João Pedro Barreiros Nunes dos Santos

Tabulação com Operadores de Modo em Programas Lógicos

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Outubro de 2010

João Pedro Barreiros Nunes dos Santos

Tabulação com Operadores de Modo em Programas Lógicos

Dissertação submetida à Faculdade de Ciências da Universidade do Porto como parte dos requisitos para a obtenção do grau de Mestre em Engenharia de Redes e Sistemas Informáticos

Orientador: Ricardo Jorge Gomes Lopes da Rocha

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Outubro de 2010

Para a minha família.

Agradecimentos

Gostaria de agradecer, em primeiro lugar, ao meu orientador, o Professor Ricardo Rocha por todo o acompanhamento e apoio prestado durante a investigação e elaboração desta tese. As suas críticas e comentários construtivos permitiram que esta tese se tornasse realidade.

Também gostaria de agradecer aos meu colegas do projecto STAMPA: José Vieira, Miguel Areias, Flávio Cruz e João Raimundo pela ajuda, companheirismo e bom ambiente criado. Desejo-vos a todos boa sorte.

Agradeço ainda ao meu amigo e antigo colega João Bento pelo companheirismo e amizade demonstrados. A todos os meus colegas do DCC que me ajudaram durante o meu percurso académico.

A Catarina Félix e a Rosa Félix agradeço as opiniões pertinentes que fizeram a esta tese e todo o apoio que me deram.

Finalmente queria agradecer aos meus pais: Isolina Santos e Mário Santos, a oportunidade que me deram de tirar um curso superior e por estarem incondicionalmente ao meu lado.

Abstract

Logic programming languages, like Prolog, allow the programmer to concentrate on the problem's declaration instead of worrying about the details of its resolution. Despite Prolog's popularity, its resolution mechanism shows some limitations that can lead to situations in which queries to syntactically correct programs get into infinite loops. Tabling is an implementation technique that enables the solution of such problems in a very elegant way. This technique is based on saving and reusing the results of sub-computations during the execution of a program and, for that, the results and the tabled subgoals are stored in a proper data structure called the table space.

When we want to evaluate a predicate using tabling, we just need to declare it as *table p/n* at the top of the program, where p is the predicate name and n its arity. In this thesis, we present an alternative form of declaration of tabled predicates and, for that, we use statements like $p(a_1, ..., a_n)$, where p is the tabled predicate and the a_i's are mode operators. These operators allow the programmer to define different criteria for optimizing the answers that are inserted into the table space. The features added by these operators can then be elegantly used and applied to solve problems of Justification, Preferences and Answer Subsumption.

This thesis addresses the use of such operators in these three areas. We start by seeing how these operators can be used in the generation of justifications for the answers found during the execution of a program. Next, we discuss the set of transformations that allow preference problems to be implemented using mode operators and, then, we discuss the implementation of an answer subsumption mechanism using mode operators. We end this thesis by explaining the changes made to the YapTab engine, the Yap Prolog's tabling mechanism, in order to support the implementation of this new type of declaration.

Resumo

As linguagens lógicas, como o Prolog, permitem que o programador se concentre na declaração do problema em vez de se preocupar com os detalhes da sua resolução. Apesar da popularidade do Prolog, o seu mecanismo de resolução apresenta algumas limitações que podem levar a que programas sintacticamente correctos entrem em ciclo infinito. A tabulação é uma técnica de implementação que permite solucionar este tipo de problemas de uma forma bastante elegante. Esta técnica baseia-se em guardar e reutilizar os resultados das sub-computações durante a execução de um programa, onde os resultados encontrados bem como os subgolos tabelados são guardados numa estrutura de dados denominada tabela.

Quando queremos que um predicado seja avaliado recorrendo à tabulação basta declarar no topo do programa *table p/n* em que p é o predicado tabelado e n a sua aridade. Nesta tese apresentamos uma outra forma de declaração de predicados tabelados, passando a declaração a ser $p(a_1, ..., a_n)$, em que p é o predicado tabelado e os a_i são operadores de modo. Estes operadores permitem ao programador definir critérios de optimização das respostas que são inseridas na tabela. As funcionalidades adicionadas por estes operadores são ideais para serem utilizadas na resolução de problemas de Justificação, Preferências e *Answer Subsumption*.

Nesta tese abordamos a utilização dos operadores de modo nestas três áreas. Começamos por ver de que forma estes operadores podem ser utilizados na geração de justificações para as respostas encontradas durante a execução de um programa. Depois abordamos o conjunto de transformações que permitem que problemas de preferências possam ser implementados recorrendo a operadores de modo. Seguidamente discutimos a implementação de um mecanismo de *answer subsumption* recorrendo aos operadores de modo. Terminamos a tese explicando as modificações feitas ao YapTab, o mecanismo de tabulação do sistema Yap Prolog, de forma a poder suportar este novo tipo de declaração.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A Programação em Lógica é um paradigma que permite que o programador se concentre na descrição do problema em vez de se preocupar com os detalhes da sua resolução. Os programas lógicos são constituídos por um conjunto de cláusulas de Horn sendo por esse motivo bastante concisos e de fácil compreensão. A linguagem lógica mais popular é, sem dúvida, o Prolog. Grande parte do sucesso desta linguagem deve-se ao aparecimento, em 1983, de uma máquina abstracta capaz de correr código Prolog de forma eficiente [21]. Esta máquina tem o nome de Warren's Abstract Machine (WAM) [24] e foi desenhada por David H. H. Warren. Ainda hoje, a WAM permanece como a especificação base da maior parte dos interpretadores de Prolog como, por exemplo, o sistema Yap Prolog.

O Prolog usa a resolução *Selective Linear Definite* (SLD) [14] para chegar ao conjunto de respostas que satisfazem as cláusulas de Horn definidas pelo programador. Contudo, este tipo de estratégia têm-se demonstrado pouco adequada em certas ocasiões, porque programas sintacticamente correctos podem entrar em ciclo infinito dependendo da ordem pela qual ocorrem as cláusulas e os subgolos. Para ultrapassar estas dificuldades surgiu uma técnica chamada *tabulação* [15]. Esta técnica baseia-se em ir guardando em tempo de execução as respostas e os subgolos que lhe deram origem para mais tarde serem reutilizados em lugar de executar as cláusulas do programa. A *tabulação* apresenta, por isso, como outra vantagem a eficiência uma vez que evita a repetição de sub-computações. Se desejarmos que um sub-golo seja avaliado recorrendo à tabulação basta incluir no inicio do programa a declaração *table p/n* em que p é o nome do predicado tabelado e n a sua aridade.

As respostas e os subgolos são guardados numa espaço próprio denominado por *tabela*.

No YapTab [20], o mecanismo de tabulação do Yap Prolog, a tabela está implementada recorrendo a uma estrutura de dados chamada *Tries* [4]. Esta estrutura foi escolhida por permitir uma melhor utilização da memória e por garantir uma inserção e acesso a dados mais eficiente [17].

A inserção de novas respostas na tabela só é feita se não existir já uma resposta variante, isto é, uma resposta igual a menos da renomeação das variáveis. Por vezes, mesmo este tipo de controle não é suficiente para eliminar completamente a ocorrência de ciclos infinitos. Guo et al. propõem uma nova forma de declarar predicados tabelados que nos permite definir sobre que argumentos da resposta a verificação de variância deve ser feita [7]. Passamos a declarar os predicados tabelados da seguinte forma *table* $p(a_1, ..., a_n)$ em que os a_i podem tomar a forma de + ou - conforme sejam argumentos indexados ou não. Este mecanismo pode mais tarde ser alargado de forma a inserir novas respostas de acordo com os critérios de optimização que o programador definir. É, por exemplo, possível definir se queremos a resposta mínima ou máxima encontrada, através da utilização dos operadores *min* e *max* respectivamente. Esta técnica é denominada tabulação com operadores de modo.

1.1 Objectivos da Tese

Nesta tese abordamos a aplicação e a implementação de operadores de modo em programas lógicos tabelados. Os parágrafos que se seguem descrevem sumariamente as diversas áreas de investigação da Programação em Lógica em que estes operadores foram por nós aplicados.

A *Justificação* é um tópico de investigação bastante activo que consiste em gerar provas para as respostas obtidas por um dado programa [22]. As duas técnicas mais conhecidas para gerar justificações são a justificação online [16], que permite que as provas para as respostas encontradas sejam geradas em tempo de execução, e a justificação em pós-processamento [22], que tal como o nome indica gera as provas após a execução. Quando comparadas, a justificação online apresenta como principal vantagem o facto de ser bastante mais rápida, ao passo, que a justificação em pós-processamento não exige que o programa seja alterado. Nesta tese iremos abordar um método de justificação online que recorre aos operadores de modo da *tabulação* [7]. Este método apresenta como principais vantagens ser mais simples de implementar e bastante mais elegante que os outros métodos de justificação online, mantendo no entanto a mesma eficiência ao nível do desempenho.

A outra área de investigação que abordamos é a das *Preferências* [6, 5]. Esta técnica desenvolvida por Govindarajan et al. é bastante utilizada em problemas de optimização uma vez que apresenta uma forma intuitiva de dividir os programas lógicos. Os programas passam a ter duas partes: a especificação do problema e a definição da solução óptima. Guo et al. propuseram um programa de transformação que permite utilizar o conceito de preferências com os operadores de modo da tabulação. Contudo, o programa resultante da transformação apresenta limitações, não permitindo que as variáveis da consulta sejam todas livres. Nesta tese propomos alterações ao programa final que permitem ultrapassar estas limitações.

