606.268
8	43.046.720	43.046.720	1,000	43.046.720,0	72	3.099.363.840
9	999.999.999	999.999.999	1,000	999.999.999,0	76	75.999.999.924
10	25.937.424.600	25.937.424.600	1,000	25.937.424.600,0	80	2.074.993.968.000

Legenda:
> Os valores não foram obtidos por experiência, foram estimados

TABELA 3.21: Resultados da resolução dos problemas GAP com algoritmo de pesquisa completa

Esta procura apresenta uma evolução exponencial, tanto a nível de consumo de memória como a nível de tempo de execução. Como podemos observar pela tabela e pelos gráficos das figuras 3.11 e 3.12, uma execução considerada simples de um problema GAP definido por nove agentes e nove tarefas apresenta resultados incomportáveis a nível de memória e de tempo necessário para a sua resolução.

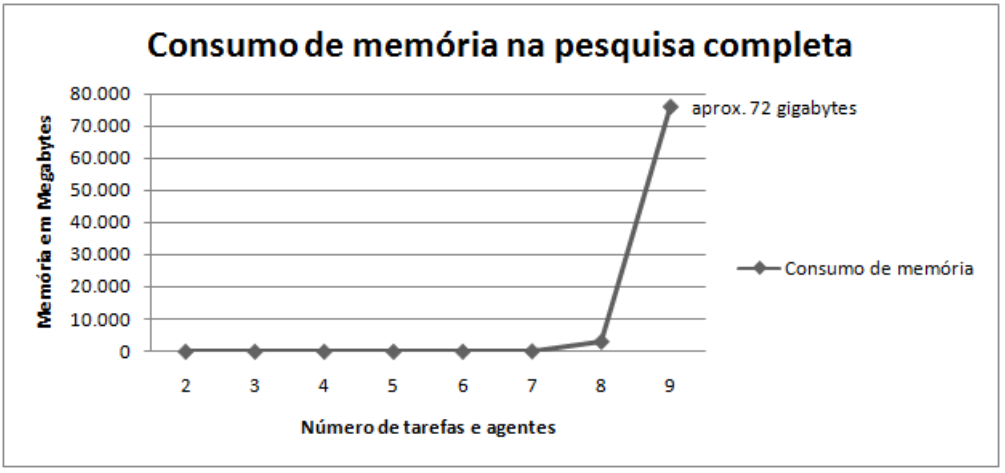

FIGURA 3.11: Gráfico do consumo de memória na pesquisa completa

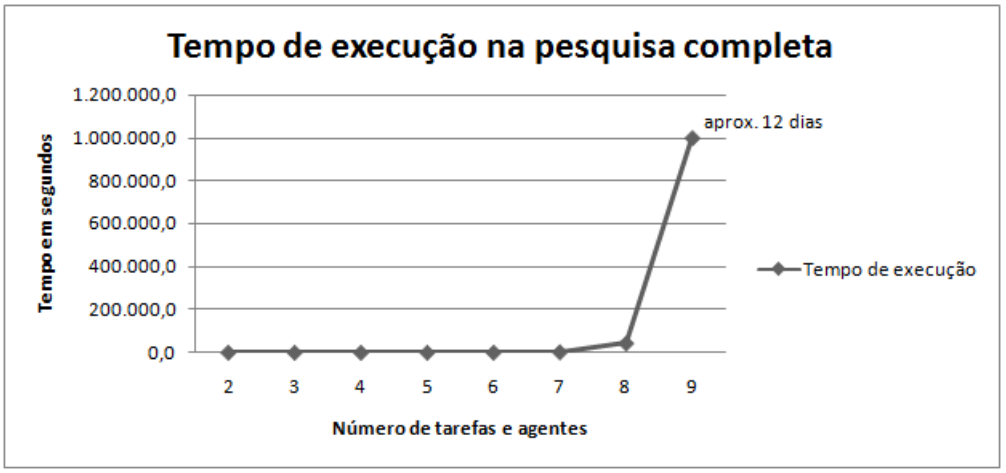

FIGURA 3.12: Gráfico do tempo de execução na pesquisa completa

3.4.4 Do GAP à privacidade

Com os testes efectuados podemos concluir que para este tipo de problema de atribuição de tarefas é difícil superar o desempenho a nível de consumos de memória e tempos de execução de um algoritmo centralizado, no entanto existem outros aspectos que podem ganhar importância na procura distribuída com agentes.

A privacidade dos dados é um aspecto que não se consegue com algoritmos centralizados que para serem executados têm que ter toda a informação reunida. No caso da pesquisa com agentes não é necessária passar toda a informação ao algoritmo para que este seja aplicado, logo não pode ser feita uma análise que tenha em conta somente o tempo de execução e consumos de memória, quando a privacidade e a característica distribuída de um problema poderá ter mais interesse.

A próxima secção vai ser dedicada à exploração do aspecto da privacidade. É utilizado um problema de escalonamento de eventos, suportado pela formulação DiMES, para salientar a importância da privacidade neste contexto.

3.5 DCOP no suporte à privacidade (DiMES)

3.5.1 Formulação DiMES

Apesar de ser promissor, dois desafios devem ser colocados ao DCOP para avançar como uma abordagem viável aos problemas do mundo real.

Primeiro, enquanto os investigadores formularam problemas específicos para o DCOP, ainda não tinha sido desenvolvido um método reutilizável que sistematicamente construisse formulações (Maheswaran et al., 2004). Desta forma é facilitado o processo de para cada domínio de problema ter de definir a modelação DCOP, com variáveis, restrições e domínios a aplicar ás variáveis.

Com o auxílio de uma formulação de um nível mais alto, a preocupação vai para a formulação do problema em si, deixando a formulação DCOP ser mapeada a partir da formulação de mais alto nível.

O segundo desafio é perceber se alguns problemas reais cuja complexidade associada é grande, conseguem ser resolvidos pelos algoritmos utilizados, ou mesmo se os algoritmos conseguem ter desempenho suficiente.

Focando-se no primeiro desafio, o DiMES (*Distributed Multi-Event Scheduling*) surge como uma formulação vocacionada para um classe de problemas como o agendamento de múltiplos eventos em que recursos partilhados participam em actividades conjuntas (Maheswaran et al., 2004). O DiMES proporciona uma maior abstracção na formulação DCOP e permite que o foco esteja direccionado para o problema em si. Deste modo formula-se o problema a partir de nível mais alto e o mapeamento para o DCOP é feito de forma sistemática.

Começamos por formalizar um problema definindo um conjunto $\mathcal{R} = \{R_1, ..., R_N\}$ de cardinalidade N onde R_n refere o recurso n, e um conjunto $\mathcal{E} = \{E^1, ..., E^K\}$ de cardinalidade K onde E^k se refere ao evento k.

Após definir os conjuntos de recursos e eventos, é necessário formalizar o intervalo de tempo em que estes podem ser alocados. De modo mais simples $[T_{earliest}, T_{latest}]$ define o intervalo de tempo disponível para as atribuições. Como $T \in \mathbb{N}$ é um número natural e Δ especifica um tamanho, $T.\Delta = T_{latest} - T_{earliest}$. Desta forma podemos caracterizar o domínio do tempo pelo conjunto $\mathcal{T} = \{1, ..., T\}$ de cardinalidade T onde um elemento $t \in \mathcal{T}$ se refere ao intervalo de tempo dado por $[T_{earliest} + (t-1)\Delta, T_{earliest} + t\Delta]$. Como exemplo, definindo um intervalo de tempo das 10 horas até às 20 horas repartido em partes de 30 minutos, em DiMES é representado por $\mathcal{T} = \{1, ..., 20\}$ onde o *slot* quatro é mapeado no intervalo [11h, 11h30m].