Para além destas duas áreas de investigação, nesta tese abordamos uma terceira área capaz de ser implementada usando operadores de modo que dá pelo nome de *answer subsumption* [18]. Em particular as preferências podem ser vistas como um sub-caso da *answer subsumption*. No caso das preferências, as cláusulas que visam optimizar o resultado são sempre aplicadas sobre duas respostas: a que era considerada o resultado óptimo até à altura e a que tenha sido encontrada entretanto. No caso da *answer subsumption*, as cláusulas de preferência podem ser aplicadas sobre a nova resposta encontrada e sobre um grupo de respostas já avaliado. Existem várias implementações de *answer subsumption* como as que estão presentes no sistema XSB Prolog. A nossa implementação é ainda mais poderosa, permitindo que da nova resposta e de um grupo de respostas surja uma terceira resposta em resultado da aplicação das cláusulas do problema em causa.

Para que os operadores de modo possam ser utilizados é necessário que o interpretador de Prolog tenha suporte para a tabulação e para estes operadores. Nesta tese demonstramos as alterações que necessitamos fazer para que o sistema YapTab fosse capaz de lidar com este tipo de operadores. As alterações principais foram feitas ao nível da função de inserção de subgolos e de respostas na tabela.

1.2 Estrutura da Tese

Iremos de seguida fazer uma descrição sumária dos capítulos que constituem esta tese.

Capítulo 1: Introdução O capítulo actual.

Capítulo 2: Programação em Lógica e Tabulação Apresenta os tópicos fundamentais para a compreensão desta tese. Começamos por introduzir conceitos básicos como a Programação em Lógica e os programas lógicos. Relatamos sumariamente

a cronologia do aparecimento da linguagem Prolog e explicamos as principais características dos seus interpretadores. Terminamos explicando o funcionamento da tabulação e de que forma pode ser implementada a tabela utilizando a estrutura de dados das *tries*.

Capítulo 3: Tabulação com Operadores de Modo Introduz uma nova forma de declarar predicados tabelados que através de operadores de modo permitem implementar novas funcionalidades na tabulação. Ilustra com exemplos práticos a utilização desses operadores.

Capítulo 4: Justificação, Preferências e *Answer Subsumption* Este capítulo apresenta três áreas de investigação em que os operadores de modo podem ser utilizados. Explica os conceitos base de cada uma das áreas apresentando exemplos e o trabalho relacionado.

Capítulo 5: Implementação Neste capítulo apresentamos as alterações que foram feitas ao YapTab de forma a dar suporte aos operadores de modo.

Capítulo 6: Avaliação Neste capítulo analisamos e avaliamos os tempos de execução obtidos pelos mecanismos propostos nesta tese quando estes implementam programas que calculam o caminho mínimo de um grafo.

Capítulo 7: Conclusão Sumaria o trabalho realizado no âmbito desta tese e as suas principais contribuições. Apresenta alguns tópicos que podem ser desenvolvidos no futuro.

Capítulo 2

Programação em Lógica e Tabulação

Este capítulo pretende enquadrar as áreas de investigação abordadas por esta tese, fazendo uma descrição sumária dos aspectos relevantes de cada uma delas. Inicialmente discute-se a Programação em Lógica com especial ênfase na linguagem Prolog e fazendo referência à WAM que é a máquina virtual mais implementada pelos interpretadores desta linguagem. Numa segunda parte é feita uma breve introdução à técnica da Tabulação descrevendo-se as suas características principais e de que forma pode ser utilizada em programas Prolog.

2.1 Programação em Lógica

A Programação em Lógica está incluída no grupo das linguagens declarativas tendo, por isso, uma forte base matemática. Neste tipo de linguagens o programador descreve o problema em vez de se preocupar com os detalhes para o tentar resolver. São, por isso, consideradas linguagens de mais alto nível, quando comparadas com o paradigma de programação imperativo, estando por isso mais próximas do raciocínio humano.

Escrever um programa, recorrendo a uma linguagem lógica, torna-se bastante simples, bastando definir factos e estabelecer relações entre eles. Dada uma consulta e um programa é utilizada a lógica de primeira ordem para procurar respostas que satisfaçam o problema. Segundo Carlsson [2] as linguagens lógicas apresentam as seguintes vantagens:

- **Semântica declarativa simples:** um programa é um conjunto de cláusulas de predicados lógicos;
- **Semântica procedimental simples:** um programa lógico pode ser visto como um conjunto de procedimentos recursivos;
- **Grande poder expressivo:** os programas são especificações executáveis que, mesmo sendo simples a nível procedimental, permitem a implementação de algoritmos complexos e eficientes;
- **Semântica não determinística:** em geral várias cláusulas podem unificar com um golo, tornado-se fácil programar algoritmos de pesquisa neste tipo de linguagens.

Estas características permitem que os programas lógicos sejam mais concisos e desenvolvidos mais rapidamente.

2.1.1 Programas Lógicos

Um programa lógico é constituído por um conjunto de cláusulas de Horn. Cada cláusula deste tipo apresenta como principal característica o facto de somente possuir um literal positivo. A cláusula de Horn mais simples é chamada de facto e não possui literais negativos sendo representada, em notação Prolog, por:

$$\boldsymbol{A.}$$

e pode ser lida como *"A é verdade"*.

As cláusulas mais complexas são chamadas regras e apresentam n literais negativos e um positivo, sendo representadas por:

$$\boldsymbol{A:-}\ B_1\boldsymbol{,}\ B_2\boldsymbol{,}\ \boldsymbol{...,}\ B_n$$

em linguagem corrente tem o seguinte significado *"A é verdade se* B_1 *é verdade e* B_2 *é verdade ... e* B_n *é verdade"*. Estas cláusulas são usadas, em programas lógicos, para definir relações entre factos e/ou regras. O literal A é chamado cabeça da cláusula, sendo o conjunto dos outros literais chamado corpo da cláusula, e cada um dos seus

constituintes, os *Bi*, denominados subgolos. Para retirar informação de um programa são utilizadas cláusulas em que a cabeça da mesma é vazia. Estas cláusulas são denominadas por consultas:

$$:\!\!- \ B_1, \ B_2, \ \ldots, \ B_n$$

Cada um dos literais que compõe uma cláusula está escrito na forma:

$$p(t_1,t_2,\ldots,t_n)$$

onde p é o functor de aridade n e os t_i representam termos que podem ser constantes, variáveis ou outros termos compostos.

O Prolog utiliza a resolução *Selective Linear Definite* (SLD) em que as cláusulas são testadas pela ordem em que aparecem no programa e os subgolos de cada cláusula são seleccionados da esquerda para a direita. A computação deste tipo de programas compreende dois mecanismos essenciais: a unificação e o mecanismo de *backtracking*.

Para melhor percebermos este tipo de mecanismos e os programas lógicos de uma forma geral vejamos o exemplo da Figura 2.1. O programa em Prolog, no topo da figura, é constituído por uma regra e três factos, sendo que a regra pode ser interpretada como "X é avô de Z se X é pai de Y e Y é pai de Z" e o primeiro facto pode ser lido como "o josé é pai do carlos". Supondo que queríamos saber "quem é o avô do nuno" bastaria executar a seguinte consulta *'avo(X, nuno)'*. A árvore de execução resultante encontra-se representada na parte de baixo da figura com os passos numerados de acordo com a ordem que são executados. Segue-se a descrição sucinta de cada um deles.

1. Começamos por executar a consulta:

   :- avo(X,nuno).

2. O golo da consulta unifica com a cabeça da única regra do programa, gerando o seguinte resultado:

   :- pai(X,Y), pai(Y,nuno).

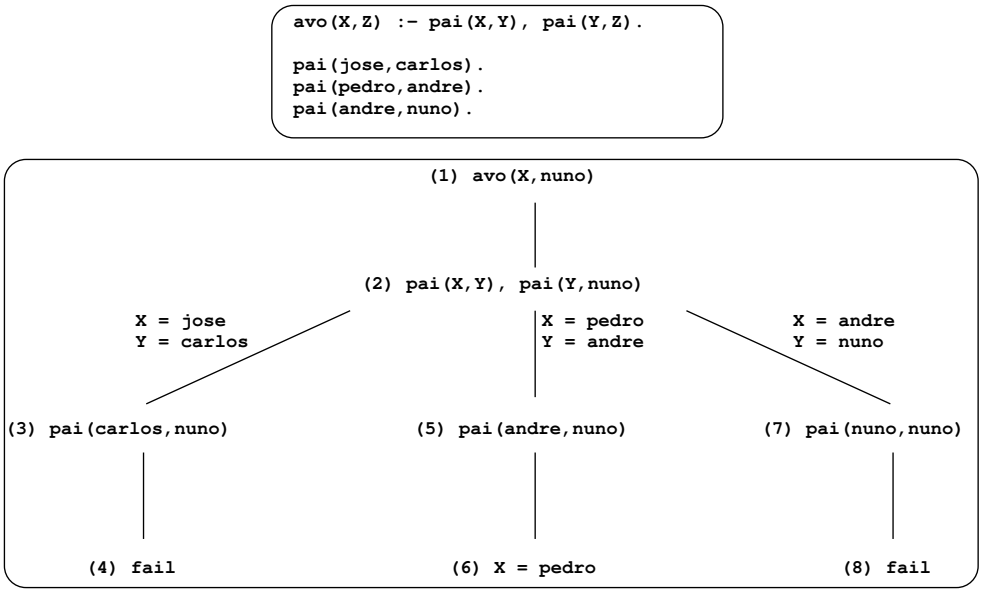

Figura 2.1: Exemplo da avaliação de um programa Prolog.