Um evento k é caracterizado por um tuplo $E^k := (A^k, L^k; V^k)$ onde $A^k \subset \mathcal{R}$ é um subconjunto dos recursos necessários pelo evento. $L^k \in \mathcal{T}$ é o número de intervalos de tempo contínuos exigidos para o evento, onde são necessários os recursos A^k. A importância dada ao evento por cada um dos recursos de A^k é especificada no vector de importância V^k, que tem a cardinalidade do conjunto A^k. Para $R_n \in A^k$, o elemento V^k_n do vector V^k, indica o valor de importância por intervalo de tempo que o recurso n atribui ao

agendamento do evento k. Cada recurso além de definir a importância de participar em determinado evento, tem também um vector que caracteriza a preferência do recurso em deixar os intervalos de tempo livres. Formalmente, $V_n^0(t) : \mathcal{T} \rightarrow \mathbb{N}^+$ define a importância atribuida a cada slot t para o deixar livre. Estes valores permitem que exista comparação da importância relativa de eventos para eventos e também do valor do tempo para os recursos. Seja $\overline{V}_n := \max_{k \in \{1,...,K\}} V_n^k$ o valor máximo dado pelo recurso n para agendar qualquer evento. Um recurso pode excluir um intervalo de tempo de ser atribuído se especificar um valor suficientemente alto, i.e. $V_n^0(t) > \overline{V}_n$.

Problema de agendamento. Começando por definir um determinado escalonamento S que é visto como uma função $S(E^k) \subset \mathcal{T}$ que determina os intervalos de tempo agendados para o evento E^k. Como já foi referido, assume-se que o agendamento dos intervalos de tempo são contínuos, logo, para que o evento E^k seja considerado agendado é preciso que todos os recursos envolvidos (A^k) estejam de acordo em agendar os intervalos de tempo dados por $S(E^k)$ ao evento E^k. Um conflito ocorre se dois eventos com recursos comuns são agendados para intervalos de tempo sobrepostos: $S(E^{k_1}) \cap S(E^{k_1}) \neq 0$, para qualquer $k_1, k_2 \in \{1, ..., K\}$, $k_1 \neq k_2$, $A^{k_1} \cap A^{k_2} \neq 0$. Um agendamento $S(E^k) = 0$ indica que o evento E^k não foi agendado, e pode ocorrer por divergência dos recursos na definição dos intervalos de tempo, por divergência no agendamento de eventos com níveis diferentes de importância, ou mesmo por ser mais importante para os recursos ter os intervalos de tempo livres.

3.5.2 Problema da agregação

Aproximando o problema de agendamento a um cenário real, definimos um problema em que uma organização quer maximizar a soma das utilidades dos seus recursos de modo a ter a melhor rentabilização possível. Deste modo, a métrica para definir a utilidade de um recurso é a diferença entre a soma dos valores de importância dados aos eventos, e o agregado dos valores dados para ter os intervalos de tempo livres. Formalmente têm-se:

$$\max_S \left\{ \sum_{k=1}^{K} \sum_{n \in A^k} \sum_{t \in S(E^k)} (V_n^k - V_n^0(t)) \right\} \tag{3.14}$$

tal que $S(E^{k_1}) \cap S(E^{k_1}) \neq 0$, $\forall k_1, k_2 \in \{1, ..., K\}$, $k_1 \neq k_2$, $A^{k_1} \cap A^{k_2} \neq 0$.

Como anteriormente foi dito, o objectivo do DiMES é proporcionar um método que facilite o mapeamento de uma gama de problemas para DCOP e aproveitar os algoritmos já desenvolvidos para o DCOP para resolver estes problemas. Existem vários modos de

efectuar o mapeamento para variáveis em DCOP: considerando os intervalos de tempo como variáveis (TSAV), os eventos como variáveis (EAV), ou os eventos privados como variáveis (PEAV). Estes vários mapeamentos são identificados em (Maheswaran et al., 2004, secção 3). Neste trabalho, motivado pelo uso da privacidade e pelo *feedback* recolhido junto do próprio autor, será utilizado o mapeamento PEAV.

Mapeamento de eventos privados como variáveis (PEAV). Este tipo de formulação é em tudo semelhante à EAV que considera os eventos como as variáveis de decisão, mas surgiu para introduzir o conceito de privacidade não existente na EAV (cf. (Maheswaran et al., 2004, secção 3)). Há informações sensíveis nos agentes que não devem ser públicas, e são esses interesses que a formulação PEAV pretende proteger. Os agentes como parte de uma equipa têm informações comuns e públicas, no entanto têm outras que são sensíveis e da responsabilidade de cada um.

No mapeamento PEAV começamos por definir um conjunto de variáveis, $X^k = \{x_n^k : n \in A^k\}$ onde $x_n^k \in \{0, 1, ..., T - L^k + 1\}$ define o tempo inicial para o evento E^k no agendamento do recurso R_n necessário para o evento. Se $x_n^k = 0$, então R_n escolhe não agendar o evento E^k. Assim, pode ser construído um conjunto de variáveis DCOP, $X = \cup_{k=1}^{K} X^k$. Para agrupar as variáveis por agente é definido o conjunto $\tilde{X}_n = \{x_m^k \in X : m = n\} \subset X$, que representa as variáveis pertencentes ao agente n. Claramente, $|\tilde{X}_n| > 0 \ \forall n$, senão o recurso não é necessário em nenhum evento. Caso $|\tilde{X}_n| = 1$, então $X_n = \tilde{X}_n \cup \{x_n^0\}$ onde $x_i^0 \equiv 0$ representa uma variável *dummy*. Esta variável *dummy* é criada para que existam funções intra-agentes válidas, porque na modelação PEAV todas as avaliações estão nas ligações intra-agentes (assegurando privacidade), e tem que existir sempre pelo menos uma destas ligações.

Passando para o domínio DCOP, as variáveis em X_n são agrupadas num agente que representa os interesses do recurso n. As variáveis internas dos agentes são todas interligadas e as ligações externas só existem entre variáveis dos recursos que participam no mesmo evento. A figura 3.13 mostra um exemplo de um problema DiMES formulado como PEAV que dá origem ao grafo de restrições DCOP representado na figura 3.14.

Dado um conjunto de variáveis, o objectivo é construir as funções de utilidade das restrições de modo a que o problema DCOP resultante seja resolvido e obtenha a solução correspondente ao problema DiMES (Maheswaran et al., 2004). Em seguida são apresentadas as proposições para a formulação PEAV:

Proposição 3.1. *A função de utilidade para DCOP, tendo em conta o mapeamento PEAV, onde as restrições entre as variáveis* $x_{n_1}^{k_1}$ *e* $x_{n_2}^{k_2}$ *quando* $x_{n_1}^{k_1} = t_1$ *e* $x_{n_2}^{k_2} = t_2$ *é dada por:*

$$f(n_1,k_1;n_2,k_2,t_2) = -MI_{\{n_1\neq n_2\}}I_{\{k_1=k_2\}}I_{\{t_1\neq t_2\}}+I_{\{n_1=n_2\}}I_{\{k_1\neq k_2\}}f_{intra}(n_1;k_1,t_1;k_2,t_2)$$

onde $f_{intra}(n;k_1,t1;k_2,t2) =$

$$\begin{cases} -M & t_1 \neq 0, t_2 \neq 0, t_1 \leq t_2 \leq t_1 + L^{k_1} - 1, \\ -M & t_1 \neq 0, t_2 \neq 0, t_2 \leq t_1 \leq t_2 + L^{k_2} - 1, \\ g(n;k_1,t1;k_2,t_2) & \text{caso contrário.} \end{cases}$$

e

$$g(n;k_1,t_1;k_2,t_2) = \frac{1}{|X_n|-1}\left(Z_n^{k_1}(t_1) + Z_n^{k_2}(t_2)\right)$$

onde

$$Z_n^{k_i} = \sum_{l=1}^{L^{k_i}} \left(V_n^{k_i} - V_n^0(t_i+l-1)\right) I_{\{t_i\neq 0\}}$$

Para que o problema DiMES produza a solução óptima $M > NTV_{max}$ onde N é o número de participantes, T é o número de intervalos de tempo e $V_{max} = \max_{k,n} V_n^k$. A prova desta formulação pode ser encontrada em (Maheswaran et al., 2004, secção 3). Neste trabalho o aspecto de maior relevância reside no modo de cálculo das funções de utilidade que mapeadas para DCOP vão representar as funções de custo das diversas restrições.