3. De seguida seleccionamos o subgolo mais à esquerda para continuar a resolução. Este unifica com o primeiro facto *pai(jose,carlos)* gerando a substituição { $X \leftarrow$ *jose,* $Y \leftarrow$ *carlos* }. O subgolo seguinte, depois do processo de substituição, é agora:

$$pai(carlos,nuno)$$

4. Mas, uma vez que nenhum facto no programa unifica com este subgolo este falha. Ocorre então o processo designado por *backtracking*, em que se restaura a computação para o estado anterior com cláusulas alternativas por explorar:

$$:\text{-}\ pai(X,Y),\ pai(Y,nuno).$$

5. É agora considerado o facto *pai(pedro,andre)* com a seguinte substituição { $X \leftarrow$ *pedro,* $Y \leftarrow$ *andre* }, ficando o subgolo seguinte como *pai(andre,nuno)*. Este facto, *pai(andre,nuno)*, existe no programa o que significa que foi encontrada uma solução para a consulta inicial, dando origem à seguinte resposta:

$$X = pedro$$

6. Mesmo depois de ter sido encontrada uma solução o programa não termina, podendo o utilizador continuar a procurar as restantes alternativas. Os programas em Prolog permitem testar sempre todas as hipóteses em aberto, e neste

programa o facto *pai(andre,nuno)* ainda não foi testado. Depois de testado, no passo 8 da figura, verificamos que não é uma solução válida e o nosso programa termina por fim.

2.1.2 Prolog

A linguagem lógica mais conhecida e utilizada é o Prolog. O nome Prolog deriva da abreviatura das palavras *PROgramation en LOGic* e foi cunhado por Colmerauer e colegas no trabalho [3]. No entanto, tudo começou em 1965 com o trabalho de Robinson em que este descrevia o processo de unificação e resolução [19]. Anos mais tarde, no início da década de setenta, Kowalski e Colmerauer [12, 3] provam que as cláusulas de Horn podem ter um significado declarativo *"A é verdadeiro se* B_1 *e ...* B_n *são verdadeiros"* e um significado procedimental: *resolver A, é o mesmo que resolver* B_1 *e ...* B_n.

Já no final da década de setenta, David H. D. Warren apresenta o primeiro compilador de Prolog [23], o que ajudou esta linguagem a ganhar adeptos ao demonstrar que esta poderia ser utilizada para implementar eficientemente uma grande quantidade de algoritmos. Em 1983, Warren propõe uma nova máquina abstracta para executar código Prolog [24], com o nome de *Warren's Abstract Machine* (WAM) que se tornou na base da maior parte dos interpretadores de Prolog dos dias de hoje.

Apesar do sucesso da WAM, para tornar esta linguagem mais adequada à programação do dia a dia foi necessário introduzir algumas funcionalidades que não estão presentes na lógica de primeira ordem, na qual o Prolog se baseia:

- **Predicados meta-lógicos** - têm como principal função dar ao programador um maior controle sobre a execução do programa, permitindo saber detalhes sobre o estado da computação e manipular termos.
- **Predicado de corte** - este predicado permite controlar o mecanismo de *back-tracking* do Prolog e cortar hipóteses não exploradas da computação o que leva a que seja possível reduzir o espaço de procura para assim conseguir programas mais eficientes. Como desvantagem pode levar a que os programas escritos sejam menos legíveis.
- **Predicados não lógicos** - estes predicados são assim chamados por não terem nenhum significado lógico. São utilizados para realizar tarefas de *input/output* ou manipular a base de dados interna do Prolog.

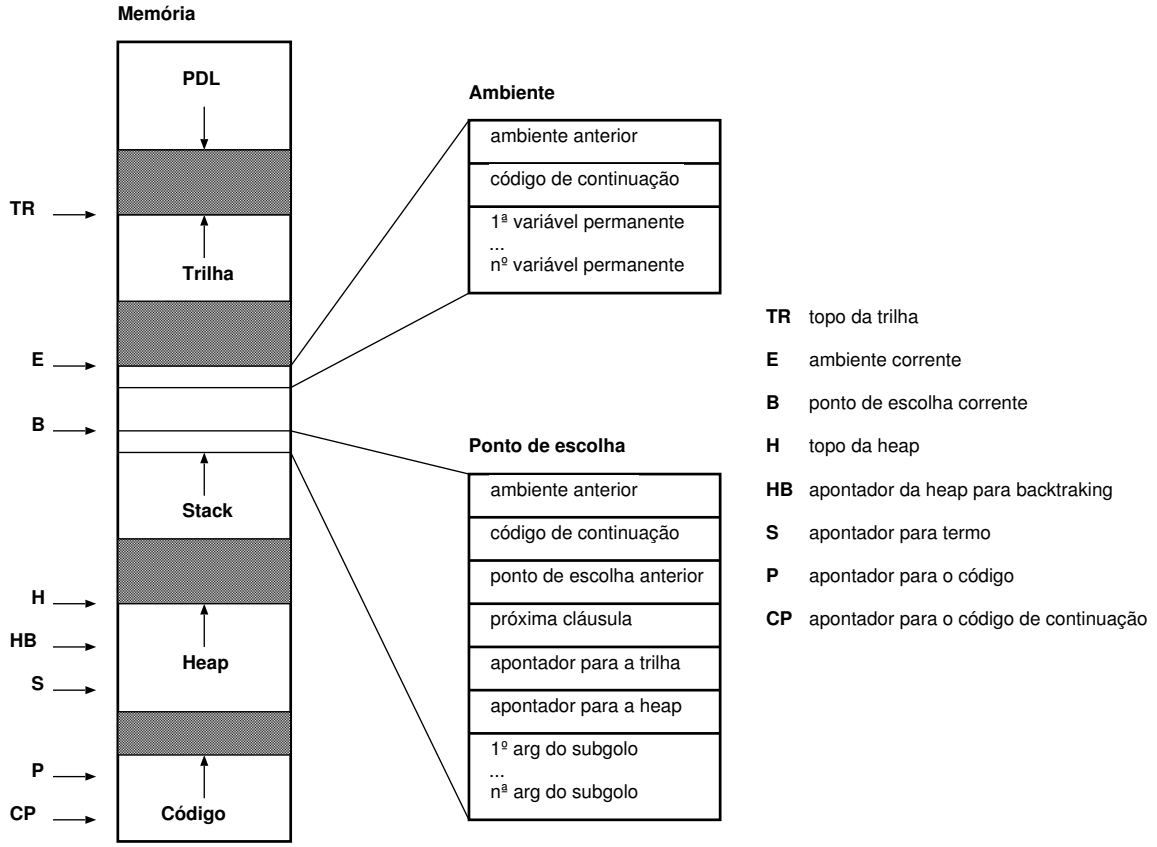

Figura 2.2: Representação esquemática da WAM.

- **Outros predicados** - nesta categoria estão incluídos predicados que permitem realizar, por exemplo, operações aritméticas, operações de controle simples e de *debugging*.

2.1.3 Warren's Abstract Machine

O sucesso do Prolog deve-se em grande parte ao sucesso da WAM, sucesso esse que é comprovado pelo facto da WAM se ter tornado no standard dos interpretadores de Prolog. Esta máquina virtual tem duas especificações fundamentais: a arquitectura da memória e o conjunto de instruções que permitem executar o código Prolog.

No que respeita à arquitectura da memória, a WAM é composta por cinco pilhas de execução e por um conjunto de registos, conforme representado na Figura 2.2. De seguida descreve-se, sumariamente, a função de cada uma das cinco pilhas de execução:

- **PDL (Push Down List):** é uma pilha auxiliar utilizada pelo processo de

unificação.

- **Trilha:** durante a execução as variáveis podem ser instanciadas, contudo quando é efectuado *backtracking* o seu estado anterior precisa de ser restaurando. Para que isto seja possível, todas as atribuições feitas a variáveis, que possam ser afectadas pelo processo de *backtracking*, são registadas nesta zona da memória. O registo TR marca o topo desta pilha.

- **Stack:** nesta pilha são guardados os pontos de escolha e os ambientes:

  - Os pontos de escolha guardam o estado da computação para que seja possível recuperá-lo no processo de *backtracking*. São criados sempre que uma chamada possa ter várias hipóteses de resolução, e nele é guardada toda a informação necessária para que seja possível restaurar o estado da máquina a fim de permitir a execução das hipóteses que ficaram em aberto. O registo B guarda o endereço para o ponto de escolha corrente.

  - Os ambientes são criados e colocados na stack sempre que uma cláusula que contem vários subgolos é executada. São utilizados para guardar informação de controle que permita retomar a execução no subgolo seguinte e para guardar as variáveis permanentes, isto é, as variáveis que aparecem em mais do que um subgolo. Os ambientes são constituídos por um endereço que guarda o ambiente anterior, outro endereço que aponta para o código a ser executado caso esta cláusula execute com sucesso e ainda pelas variáveis permanentes da cláusula. O registo E guarda o endereço do ambiente corrente.

- **Heap:** é a área de dados utilizada para representar as variáveis e os termos Prolog. O registo H marca o topo desta área.

- **Área de Código:** nesta área estão as instruções WAM dos programas carregados.

Para além dos registos já mencionados a WAM utiliza ainda o registo S para o processo de unificação de termos compostos, o registo HB para a determinação das atribuições condicionais isto é, a serem guardadas na trilha, o registo P aponta para a instrução WAM que está a ser executada e o registo CP aponta para onde retornar após uma execução com sucesso da cláusula corrente.

Relativamente às instruções da WAM estas foram desenhadas de modo a que [11]:

- fosse fácil mapear código Prolog nestas instruções;
- pudessem ser facilmente traduzidas ou emuladas para código nativo do processador.