Na subsecção seguinte irá ser apresentado um exemplo de utilização da formulação DiMES para definir um problema de agendamento de eventos.

3.5.3 Exemplo DiMES

De modo a exemplificar a utilização da formulação DiMES e o respectivo mapeamento para DCOP, vai ser definido um problema em que três recursos podem ser atribuídos a dois eventos. Os recursos são identificados pelo conjunto $\mathcal{R} = \{A, B, C\}$ de cardinalidade $N = 3$. Os dois eventos são representados pelo conjunto $\mathcal{E} = \{E^1, E^2\}$ de cardinalidade $K = 2$. Estes dois eventos têm três intervalos de tempo possíveis para serem alocados, identificados por $\mathcal{T} = \{1, 2, 3\}$ de cardinalidade $T = 3$ onde os intervalos são de uma hora, correspondendo aos intervalos [10h, 11h, 12h].

Relembrando da especificação DiMES, os eventos são definidos por um vector $E^k := (A^k, L^k, V^k)$, onde, A^k define os recursos necessários no evento, L^k representa o número

de intervalos contínuos do evento, e V^k contém os valores de importância do evento para os recursos de A^k.

Para este exemplo definimos dois eventos de uma hora cada um. O evento E^1 necessita dos recursos A e B que atribuem os valores de importância de 2 e 3 para a sua participação no evento. O evento E^2 necessita dos recursos B e C que atribuem os valores de importância 1 e 4. Definindo formalmente os eventos, temos para o evento E^1 e E^2:

$$E^1 := \{\{A, B\}, 1, \{2, 3\}\}, E^2 := \{\{B, C\}, 1, \{1, 4\}\}$$

A figura 3.13 mostra este exemplo DiMES mapeado como PEAV, que é convertido no problema DCOP ilustrado na figura 3.14

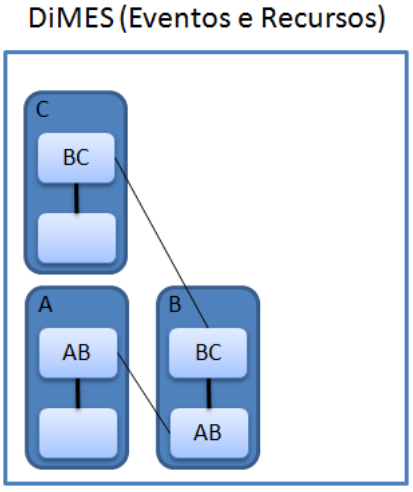

FIGURA 3.13: Problema DiMES formulado como PEAV

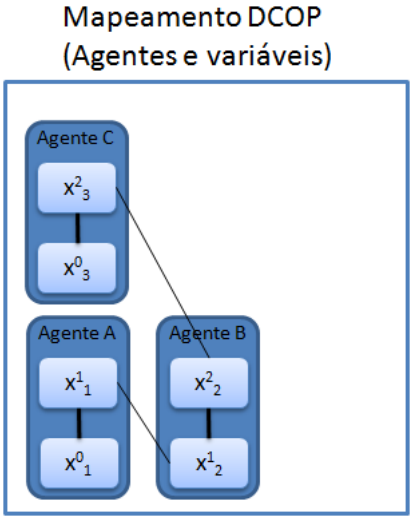

FIGURA 3.14: Mapeamento do problema DiMES para DCOP

Ainda considerando este exemplo, para que possa ser resolvido é necessário que cada agente defina a sua parte privada, especificando para cada intervalo de tempo, o valor de importância de ter esse intervalo livre. Neste exemplo são definidos os vectores: $V_0^1 = \{2, 1, 2\}$, $V_0^2 = \{1, 2, 3\}$, $V_0^3 = \{1, 1, 1\}$.

Ao resolver este problema, verifica-se que a melhor atribuição é a representada na figura 3.15, em que o evento E^1 é agendado para o intervalo das 10 horas e o evento E^2 é agendado para as onze horas, ficando o último intervalo livre.

Intervalo 1 (10h)	Intervalo 2 (11h)	Intervalo 3 (12h)
BC	AB	----

FIGURA 3.15: Atribuição resultante da resolução do problema DiMES

A título exemplificativo, a figura 3.16 demonstra as alterações nas restrições DiMES ao ser adicionado um novo recurso D e um novo evento em que participam os recursos A, C e D.

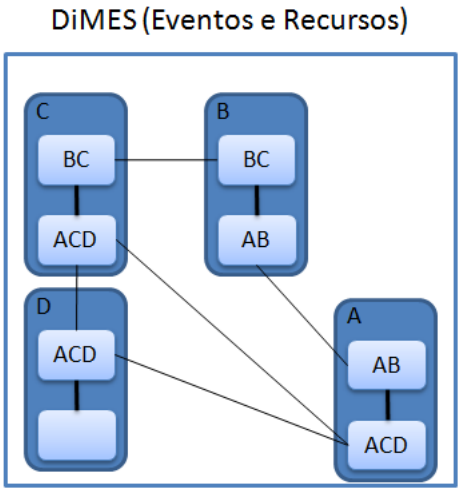

FIGURA 3.16: Resultado da formulação DiMES ao adicionar um novo evento e recurso

Capítulo 4

Modelo de integração DiMES-GAP

Este capítulo apresenta o modelo de integração das formulações DiMES e GAP (DiMES-GAP) numa só bancada de resolução DCOP. Este modelo permite definir problemas utilizando a formulação GAP que não contempla a noção de privacidade, ou utilizando a formulação DiMES com suporte à privacidade. Com este modelo temos um modo automático e sistemático para explorar e resolver este tipo de problemas.

A figura 4.1 ilustra o modelo apresentado, que é constituido por quatro componentes:

- geração automática DCOP a partir de GAP
- geração automática DCOP a partir de DiMES
- ajuste da implementação ADOPT original
- aproximação das formulações ao utilizador

Estes componentes vão ser explicados de forma mais detalhada nas secções seguintes.

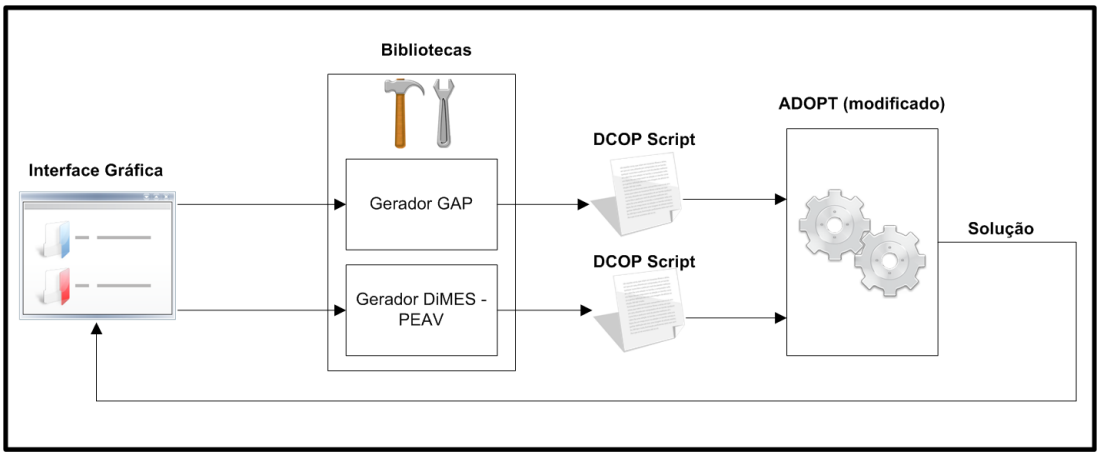

FIGURA 4.1: Modelo de integração DiMES-GAP

4.1 Guião de formulação DCOP

Antes de apresentar outros aspectos do modelo, importa introduzir o formato do guião (*script*) usado para formular o problema DCOP a dar como entrada ao algoritmo ADOPT. Este *script* contém uma lista de agentes, uma lista de variáveis e uma lista de restrições, onde para cada restrição existe a definição dos pares de valores não permitidos (*NOGOODS*). As definições são feitas do seguinte modo:

- para definir um agente é utilizado: AGENT <identificador do agente>
- para definir uma variável associada a um agente utiliza-se:
  VARIABLE <identificador da variável> <identificador do agente associado> <dimensão do domínio>
- para definir uma restrição e os pares de valores não permitidos:
  CONSTRAINT <identificador da variável> <custo>
  NOGOOD <valor><valor>

Em seguida é apresentado um exemplo simples de um *script* de configuração de um problema DCOP com três agentes e três variáveis associadas a cada um deles, e duas restrições que definem que os valores não podem ser iguais entre os agentes implicados na restrição.

```
AGENT 1
AGENT 2
```

```
AGENT 3
VARIABLE 0 1 3
VARIABLE 1 2 3
VARIABLE 2 3 3
CONSTRAINT 0 2
NOGOOD 0 0
NOGOOD 1 1
NOGOOD 2 2
CONSTRAINT 1 2
NOGOOD 0 0
NOGOOD 1 1
NOGOOD 2 2
```

Para a formulação dos problemas tratados neste trabalho, a definição original do *script* não é suficiente, pois só permite definir para uma restrição, pares de valores não permitidos. Assim, foi necessário adicionar um novo comando para atribuir um custo a um par de valores de uma restrição. Este aspecto é referido na secção de ajuste da implementação original ADOPT, no entanto a nível de *script* foi adicionada uma nova definição para suporte de custos para os pares de valores das restrições:

```
FCCOST <valor> <valor> <custo>
```

4.2 Geração automática DCOP a partir de GAP

A geração automática do *script* DCOP a partir de GAP segue os passos apresentados na secção 3.3.1. A biblioteca desenvolvida aceita como entrada:

- o número de agentes e tarefas do problema
- a matriz de capacidades globais dos agentes
- a matriz de capacidades dos agentes por tarefa
- a matriz de consumos dos agentes por tarefa

e automatiza os seguintes aspectos:

- geração da matriz binária com a combinação de todas tarefas para formar o domínio global

- geração da tabela de recursos consumidos por agente para o domínio global, de modo a utiliza-la para definir o domínio local de cada agente
- iniciação do ficheiro de *script*
- declaração dos agentes e variáveis geradas no ficheiro de *script*
- calculo da tabela de custo para todas as restrições
- declaração das funções de custo das restrições no ficheiro de *script*

Para se perceber melhor os passos efectuados na geração automática do problema DCOP é apresentado no algoritmo 5, o pseudo-código desenvolvido:

Algoritmo 5 Pseudo-código da geração automática de DCOP a partir de GAP

```
1: procedure GAP-DCOP(Agentes, matrizCap, matrizCapTarefa, matrizConTarefa)
2:     GerarMatrizDominioGlobal()
3:     Gerar tabelas do domínio de cada agente a partir do domínio global
4:     Criar ficheiro de script
5:     for all a ∈ Agentes do
6:         Adicionar a ao ficheiro de script
7:     end for
8:     for all Restrições do
9:         Inserir no ficheiro de script nova restrição
10:        CalcularTabelaAuxiliarDeUtilidade()
11:        CalcularTabelaDeCusto()
12:        Inserir no ficheiro de script os valores de custo da restrição
13:    end for
14: end procedure
```

Após terminada a execução dos passos apresentados em cima, é gerado o ficheiro com o *script* DCOP, que posteriormente vai ser passado ao algoritmo ADOPT.

4.3 Geração automática DCOP a partir de DiMES

Para efectuar a geração de DCOP a partir de DiMES foi seguida a formulação PEAV. Para se perceber os pontos essenciais na geração automática do problema DCOP, em seguida é apresentada a sequência necessária de passos a efectuar. A biblioteca têm como entrada:

- o conjunto de recursos DiMES
- o conjunto de eventos DiMES

- o número de intervalos de tempo
- os valores privados da importância atribuída pelos recursos aos intervalos de tempo livres

e automatiza os processos de:

- calculo do V_{max}(máximo valor atribuído aos eventos pelos recursos)
- geração do conjunto de variáveis DCOP
- agrupamento das variáveis DCOP por recursos
- agrupamento das variáveis DCOP por eventos para gerar restrições inter-agente
- adição de variáveis *Dummy* aos agentes que precisem
- ordenação das variáveis DCOP para escrita sequencial no ficheiro
- geração das funções de utilidade para a formulação PEAV
- geração das funções de custo a partir das funções de utilidade DiMES

Passando todos estes passos que acabaram de ser identificados, é gerado o ficheiro com a descrição do problema DCOP. O algoritmo 6 apresenta o pseudo-código da geração automática de DCOP a partir de DiMES.

Algoritmo 6 Pseudo-código da geração automática de DCOP a partir de DiMES

```
1: procedure DiMES-DCOP(recursos, eventos, nIntervalos)
2:     CalcularVmax()
3:     Criar ficheiro de script
4:     DCOPvars ← {}
5:     for all ev ∈ eventos do
6:         for all rs ∈ ev.A^k do
7:             dominio ← 1 + nIntervalos − ev.L^k + 1
8:             DCOPvars ← DCOPvars ⊕ criarDCOPvar(rs, ev, dominio)
9:         end for
10:    end for
11:    DCOPres ← AgruparVariaveisDCOPporRecurso(DCOPvars)
12:    DCOPevt ← AgruparVariaveisDCOPporEvento(DCOPvars)
13:    for all dcopVarRes ∈ DCOPres do
14:        if |dcopVarRes| = 1 then
15:            dcopVarRes ← dcopVarRes ⊕ criarDummyVar()
16:        end if
17:    end for
18:    OrdenarDCOPvars(DCOPres)
19:    funUt ← GerarFuncoesDeUtilidade(DCOPres)
20:    MinimizarFuncoesDeUtilidade(funUt)
21:    ConstruirScriptDCOP(funUt)
22: end procedure
```

4.4 Implementação ADOPT

Nesta secção são caracterizados os aspectos da implementação original do algoritmo ADOPT, e são apresentados os ajustes que lhe foram efectuados.

4.4.1 Original

A implementação do algoritmo ADOPT utilizada neste modelo, baseou-se na proposta por Modi (Modi et al., 2006), sendo efectuadas alterações à disponibilizada pelo autor. A implementação original não está completamente estável e não suporta a definição das funções de custo das restrições DCOP, permitindo somente definir para as restrições, pares de valores admissíveis e não admissíveis. As funções de custo nesta implementação são definidas por:

$$f_{ij} : D_i \times D_j \rightarrow \{0, 1\} \tag{4.1}$$

devolvendo para cada par de valores:

- zero, indicando valor admissível
- um, indicando valor não admissível

Esta implementação base recebe como entrada um *script* (cf. secção 4.1) com a formulação do problema, e devolve a solução encontrada. As funções de custo são definidas no *script* através de comandos "NOGOOD" que indicam que um par de valores é "não admissível". Para definir pares de valores admissíveis, é simplesmente não os inserir no *script*. Em seguida é apresentado um exemplo de um *script* com a definição dos pares <0,0> e <1,1> como valores não admissíveis.

```
...
CONSTRAINT 0 2
NOGOOD 0 0
NOGOOD 1 1
...
```

4.4.2 Ajuste da implementação original

Para poder utilizar a implementação deste algoritmo na resolução dos problemas apresentados neste trabalho, foram efectuadas quatro alterações à implementação original:

- adicionada a possibilidade de definir custos para um par de valores de uma restrição
- modificado o analisador de *script* para suportar definição de custos para as restrições
- modificado o modo de avaliação das restrições
- adicionada a opção de ser recebida uma notificação com os resultados da solução encontrada

Ao adicionar a possibilidade de definir custos para um par de valores de uma restrição, passamos a cumprir a especificação das funções de custo definidas no DCOP. Com a alteração efectuada, passamos a definir as funções de custo do seguinte modo:

$$f_{ij} : D_i \times D_j \rightarrow [0, +\infty[\tag{4.2}$$

Esta modificação implicou uma alteração no analisador de *script*, adicionando um novo comando para definir os valores de custo. De maneira semelhante ao comando "NO-GOOD", adicionou-se o novo comando:

```
FCCOST <valor> <valor> <custo>
```

Em seguida é apresentado um exemplo em que é atribuído o custo 3 a um par de valores associado a uma restrição.

```
...
CONSTRAINT 0 2
FCCOST 0 0 3
...
```

Para que a solução dada pelo algoritmo seja a correcta foi necessário também modificar a função de avaliação das restrições, que anteriormente só verificava se existiam pares de valores não admissíveis para as restrição, e passou a ter em conta os vários custos. Ao nível da implementação, foi modificado o modo de cálculo do custo local, onde são avaliadas as funções de custo.

$$\delta(d_i) = \sum_{(x_j,d_j) \in CurrentContext} f_{ij}(d_i, d_j) \tag{4.3}$$

De modo a interagir com o algoritmo, existiu a necessidade de adicionar uma funcionalidade para permitir que o algoritmo pudesse ser chamado para resolver um problema, e tivesse a capacidade de enviar esse resultado após encontrar a solução, para a entidade que lhe fez o pedido. Este aspecto aplica-se na interacção com a interface gráfica, que é notificada com a solução a apresentar.

4.5 Aproximação das formulações ao utilizador

De forma a tornar mais fácil, automática e sistemática a tarefa de formulação dos problemas DiMES e GAP, foi desenvolvida uma interface gráfica. Esta interface gráfica é composta por duas partes, uma para definir problemas GAP sem atender ao aspecto

da privacidade, e outra para formular problemas DiMES tendo em conta o aspecto da privacidade. Esta interface gráfica tem também o objectivo de apresentar a solução, devolvida pelo algoritmo ADOPT, de uma forma tratada e fácil de perceber.

O desenvolvimento desta interface gráfica implicou algumas alterações na biblioteca ADOPT, de forma a existir uma comunicação assíncrona da solução encontrada.

Após ser formulado o problema (DiMES ou GAP), a interface gráfica pede à biblioteca (Gerador GAP ou Gerador DiMES) para gerar o respectivo *script* DCOP. Após ser produzido o *script* DCOP, é executado o algoritmo ADOPT num novo fio de execução, que ao terminar, comunica a solução encontrada à interface gráfica.

O aspecto gráfico do protótipo é apresentado no apêndice A.

4.6 Exploração do modelo

Nesta secção são explorados os dois tipos de problemas apresentados previamente (GAP e DiMES) para serem resolvidos utilizando o modelo DiMES-GAP.

4.6.1 DiMES

Tendo como base o problema DiMES introduzido na secção 3.5.3, irão ser mostrados os passos para o resolver, utilizando o modelo DiMES-GAP, nomeadamente a sua interface gráfica.

A primeira etapa, ilustrada na figura 4.2, consiste em definir todos os recursos que intervêm no problema DiMES em causa, que neste exemplo vão ser $\mathcal{R} = \{A, B, C\}$.

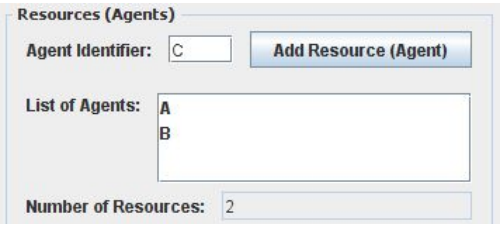

FIGURA 4.2: Definir recursos do problema DiMES

É também necessário configurar o número de intervalos de tempo disponíveis, bem como o tempo inicial de referência.

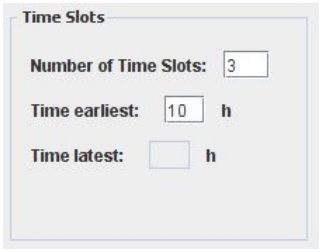

FIGURA 4.3: Definir os intervalos de tempo

Após serem declarados todos os recursos DiMES e definidos os intervalos de tempo, são criados os eventos, definindo a sua duração e associando-lhes os recursos. A criação de um evento é feita do seguinte modo (figura 4.4), define-se a identificação do evento e a sua duração em intervalos de tempo, e sem seguida adiciona-se cada um dos recursos pertencentes ao evento, especificando a importância do evento para esse recurso.

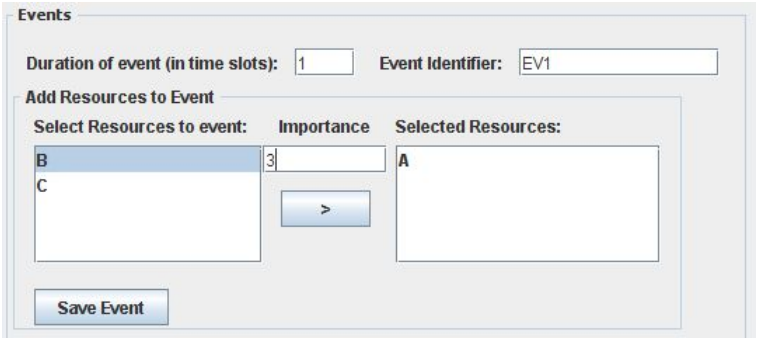

FIGURA 4.4: Definir eventos do problema DiMES

Ao ser definido o evento anterior, podemos observar na figura 4.5 a lista com os eventos, respectivos recursos associados e o seu valor de importância. Neste caso, só está listado o evento anterior, mas para resolver o exemplo do problema DiMES terá que ser adicionado do mesmo modo, um segundo evento.

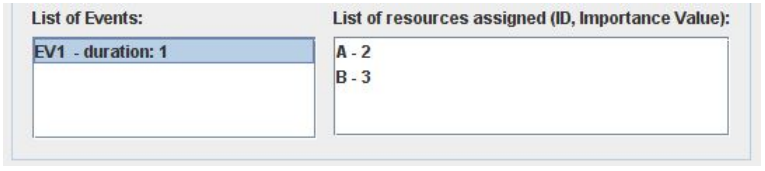

FIGURA 4.5: Lista dos eventos e recursos associados ao problema DiMES

Chegando a esta fase, todas as informações públicas do problema DiMES estão definidas, ficando somente a faltar as informações privadas dos recursos. Para o problema DiMES

poder ser resolvido, tem que ser definida a parte privada de cada recurso, ou seja, os vectores que indicam a importância dos intervalos de tempo ficarem livres para os diversos recursos. No modelo DiMES-GAP, de forma a exemplificar a diferença entre as informações públicas e privadas dos recursos, para definir este vector privado é necessário efectuar um login, passando o identificador do agente. Depois de identificado o agente, podem ser definidos os valores de importância dos intervalos de tempo ficarem livres. A definição das informações privadas do problema DiMES é ilustrada na figura 4.6 e 4.7.