As instruções WAM tem um papel importante na execução de código Prolog existindo quatro categorias principais a destacar:

- **Instruções de ponto de escolha** - são responsáveis pela manipulação dos pontos de escolha e pela utilização dos dados lá contidos de modo a permitir o restauro do estado da computação.
- **Instruções de controle** - alocam ou removem ambientes e são responsáveis pela gestão da sequência de chamada e retorno dos subgolos.
- **Instruções de unificação** - existem vários tipos de unificação consoante o tipo de argumentos e a posição em que eles se encontram.
- **Instruções de indexação** - estas permitem a aceleração do processo de selecção da cláusula que unifica com uma dada chamada de um subgolo.

Uma introdução bastante detalhada da WAM é feita por Ait-Kaci no seu tutorial sobre a WAM [1].

2.2 Tabulação

Apesar da popularidade do Prolog, o mecanismo de resolução SLD têm-se demonstrado pouco adequado em alguns casos, podendo levar à ocorrência de ciclos infinitos. Na Figura 2.3 podemos ver um desses exemplos.

Este exemplo é constituído por dois predicados, o predicado *edge/2* define um pequeno grafo e o predicado *path/2* define uma relação de conectividade entre os pontos do grafo. Como podemos ver, na parte de baixo da figura, ao executarmos a consulta *path(a,Z)* o programa entra em ciclo infinito. Isto deve-se ao facto de a exploração do ramo esquerdo da árvore levar a que seja chamado indefinidamente o subgolo *path(a,Y)*. A resolução deste tipo de problema pode ser conseguida por utilização de uma técnica chamada de *tabulação*. A tabulação consiste em guardar e reutilizar os resultados das sub-computações durante a execução de um programa [15]. Nesta

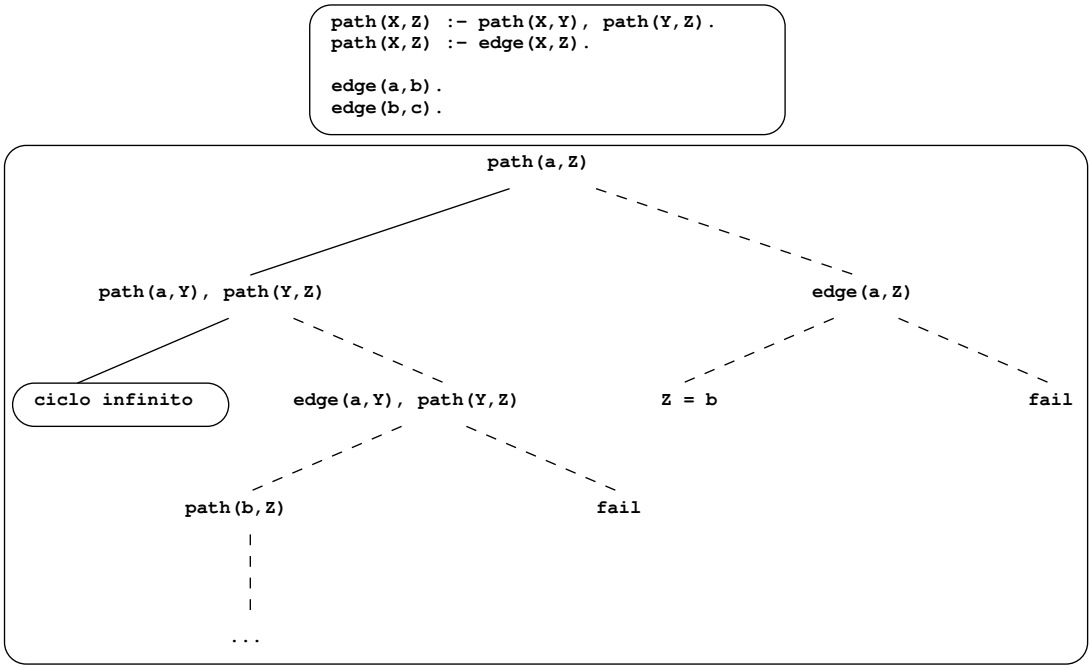

Figura 2.3: Programa onde ocorre um ciclo infinito por resolução SLD.

tese será dado ênfase ao YapTab [20] que é o modelo de tabulação do sistema Yap Prolog. Para que seja possível utilizar a tabulação na avaliação de um programa é necessário declarar no início do mesmo os predicados a tabelar. Por exemplo, caso queiramos tabelar o predicado *path/2* devemos incluir a declaração *table path/2*.

O programa da Figura 2.4 é o mesmo da Figura 2.3, com a diferença do predicado *path/2* estar declarado como sendo tabelado. Daí, a figura incluir no canto superior direito uma nova componente - a tabela. Esta estrutura tem como função guardar os subgolos e as respostas encontrados durante a execução dos predicados tabelados. A árvore que se encontra na parte de baixo ilustra a sequência de execução do programa e está numerada pela ordem pela qual a avaliação é feita. Vejamos agora em pormenor a execução do programa. Começamos por executar a consulta *path(a,Z)*, mas por este subgolo se tratar de um predicado tabelado, e ser a primeira vez que é chamado, é adicionada uma entrada à tabela. Em seguida é seleccionada a primeira cláusula do programa que unifica com a consulta, levando à criação do nó 1 da árvore. A computação passa agora para o subgolo *path(a,Y)*, que por se tratar de uma variante da chamada do nó 0, leva a que a tabela seja consultada. Ao verificar-se que não existem respostas para este subgolo na tabela, a computação neste nó é suspensa o que permite evitar a ocorrência do ciclo infinito do exemplo anterior. Passa a ser testada a segunda cláusula do predicado *path/2* e o nó 2 é criado. No decorrer da

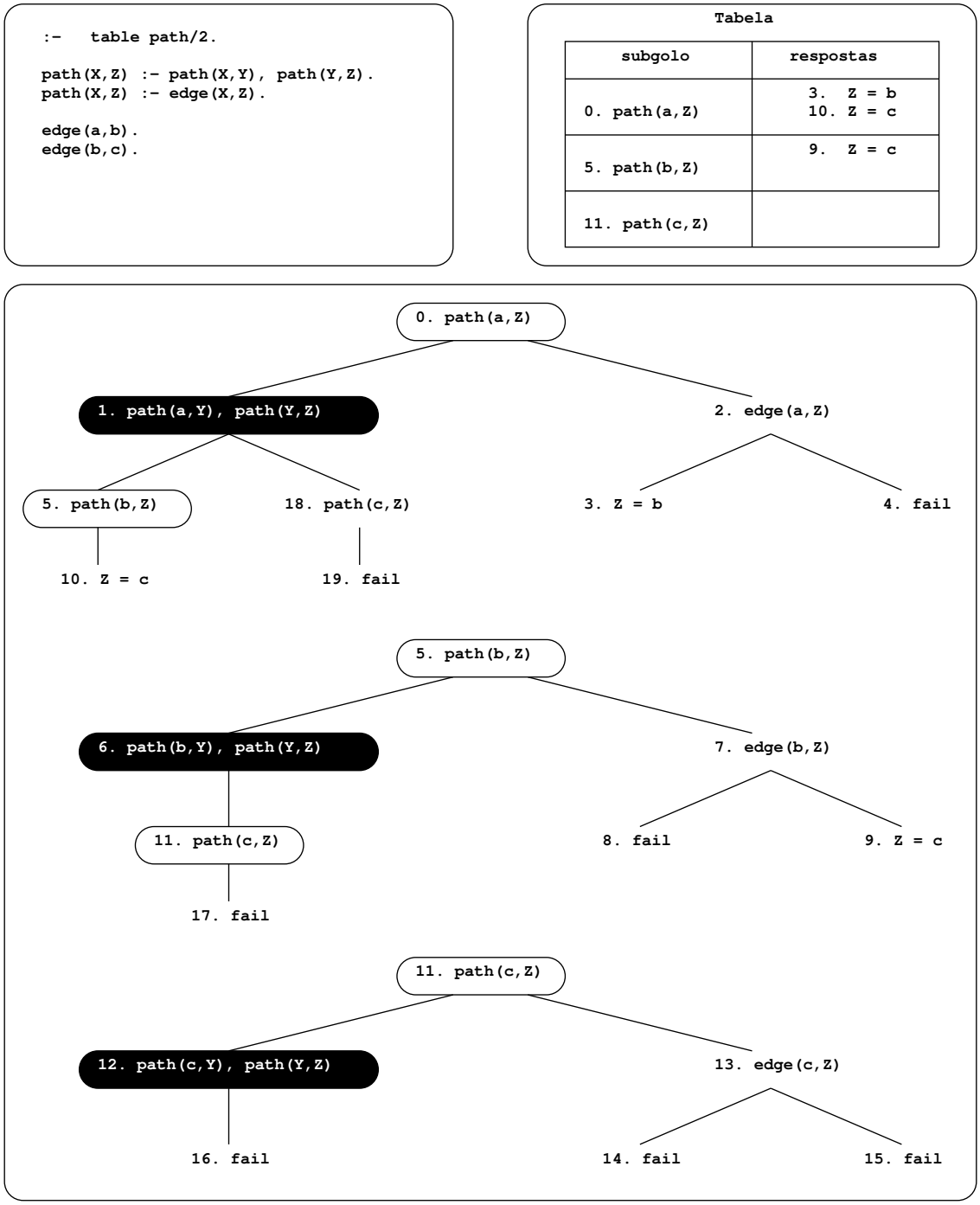

Figura 2.4: Exemplo da avaliação de um programa tabelado.

computação do nó 2 é encontrada a resposta $Z = b$ e inserida na tabela para *path(a,Z)* (passo 3). Esta resposta permite mais tarde reactivar o nó 1. A substituição originada pelo consumo desta resposta gera a chamada ao subgolo *path(b,Z)* que por se tratar de uma nova chamada tabelada é adicionada uma nova entrada na tabela (passo 5). A computação a partir deste ponto prossegue como anteriormente, sendo descoberta mais tarde uma nova resposta para a consulta inicial (passo 10), e adicionada mais uma entrada à tabela para o subgolo *path(c,Z)* (passo 11). Por não haver mais hipóteses em aberto ocorre *backtracking* para o subgolo de topo (nó 0) que fica completamente avaliado e sucede com duas respostas, conforme podemos ver na tabela.

Outra vantagem a destacar na tabulação é a eficiência. Conforme descrito anteriormente este mecanismo reutiliza partes da computação, o que permite evitar que sejam calculadas várias vezes as mesmas chamadas de um predicado tabelado. Na Figura 2.5 temos um programa para calcular o número de Fibonacci onde esta situação é ilustrada. Como podemos ver na parte em que é usada a resolução SLD, o *fib(2)* é calculado várias vezes ao passo que com a tabulação o resultado é calculado uma única vez sendo depois reutilizado.

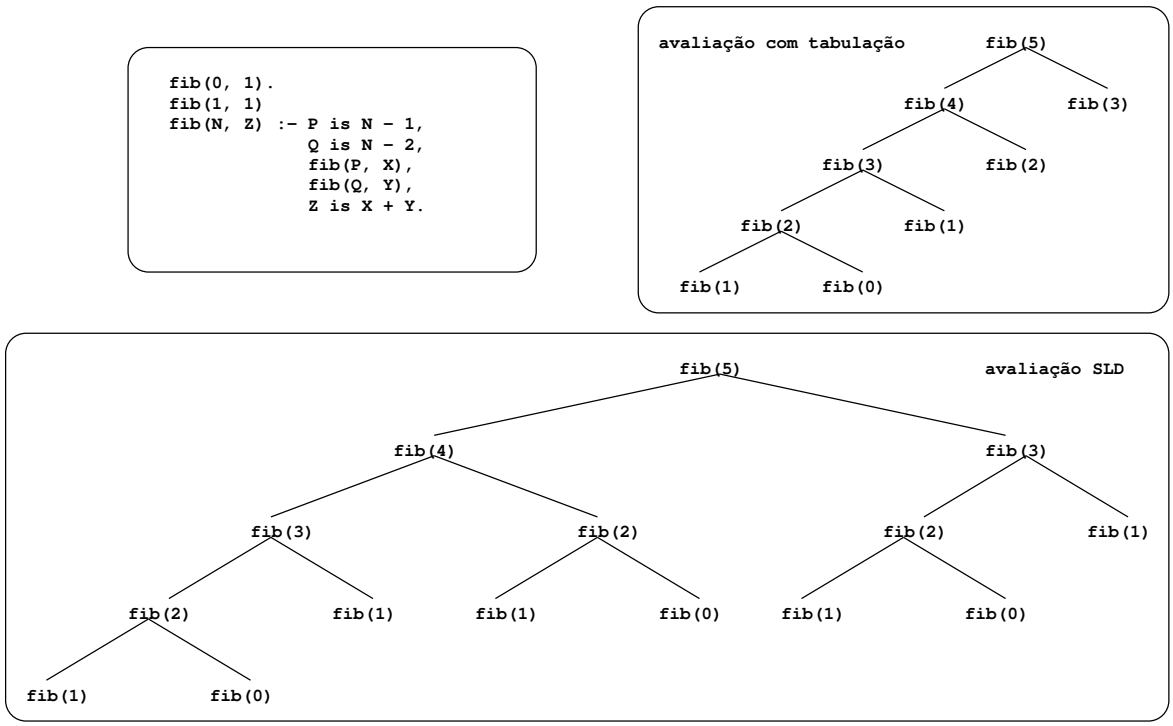

Figura 2.5: Comparação a nível da eficiência entre a tabulação e a avaliação SLD.

2.2.1 Estratégias de Escalonamento

Quando executamos um programa utilizando o sistema YapTab, estão ao nosso dispor duas estratégias de escalonamento: *batched* e *local* [20]. A escolha da estratégia a utilizar pode ter impacto no tempo de execução do programa e no consumo de memória.

A estratégia de escalonamento *batched*, privilegia a computação em profundidade pois sempre que é encontrada uma nova resposta é adicionada à tabela e a avaliação continua. Por vezes este tipo de estratégia pode levar a que haja maiores dependências entre os subgolos tabelados, o que aumenta a complexidade da computação. No que respeita ao mecanismo de escalonamento *local*, este tenta avaliar cada subgolo da forma mais independente possível e quando uma nova resposta é encontrada é adicionada à tabela mas, neste caso, a execução falha para permitir que aquele subgolo seja completamente avaliado, ou seja, que todas as respostas para aquele subgolo sejam encontradas antes de as devolver para o subgolo de continuação.

2.2.2 Tries

Como vimos, a tabulação faz uso de uma tabela para guardar as chamadas e as respostas de predicados tabelados encontradas durante a execução de um programa. Se pensarmos na quantidade de vezes em que pode ser necessário consultar e actualizar a tabela, é fácil perceber que a eficiência destas operações pode ter um grande impacto no desempenho global do sistema de tabulação. É por isso necessário tirar partido de uma estrutura de dados que permita que essas operações sejam realizadas de um modo bastante eficiente. O YapTab utiliza *tries* [20] que, para além de garantirem o requisito anterior, permitem que haja um consumo de memória eficiente ao evitarem a representação repetida dos elementos dos subgolos [17]. Considerando o seguinte exemplo, suponhamos que queremos inserir na tabela os subgolos *f(a,t(X,b),X)* e *f(b,X,c)*. A representação numa trie ficaria semelhante ao que podemos ver esquematicamente na Figura 2.6.

Se mais tarde fosse necessário inserir o subgolo *f(a,t(X,a),Y)*, como representa a Figura 2.7, bastaria ir percorrendo a trie já existente da seguinte forma:

- começamos por considerar o primeiro argumento do subgolo, que neste caso é a constante *a*, como este termo já está representado na trie não é necessário repeti-lo.

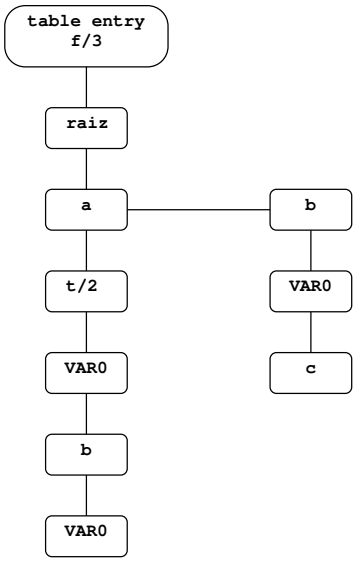

Figura 2.6: Trie do predicado *f/3* com os subgolos *f(a,t(X,b),X)* e *f(b,X,c)*.

- Passamos agora a considerar o segundo argumento do subgolo que é o termo complexo *t/2*, cujos argumentos são uma variável e a constante *a*. Quando comparamos estes elementos com os que já estavam na trie vemos que a única diferença entre eles é a constante *a* em vez de *b*. É por isso acrescentado um novo nó contendo a constante *a* no mesmo nível do nó contendo *b*.

- Por fim resta verificar o terceiro argumento do termo que por ter sido criado um nó no nível acima é inserido sem ser necessário fazer qualquer tipo de comparação.

Como podemos verificar os termos com prefixos comuns são representados uma única vez, o que permite reduzir a quantidade de memória necessária à sua representação. Por exemplo, neste caso concreto houve uma economia de 3 nós.

Para além de guardarmos os subgolos que são chamados durante a execução do programa, é também necessário guardar as respostas encontradas. Para tal, o último nó de cada subgolo aponta para a trie que contém as respostas conforme está representado na Figura 2.8. Cada trie de respostas tem tantos níveis quanto o número de variáveis distintas que ocorrem nesse subgolo. Observando a figura, podemos constatar que o subgolo *f(a,t(X,a),Y)* tem dois pares de respostas $\{X = a,\ Y = b\}$ e $\{X = c,\ Y = b\}$, o subgolo *f(a,t(X,a),X)* tem as respostas $X = b$ e $X = c$ (embora a variável X ocorra duas vezes no subgolo, a solução só necessita de ser representada na trie de respostas uma única vez). Por último, o subgolo *f(b,X,c)* tem $X = a$ e $X = c$ como soluções.

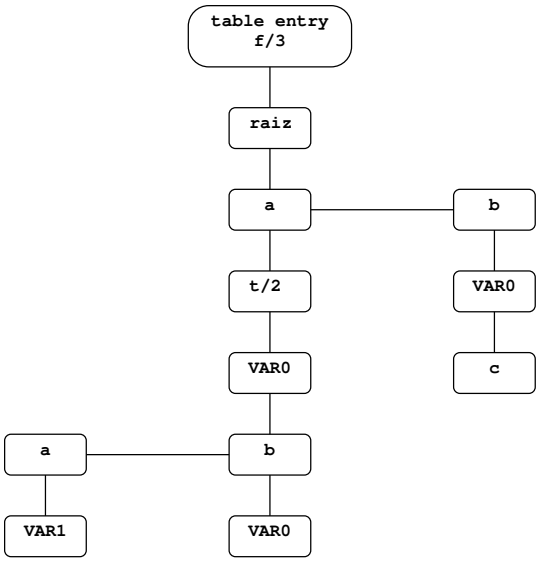

Figura 2.7: Nova representação da trie da Figura 2.6 caso fosse adicionado o subgolo *f(a,t(X, a),Y)*.