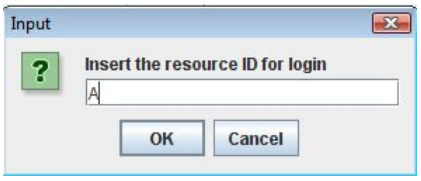

FIGURA 4.6: Login para informação privada

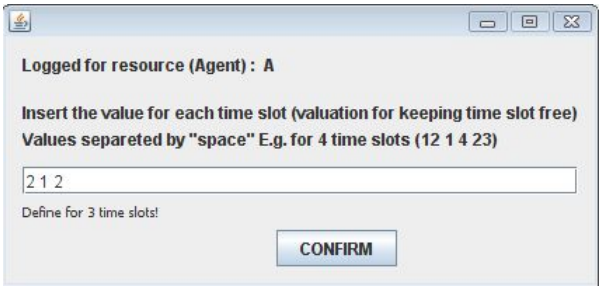

FIGURA 4.7: Definição da informação privada

Para estar concluida a exploração do problema DiMES utilizando o modelo DiMES-GAP, em seguida é mostrada uma figura com o resultado da resolução.

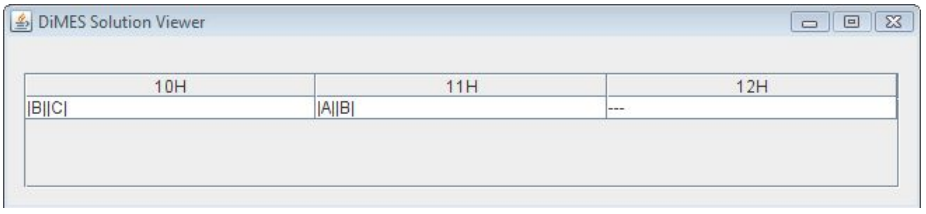

FIGURA 4.8: Solução encontrada pelo modelo DiMES-GAP

4.6.2 GAP

Nesta sub-secção é explorado o problema de atribuição de tarefas (GAP) anteriormente exemplificado na secção 3.3.2, através do modelo DiMES-GAP. Os passos a efectuar para

formular o problema GAP já foram apresentados, no entanto em seguida são contextualizados com a interface gráfica do modelo DiMES-GAP.

De modo a preparar o modelo para definir correctamente as matrizes de formulação do problema , é necessário introduzir o número de agentes e tarefas do problema GAP. Desta forma, o modelo DiMES-GAP apresenta as várias matrizes já conhecidas da formulação GAP, para que sejam inseridos os diversos valores de capacidades e consumos dos agentes que caracterizam o problema.

A matriz de capacidades globais dos agentes apresenta-se na figura 4.9.

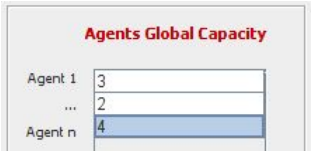

FIGURA 4.9: Definição da matriz global de capacidades dos agentes

As matrizes de capacidades e consumos dos agentes necessárias à formulação do problema são apresentadas nas figuras 4.10 e 4.11.

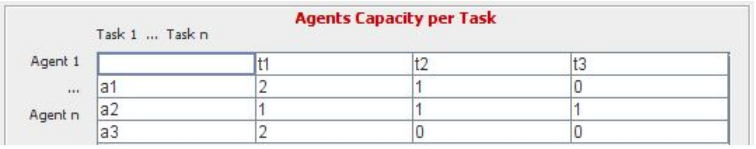

FIGURA 4.10: Definição da matriz de capacidades dos agentes por tarefa

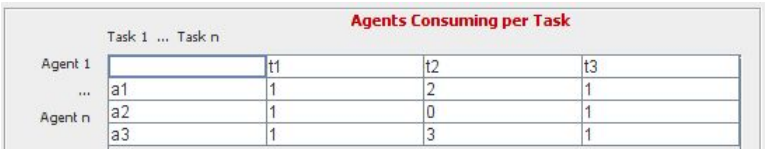

FIGURA 4.11: Definição da matriz de consumos dos agentes por tarefa

Para concluir a exploração do problema GAP utilizando o modelo, é apresentada na figura seguinte, a solução encontrada pelo modelo.

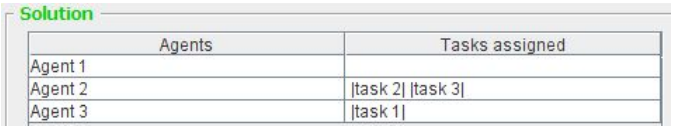

FIGURA 4.12: Solução encontrada para o problema GAP

Capítulo 5

Conclusões e trabalho futuro

Com base nos problemas de atribuição de tarefas e agendamento de eventos, motivados por cenários reais, compararam-se técnicas diferentes na sua resolução. A atribuição de tarefas foi ilustrada num cenário de busca e salvamento, e o agendamento de eventos foi ilustrado num cenário em que uma organização pretende maximizar o valor dos seus empregados, enquanto é mantida a privacidade da informação, como a importância relativa dos eventos ou do tempo do utilizador. Este trabalho propõe uma integração das formulações de forma a explorar e resolver os problemas de maneira distribuída e verificar em que contextos cada abordagem (centralizada e distribuída) faz mais sentido.

Numa análise feita à resolução destes problemas consideraram-se factores como o tempo de execução dos algoritmos e a memória necessária para a sua resolução. O comparativo leva-nos a concluir que para problemas que apresentem um grafo de restrições muito denso (e.g. problema de atribuição de tarefas), a resolução distribuída envolve a troca de um elevado número de mensagens, o que para problemas reais não é uma abordagem viável.

Ao analisar estes problemas, emergiu o conceito de privacidade que muitas vezes pode fazer a diferença na escolha da abordagem distribuída em detrimento da centralizada. A noção de privacidade geralmente não é um factor tido em conta nas análises feitas à resolução de problemas de modo distribuído *versus* centralizado, mas problemas como o do agendamento de eventos, fazem sobressair a necessidade de considerar este factor porque há informação sensível que não se quer disponibilizar. Para este tipo de problemas muitas vezes o desempenho ao nível do tempo de execução é um factor menos relevante quando comparado com a privacidade, e com a opção de não ter que disponibilizar toda a informação para que o problema seja solucionado.

Em alguns cenários não é mesmo possível a aplicação de algoritmos centralizados devido às particularidades dos problemas ou dos ambientes em que os problemas são resolvidos. Deste modo há problemas para os quais não é possível reunir a informação toda num só local ou não se quer disponibilizar toda essa informação, e ambientes em que existem falhas na comunicação e mesmo assim deseja-se que o algoritmo execute.

Existem no entanto alguns desafios que devem ser ultrapassados de maneira a que o DCOP seja uma aproximação viável na resolução de problemas reais, tais como a velocidade de resolução dos algoritmos utilizados e a existência de camadas de nível superior que permitam fazer mapeamentos dos problemas para DCOP, facilitando o processo de formulação e permitindo explorar os problemas de maneira mais sistemática e fácil.

Esta dissertação pretende dar contributo, sistematizando e automatizando a formulação dos problemas de atribuição de tarefas e agendamento de eventos, para resolução de modo distribuído e também avaliando os métodos de resolução dos problemas enunciados.

Trabalho futuro. Tendo como base este trabalho, existem na área da procura distribuída utilizando agentes, diversos aspectos por explorar.

A bancada DCOPolis (Sultanik et al., 2007), apesar de ainda estar em evolução, apresenta-se como uma bancada para simulação e desenvolvimento de algoritmos para resolução de problemas de optimização de restrições distribuídos. A DCOPolis foi desenvolvida para comparar e desenvolver problemas de optimização de restrições distribuídos num ambiente heterogéneo, onde os algoritmos e problemas estão contidos numa mesma plataforma, retirando os factores de desigualdade existentes quando estão envolvidos ambientes de desenvolvimento com diferentes condições.

O modelo apresentado neste trabalho pode também ser adaptado para ser utilizado com esta bancada, de maneira a adicionar mais formulações de problemas e de maneira a permitir a sua resolução com os algoritmos disponibilizados. Foram feitas algumas aproximações à bancada com um esforço conjunto com Evan A. Sultanik (Sultanik et al., 2007), autor da mesma, mas a integração deste modelo com esta *framework* só pode ainda ser considerado como trabalho futuro.