2.3 Resumo

Neste capítulo começamos por descrever as principais características das linguagens lógicas e de seguida abordamos a linguagem Prolog, a mais utilizada hoje em dia neste paradigma. Vimos que em grande parte a WAM tinha sido responsável por esse sucesso e referimos as suas principais características. Terminamos este capítulo com o mecanismo de Tabulação e apresentamos as suas vantagens em relação ao mecanismo de resolução SLD tradicional do Prolog.

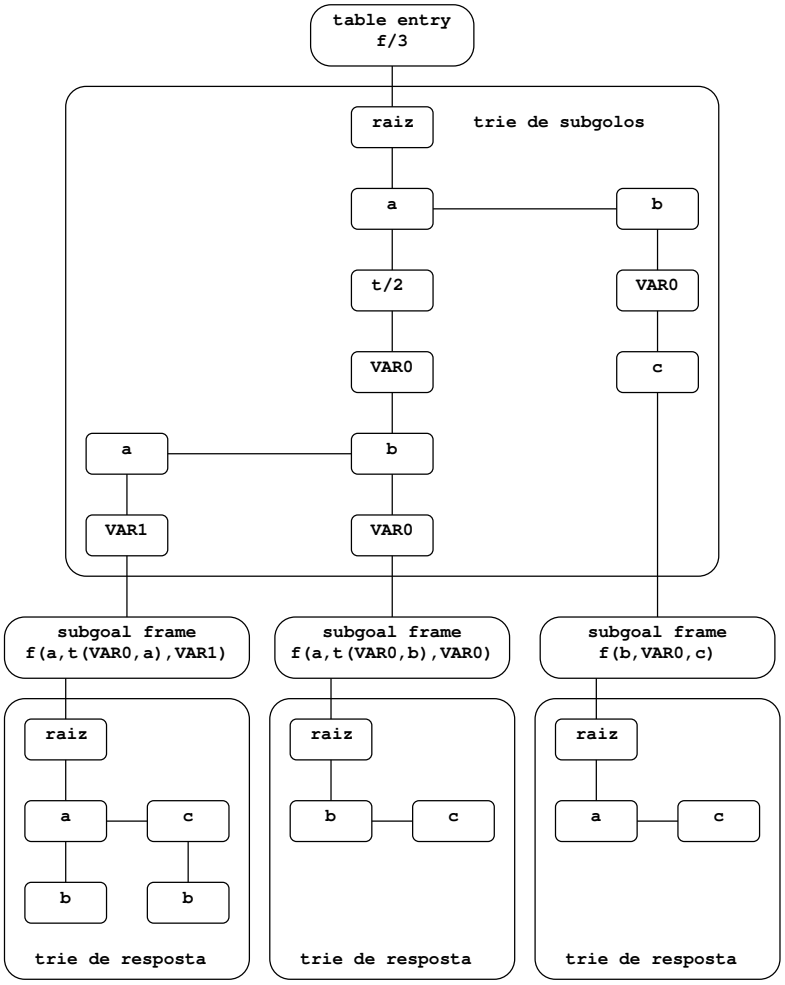

Figura 2.8: Representação esquemática de uma tabela.

Capítulo 3

Tabulação com Operadores de Modo

Neste capítulo apresentamos uma forma de declaração de predicados tabelados por utilização de operadores de modo que nos permite extender, de uma maneira simples e elegante, o poder declarativo da tabulação como forma de resolver problemas em programação dinâmica.

3.1 Ideia Geral

A programação dinâmica é muito utilizada para resolver problemas no âmbito de diversas áreas tais como a Ciência de Computadores, a Biologia Molecular [13], Investigação Operacional, entre outras. Este paradigma consiste em dividir o problema em sub-problemas mais simples que, depois de resolvidos, irão constituir a solução final. O cálculo do número de Fibonacci é um óptimo exemplo da utilização deste tipo de técnica. Conforme vimos na Figura 2.5 do capítulo anterior, calculamos o número de Fibonacci de 5 dividindo o problema no cálculo do número de Fibonacci de 4 até ao Fibonacci de 1. A tabulação, conforme vimos, é bastante eficaz a lidar com este tipo de problemas, pois evita que respostas para o mesmo sub-problema sejam calculadas diversas vezes.

Escrever um algoritmo de programação dinâmica pode ser bastante complicado. Normalmente este tipo de algoritmos têm por objectivo calcular a solução óptima para o problema sendo, por vezes, necessário passar recursivamente o argumento com a

solução óptima. Este tipo de estratégia pode levar à introdução de erros de difícil detecção. Um outro problema que normalmente surge é o de guardar uma justificação para cada resultado encontrado pois, caso haja várias explicações para um mesmo resultado, tal pode levar a um gasto bastante grande de memória [7]. Neste capítulo apresentamos uma forma de declarar predicados tabelados que nos permite resolver de forma elegante alguns destes problemas. Quando queremos definir um predicado como sendo tabelado basta escrever no topo do programa a declaração:

:- table p/3

onde *p/3*, é o predicado que queremos tabelar. Para incorporar as novas funcionalidades, acima descritas, passamos a declarar da seguinte forma [7]:

:-table $p(a_1,a_2,a_3)$

onde cada um dos a_i define o operador de modo que irá ser aplicado sobre os resultados do argumento respectivo. Nas próximas secções, iremos apresentar os operadores de modo que podem ser aplicados sobre os argumentos dos predicados tabelados, acompanhados de alguns exemplos.

3.2 Operadores de Indexação

No capítulo anterior referimos o facto da tabulação quebrar ciclos infinitos ao evitar a execução de chamadas repetidas. Mesmo no caso de haver um ciclo no grafo, como no exemplo representado na Figura 3.1, a tabulação consegue evitar potenciais ciclos infinitos. Na parte de baixo da figura, vemos a árvore de execução do programa. A consulta executada é *path(a,Z)* (passo 0) que irá unificar com a primeira cláusula do programa, dando origem por sua vez à chamada *path(a,Z)*, que por se tratar de uma chamada repetida, leva a que a computação seja suspensa nesse nó (passo 1). É agora considerada a segunda cláusula (passo 2), do programa o que leva à descoberta da resposta $Z = b$ (passo 3). Como foi visto anteriormente, é agora altura de reactivar o nó 1 para consumir a resposta encontrada (passo 4), o que leva à descoberta da resposta $Z = a$ (passo 5). Esta nova resposta irá, por sua vez, ser também consumida pelo nó 1 levando a que a resposta $Z = b$ seja encontrada novamente (passo 7). Quando descobrimos uma resposta, e a tentamos inserir na tabela, é verificado se é uma variante

das que já lá se encontram, isto é, se são iguais a menos da renomeação das variáveis. Neste caso a resposta encontrada no passo 7 é uma variante da encontrada no passo 3, o que leva a que não seja inserida na tabela e a execução falhe. De seguida, como não, existem mais respostas por consumir, o programa termina.

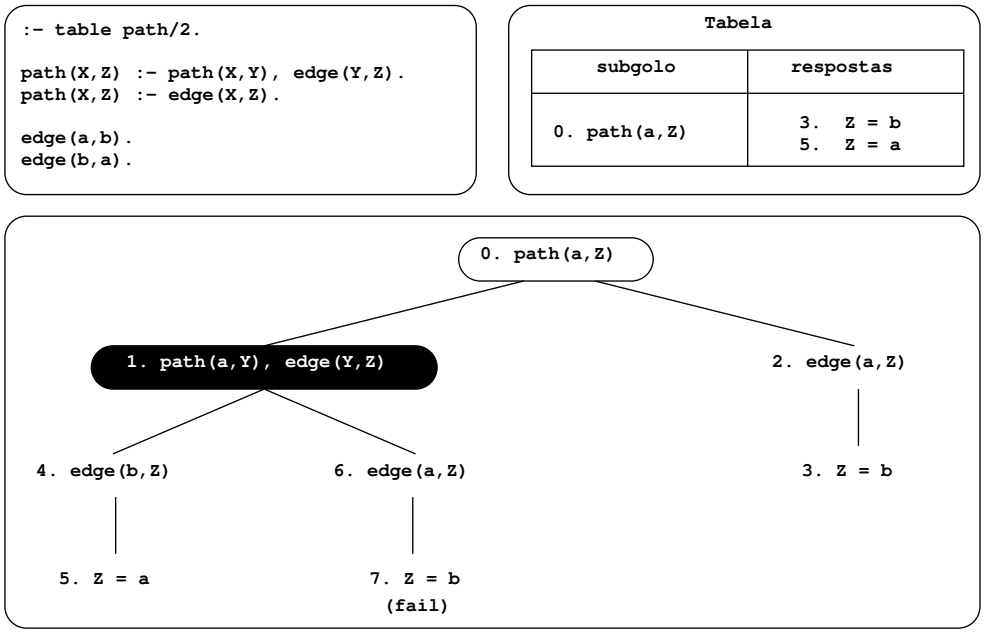

Figura 3.1: Avaliação com tabulação de um grafo com ciclos.

Se tal falha não acontecesse, o programa entraria em ciclo infinito, descobrindo alternadamente as respostas $Z = a$ e $Z = b$. A tabulação é bastante eficaz a evitar ciclos infinitos, contudo pequenas alterações ao programa podem levar a que isto nem sempre seja verdade. Por exemplo, se para além de calcular os caminhos de a para todos os pontos do grafo, quiséssemos também contar o número de passos necessários para chegar a esse caminho, poderíamos vir ter problemas a Figura 3.2 apresenta essa situação. Isto acontece porque existe um caminho com um número infinito de passos entre a e b. Neste caso o programa entra em ciclo infinito. Assim sendo as respostas encontradas, por exemplo, nos passos 7 e 9, embora sejam referentes aos mesmos vértices das que foram encontradas nos passos 3 e 5, não são variantes e por isso são inseridas na tabela na mesma.