A bancada DCOPolis Utilizando o modelo de integração proposto nesta dissertação

A distribuição física é também um dos caminhos a seguir no futuro, podendo utilizar a plataforma de comunicação entre agentes JADE, que implementa as normas da FIPA (*Foundation for Intelligent Physical Agents*), para proporcionar uma camada de comunicação que permita a distribuição física dos agentes. Com uma distribuição física, a análise dos diversos algoritmos descentralizados pode ser mais rigorosa, e pode ser avaliado o

real desempenho, ao nível das comunicações, nomeadamente ao número de mensagens necessárias para que os algoritmos funcionem.

Apêndice A

Implementação em java do algoritmo BnB

```java
public interface Solver {
    Solution solve (Solution initial);
}
```

FIGURA A.1: Interface genérica *Solver*

```java
public abstract class AbstractSolver implements Solver {

    protected Solution bestSolution;
    protected int bestObjective;

    protected abstract void search (Solution initial);

    public Solution solve (Solution initial){
        bestSolution = null;
        bestObjective = Integer.MAX_VALUE;
        search (initial);
        return bestSolution;
    }

    public void updateBest (Solution solution){
        if (solution.isComplete () && solution.isFeasible ()
            && solution.getObjective () < bestObjective){
            bestSolution = solution;
            bestObjective = solution.getObjective ();
        }
    }
}
```

FIGURA A.2: Implementação genérica de *Solver*

```
public class DepthFirstBranchAndBoundSolver extends AbstractSolver {

    protected void search (Solution solution){
        if (solution.isComplete())
            updateBest (solution);
        else{
            Enumeration i = solution.getSuccessors();
            while(i.hasMoreElements()){
                Solution successor = (Solution) i.nextElement();
                if( successor.isFeasible() && successor.getBound() < bestObjective)
                    search(successor);
            }
        }
    }
}
```

FIGURA A.3: Implementação do algoritmo *Branch and Bound*

Apêndice B

Algoritmo ADOPT

```
Initialize
(1)   threshold ← 0; CurrentContext ← {}
(2)   forall d ∈ D_i, x_l ∈ Children do
(3)     lb(d, x_l) ← 0; t(d, x_l) ← 0
(4)     ub(d, x_l) ← Inf; context(d, x_l) ← {}; enddo
(5)   d_i ← d that minimizes LB(d)
(6)   backTrack

when received (THRESHOLD, t, context)
(7)   if context compatible with CurrentContext:
(8)     threshold ← t
(9)     maintainThresholdInvariant
(10)    backTrack; endif

when received (TERMINATE, context)
(11)  record TERMINATE received from parent
(12)  CurrentContext ← context
(13)  backTrack

when received (VALUE, (x_j,d_j))
(14)  if TERMINATE not received from parent:
(15)    add (x_j,d_j) to CurrentContext
(16)    forall d ∈ D_i, x_l ∈ Children do
(17)      if context(d, x_l) incompatible with CurrentContext:
(18)        lb(d, x_l) ← 0; t(d, x_l) ← 0
(19)        ub(d, x_l) ← Inf; context(d, x_l) ← {}; endif; enddo
(20)    maintainThresholdInvariant
(21)    backTrack; endif

when received (COST, x_k, context, lb, ub)
(22)  d ← value of x_i in context
(23)  remove (x_i,d) from context
(24)  if TERMINATE not received from parent:
(25)    forall (x_j,d_j) ∈ context and x_j is not my neighbor do
(26)      add (x_j,d_j) to CurrentContext;enddo
(27)    forall d' ∈ D_i, x_l ∈ Children do
(28)      if context(d', x_l) incompatible with CurrentContext:
(29)        lb(d', x_l) ← 0; t(d', x_l) ← 0
(30)        ub(d', x_l) ← Inf; context(d', x_l) ← {};endif;enddo;endif
(31)  if context compatible with CurrentContext:
(32)    lb(d, x_k) ← lb
(33)    ub(d, x_k) ← ub
(34)    context(d, x_k) ← context
(35)    maintainChildThresholdInvariant
(36)    maintainThresholdInvariant; endif
(37)  backTrack
```

FIGURA B.1: Corpo principal do algoritmo ADOPT (1) (Modi et al., 2006)

```
procedure backTrack
(38)  if threshold == UB:
(39)     d_i ← d that minimizes UB(d)
(40)  else if LB(d_i) > threshold:
(41)     d_i ← d that minimizes LB(d);endif
(42)  SEND (VALUE, (x_i, d_i))
(43)        to each lower priority neighbor
(44)  maintainAllocationInvariant
(45)  if threshold == UB:
(46)     if TERMINATE received from parent
(47)     or x_i is root:
(48)        SEND (TERMINATE,
(49)        CurrentContext ∪ {(x_i, d_i)})
(50)           to each child
(51)        Terminate execution; endif;endif
(52)  SEND (COST, x_i, CurrentContext, LB, UB)
         to parent
```

FIGURA B.2: Corpo principal do algoritmo ADOPT (2) (Modi et al., 2006)

```
procedure maintainThresholdInvariant
(53)  if threshold < LB
(54)     threshold ← LB; endif
(55)  if threshold > UB
(56)     threshold ← UB; endif

%note: procedure assumes ThresholdInvariant is satisfied
procedure maintainAllocationInvariant
(57)  while threshold > δ(d_i) + Σ_{x_l∈Children} t(d_i, x_l) do
(58)     choose x_l ∈ Children where ub(d_i, x_l) > t(d_i, x_l)
(59)     increment t(d_i, x_l); enddo
(60)  while threshold < δ(d_i) + Σ_{x_l∈Children} t(d_i, x_l) do
(61)     choose x_l ∈ Children where t(d_i, x_l) > lb(d_i, x_l)
(62)     decrement t(d_i, x_l); enddo
(63)  SEND (THRESHOLD, t(d_i, x_l), CurrentContext )
         to each child x_l

procedure maintainChildThresholdInvariant
(64)  forall d ∈ D_i, x_l ∈ Children do
(65)     while lb(d, x_l) > t(d, x_l) do
(66)        increment t(d, x_l); enddo;endo
(67)  forall d ∈ D_i, x_l ∈ Children do
(68)     while t(d, x_l) > ub(d, x_l) do
(69)        decrement t(d, x_l); enddo;enddo
```

FIGURA B.3: Procedimentos para actualizar os *thresholds* no ADOPT (Modi et al., 2006)