Como iremos ver, o método de declaração de predicados tabelados de que falámos anteriormente, soluciona este de tipo problema. Recorrendo a dois operadores, + e -, que significam respectivamente que o argumento em causa deve ser indexado ou não, quando uma nova resposta está prestes a ser inserida na tabela, a verificação de que se trata de uma variante ou não passa a ser feita apenas sobre os elementos indexados.

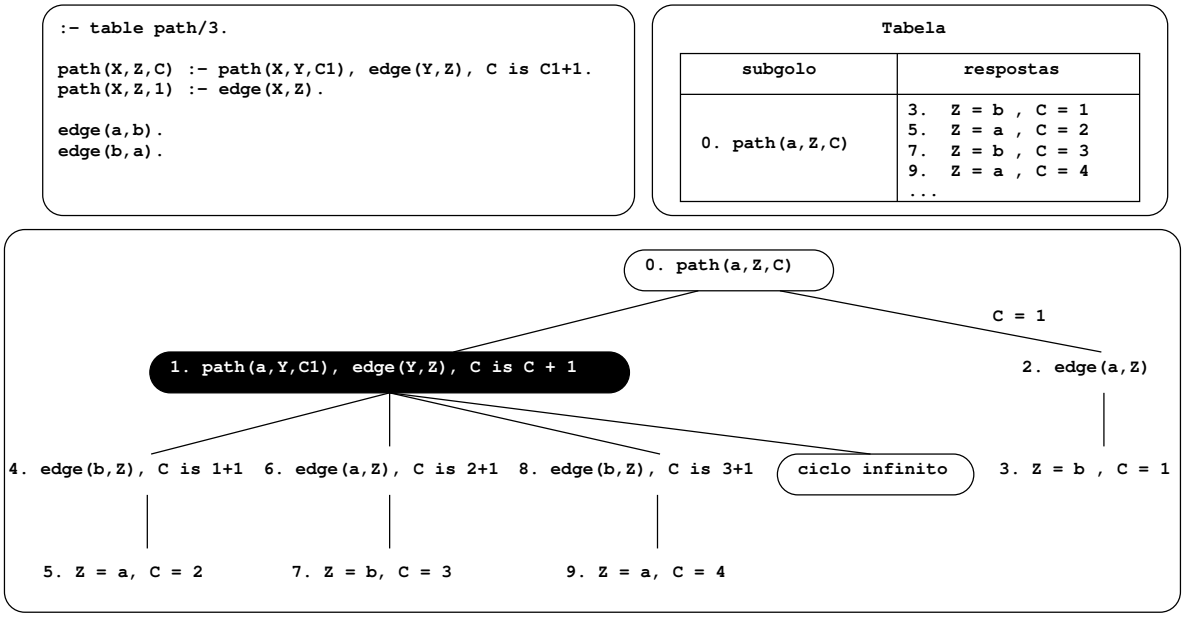

Figura 3.2: Avaliação com tabulação de um programa com um número infinito de respostas.

Alterar o problema anterior para utilizar este método é bastante simples. Sabendo que o problema reside no facto de haver uma infinidade de saltos entre os vértices *a* e *b*, basta definir este argumento como não indexado e os restantes como indexados. Na Figura 3.3 podemos ver em detalhe o programa modificado bem como a sua tabela e árvore de execução. Se o compararmos com o programa anterior, a única parte do código que foi necessário reescrever foi a da declaração do predicado tabelado que passou a ser *table path(+,+,-)*. Como podemos ver na Figura 3.3, a resposta $\{Y = b,\ C = 3\}$ já não é inserida na tabela. Isto acontece pelo facto de a resposta $\{Y = b,\ C = 1\}$ ser, para a declaração anterior, considerada uma variante da resposta $\{Y = b,\ C = 3\}$, levando a que esta última seja descartada e, por consequência, o ciclo é quebrado em oposição com o que acontecia no exemplo anterior da Figura 3.2.

3.3 Operadores de Agregação

O operador de não indexação apresentado anteriormente pode ser extendido de forma a permitir a agregação das repostas que vão sendo encontradas. No exemplo da Figura 3.3 é guardada apenas a primeira resposta encontrada, no entanto seria interessante que fosse o próprio utilizador a declarar que resposta deve ser guardada. Vão ser por isso introduzidos dois novos operadores, os operadores *max* e *min*, que permitem,

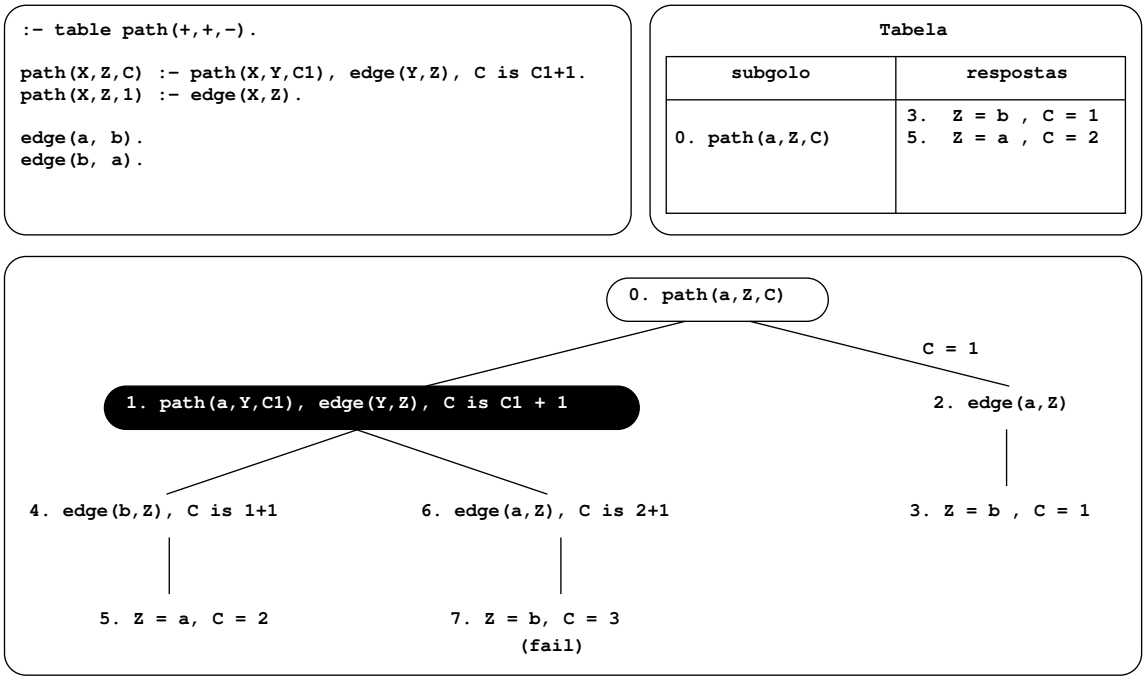

Figura 3.3: Utilizando a tabulação com os operadores de indexação + e -.

respectivamente, guardar a resposta máxima e a resposta mínima encontrada até aquele momento. Para melhor percebermos em que situações este tipo de operadores pode ser útil, vejamos o exemplo da Figura 3.4. O programa tem por objectivo encontrar o caminho de menor custo entre a e os restantes pontos do grafo. A declaração que deve ser feita no início é *table path(+,+,min)*, que significa que vamos indexar os primeiros dois argumentos, e que o terceiro elemento guardará a resposta mínima relativa aos dois primeiros argumentos.

Como podemos ver na árvore de execução da Figura 3.4 a avaliação segue a sequência de um programa tabelado normal. As respostas são colocadas na tabela conforme vão sendo encontradas. A parte mais interessante para vermos a diferença entre ter o operador - e o operador *min* no terceiro argumento acontece no passo 8 aquando da descoberta da resposta $\{Z = d,\ C = 3\}$. Caso se tratasse do operador - a resposta que estava na tabela seria mantida mas como este é um operador de agregação que guarda a resposta mínima, a resposta anterior, em que $C = 5$, é substituída pela nova resposta $C = 3$ e colocada na tabela.

O operado *max* funciona da mesma forma que o operador *min* com a diferença de guardar sempre a resposta máxima. Como seria de esperar é preciso algum cuidado no uso destes operadores pois o mau uso pode levar a que o programa continue a entrar em ciclo infinito, como aconteceria se usássemos o operador *max* no programa

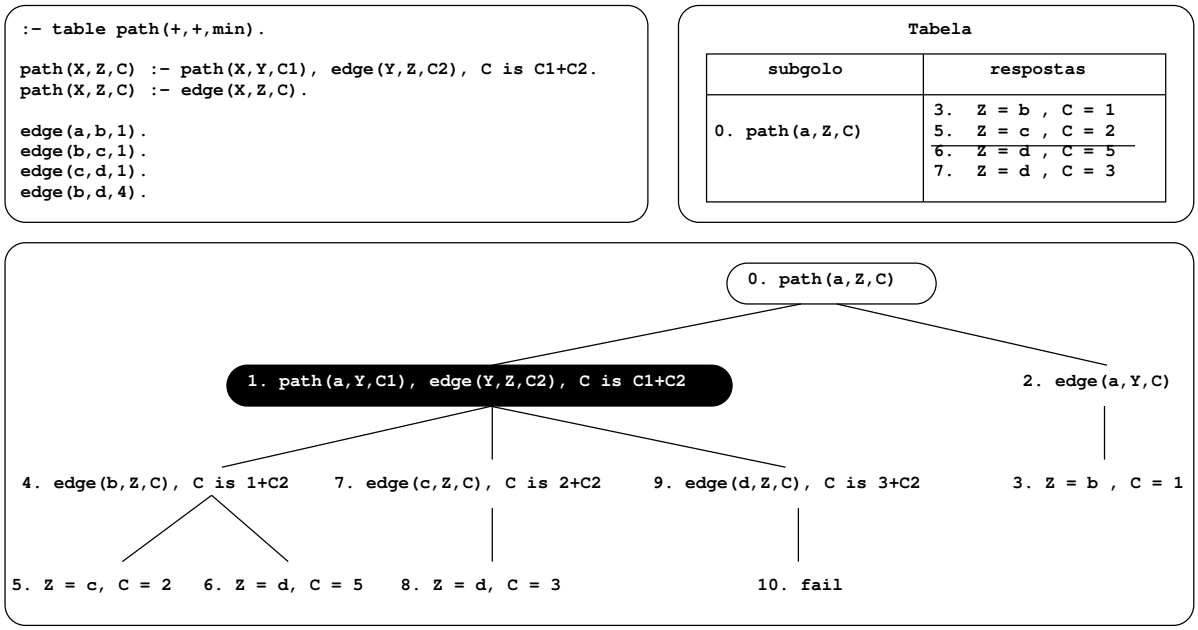

Figura 3.4: Utilizando a tabulação com o operador de agregação *min*.

da Figura 3.4.