Apêndice C

Interfaces gráficas

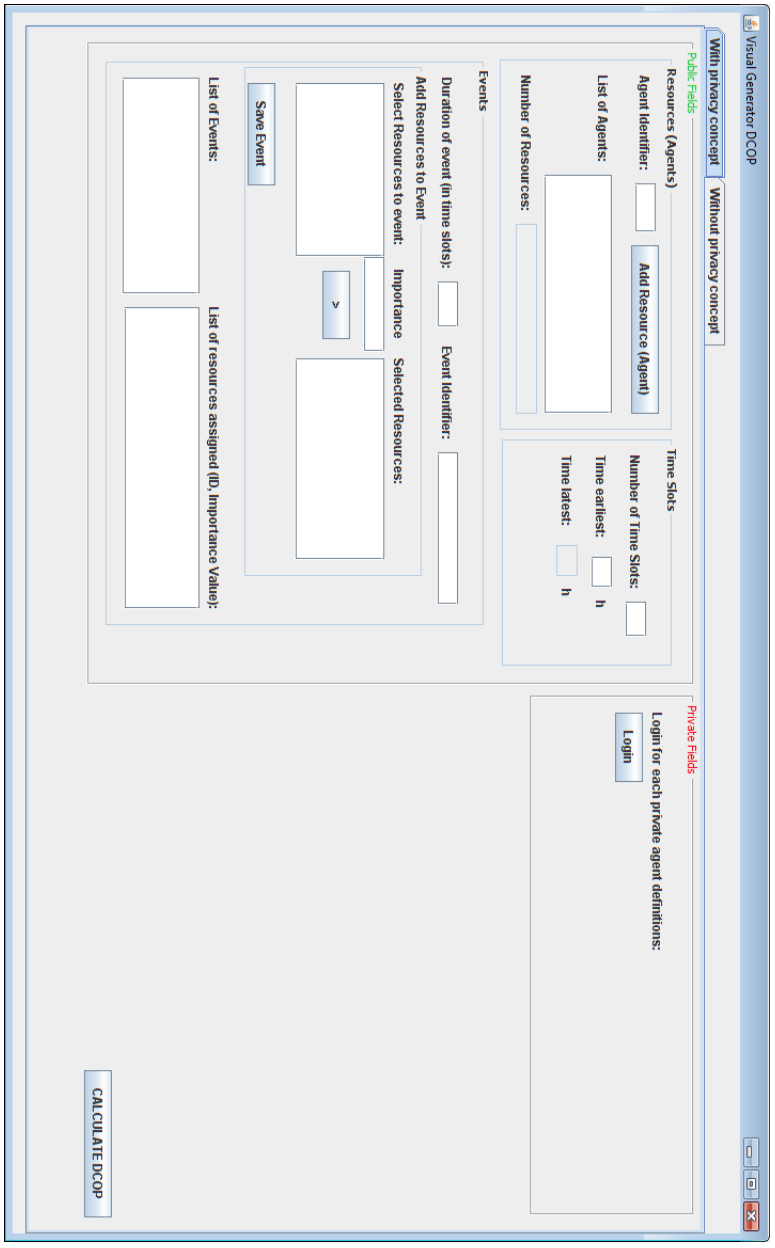

FIGURA C.1: Interface visual para formulação de problemas com DiMES

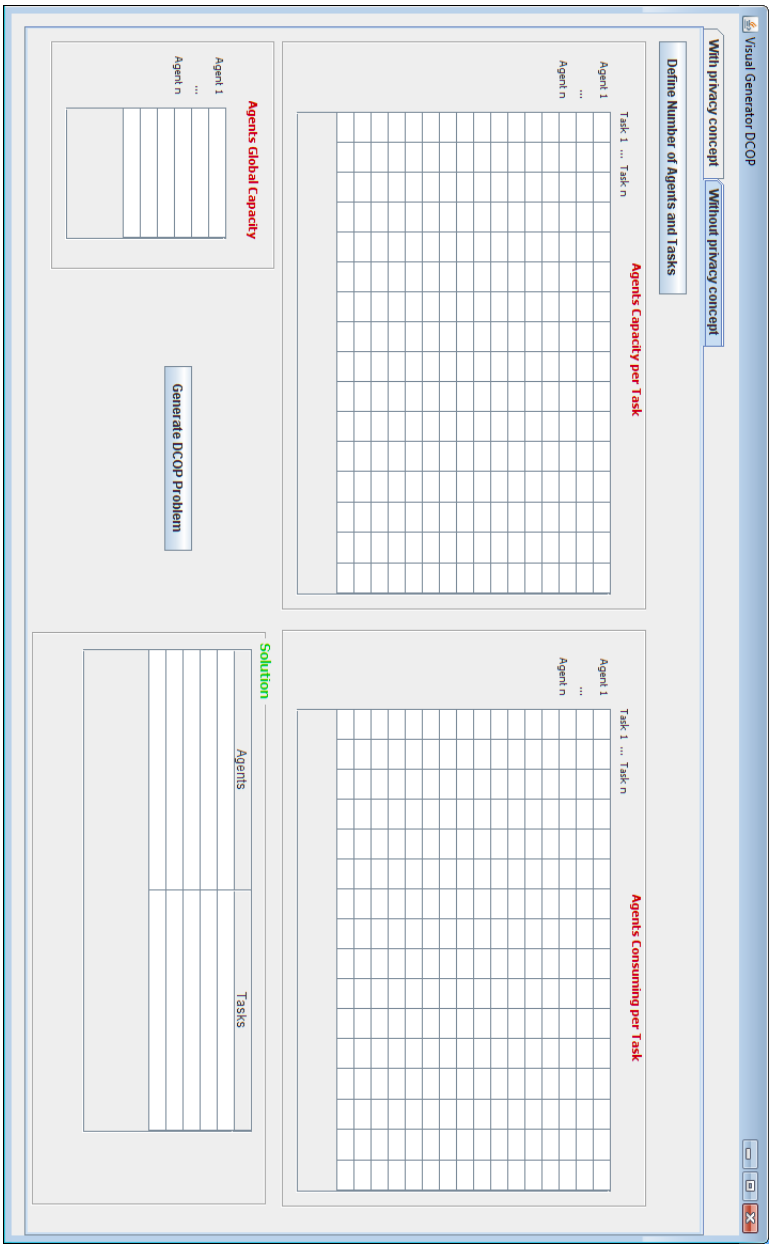

FIGURA C.2: Interface visual para formulação de problemas com GAP

Bibliografia

Ali, S., Koenig, S., and Tambe, M. (2005). Preprocessing techniques for accelerating the DCOP algorithm ADOPT. *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 1041–1048.

dos Santos, F. (2007). Estudo comparativo de métodos de optimização de restrições distribuidas. Pós-graduação, Universidade Federal do Rio Grande do Sul, Porto Alegre.

Greenstadt, R., Grosz, B., and Smith, M. (2007). SSDPOP: improving the privacy of DCOP with secret sharing. *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*.

Kitano, H., Tadokoro, S., Noda, I., Matsubara, H., Takahashi, T., Shinjou, A., and Shimada, S. (1999). Robocup rescue: search and rescue in large-scale disasters as a domain for autonomous agents research. *Systems, Man, and Cybernetics, 1999. IEEE SMC '99 Conference Proceedings. 1999 IEEE International Conference on*, 6:739–743 vol.6.

Kleinberg, J. and Tardos, E. (2005). *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Maheswaran, R., Tambe, M., Bowring, E., Pearce, J., and Varakantham, P. (2004). Taking DCOP to the Real World: Efficient Complete Solutions for Distributed Multi-Event Scheduling. *International Conference on Autonomous Agents: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-*, 1:310–317.

Meisels, A. (2008). *Distributed Search by Constrained Agents*. Springer.

Modi, P., Shen, W., Tambe, M., and Yokoo, M. (2003). An asynchronous complete method for distributed constraint optimization. *International Conference on Autonomous Agents: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, 14(18):161–168.

Modi, P., Shen, W., Tambe, M., and Yokoo, M. (2006). ADOPT: asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1-2):149–180.

Pereira, B. (2008). Agentes para procura distribuída de soluções com restrições locais e globais. In *Quartas Jornadas de Engenharia de Electrónica e Telecomunicações e de Computadores*.

Russell, S. and Norvig (2003). *Artificial intelligence: a modern approach*. Prentice Hall Englewood Cliffs, NJ.

Scerri, P., Farinelli, A., Okamoto, S., and Tambe, M. (2005). Allocating tasks in extreme teams. *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 727–734.

Shmoys, D. and Tardos, É. (1993). An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62(1):461–474.

Sultanik, E., Lass, R., and Regli, W. (2007). DCOPolis: A framework for simulating and deploying distributed constraint optimization algorithms. *The Ninth International Workshop on Distributed Constraint Reasoning Workshop, Providence, RI, September*.

Tsang, E. (1993). *Foundations of constraint satisfaction*. Academic Press San Diego.

Yeoh, W., Felner, A., and Koenig, S. (2008). BnB-ADOPT: an asynchronous branch-and-bound DCOP algorithm. *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 2*, pages 591–598.

Yokoo, M. (2001). *Distributed constraint satisfaction: foundations of cooperation in multi-agent systems*. Springer-Verlag, London, UK.