Outro operador que pode ser bastante útil é o definido pelo simbolo @ e que nos permite guardar todas as respostas para um determinado argumento. Vejamos o exemplo da Figura 3.5, que é uma espécie de junção dos programas das Figuras 3.3 e 3.4. O programa é declarado com *table path(+,+,min,@)*, o que significa que queremos guardar o custo mínimo do caminho entre dois pontos no terceiro argumento e, ao mesmo tempo, queremos guardar no quarto argumento o número de passos dados pelos caminhos com custo mínimo, caso haja mais do que um. A Figura 3.5 apresenta o programa e a sua árvore de execução. O programa inicia-se tal como nos exemplos anteriores, sendo que a parte interessante acontece no passo 8 quando é descoberta a nova resposta $\{Z = b,\ C = 2,\ N = 2\}$. Aí verificamos que já existe uma variante na tabela, contudo a nova resposta é inserida na mesma na tabela porque apesar de ter o mesmo valor mínimo que a que já lá estava, tem um número de passos diferente. É importante notar que a agregação de respostas, feita pelo operador @, é realizada ao nível do argumento que conta o número de passos. Se por exemplo, fosse encontrada a respostas $\{Z = b,\ C = 1,\ N = 4\}$ as resposta $\{Z = b,\ C = 2,\ N = 2\}$ e $\{Z = b,\ C = 2,\ N = 3\}$ seriam eliminadas pois, neste caso, teria sido descoberto um caminho mais curto tal como definido pelo operador *min*.

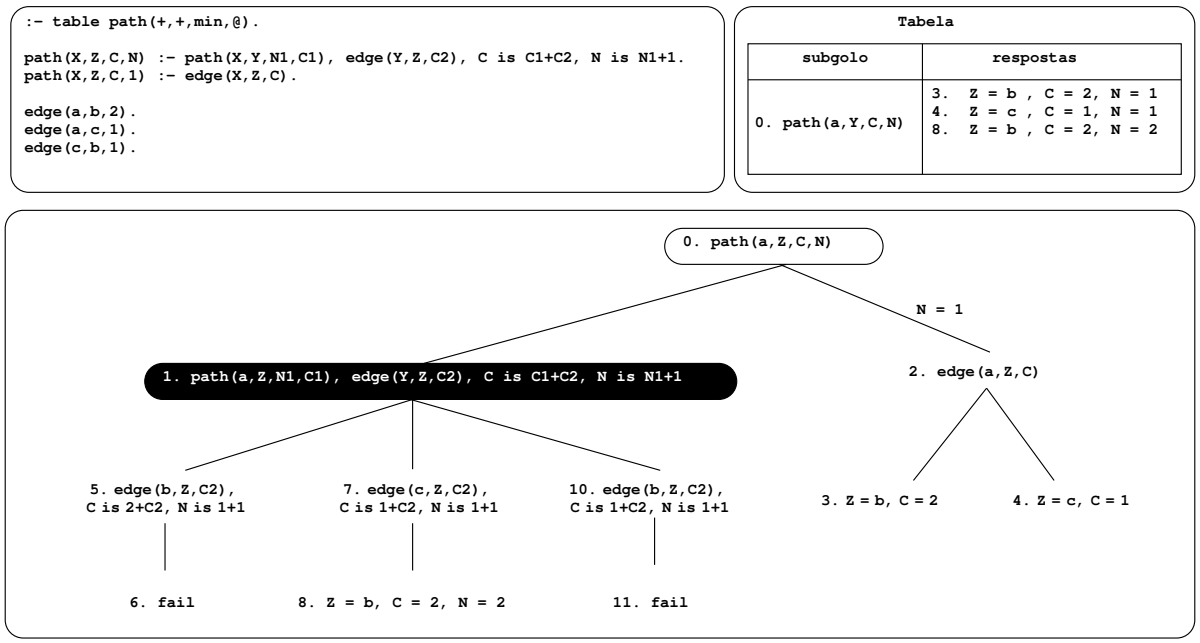

Figura 3.5: Exemplo da utilização do operador @ com o operador *min*.

3.4 Operador *last*

O operador *last* permite fazer o inverso do operador - ou seja, guarda sempre a última resposta encontrada. Este operador como vamos ver nos capítulos seguintes vai-nos permitir implementar um interface elegante para implementar um sistema de preferências e de *answer subsumption* em programas lógicos com tabulação.

3.5 Resumo

Neste capítulo abordamos um método diferente do habitual para declarar predicados tabelados que permite implementar funcionalidades úteis para alguns problemas de programação dinâmica. Descrevemos cada um dos operadores que podem ser utilizados neste método e ilustramos a sua funcionalidade com pequenos exemplos.

Capítulo 4

Justificação, Preferências e *Answer Subsumption*

Neste capítulo iremos demonstrar como os operadores introduzidos no capítulo anterior podem ser utilizados para resolver problemas de Justificação, Preferências e de *Answer Subsumption*. Discutiremos algum do trabalho relacionado desenvolvido nessas áreas e realçaremos as vantagens e as desvantagens da nossa abordagem.

4.1 Justificação

Quando a execução de um programa gera um conjunto de respostas pode ser importante ter acesso a uma prova de que aqueles resultados estão de facto correctos. Este tipo de provas ajuda o programador a perceber a origem dos resultados e, se for caso disso, a fazer *debugging*. Ao processo de geração de provas dá-se o nome de *Justificação*.

Considerando novamente o problema de encontrar todos os caminhos possíveis entre os vértices de um grafo, a justificação destas respostas seria, por exemplo, uma lista com todos os vértices que constituem um determinado caminho. Apesar desta aparente simplicidade, gerar uma justificação pode ser uma tarefa bastante complicada e tem sido um tópico de investigação bastante activo [22, 16].

Existem duas técnicas mais conhecidas e mais populares para gerar justificações. A primeira é feita em pós-processamento [22] e tem a vantagem de ser totalmente independente do processo de execução do programa. A outra é chamada de justificação

online [16] e é feita em simultâneo com a avaliação da consulta, apresentando como principal vantagem a maior eficiência com que é gerada. De seguida iremos ver em mais pormenor estas duas técnicas.

4.1.1 Justificação em Pós-Processamento

A justificação em pós-processamento é feita analisando as tabelas e/ou respostas geradas no final da execução e, posteriormente, interpretando-as de acordo com as cláusulas do programa [22]. Vejamos o exemplo que se segue para melhor compreendermos este método.

$$p\text{:-}\ q.$$

$$q.$$

Supondo que fazemos a consulta p, a resposta dada pelo interpretador de Prolog é que p é verdadeiro. A responsabilidade de fazer a análise e gerar uma justificação é de um meta-interpretador que irá pegar na resposta p e procurar a cláusula que lhe poderá ter dado origem. Conforme podemos verificar, o programa só tem uma cláusula capaz de gerar tal resultado: p:- q. Seguidamente o meta-interpretador irá aperceber-se de que q é um facto, podendo daí retirar que p é verdade por q ser verdade. Logo a resposta p é justificada com o facto q.

O exemplo que acabamos de ver é bastante simples. No entanto programas mais complexos, por exemplo com ciclos, obrigam a que o meta-interpretador possua mecanismos bastante elaborados. Por este motivo a justificação em pós-processamento pode tornar-se um processo lento e pouco escalável.

De facto, gerar a justificação com esta técnica implica fazer a mesma operação duas vezes: primeiro descobrir as respostas e depois fazer o processo inverso para gerar a justificação. Métodos mais avançados conseguem fazer as duas tarefas simultaneamente sendo, dessa forma, bastante mais rápidos. Essa técnica é apresentada de seguida.

4.1.2 Justificação Online

Como já tínhamos referido, na justificação online os processos de justificação e avaliação duma consulta são feitos ao mesmo tempo. Pemmasani et al. propõem uma

transformação aos programas que nos permite implementar a justificação online de forma simples [16]. Vejamos a Figura 4.1.

```
:- table path/2.

path(X,Z):-edge(X,Y), path(Y,Z).
path(X,Z):-edge(X,Z).

edge(a,b).
edge(b,a).
```

```
:- table path/2.

path(X,Z):-edge(X,Y), path(Y,Z)
           store_evid(path(X,Y),[((edge(X,Y,true),[]),
             ((path(Y,Z),true),ref(path(Y,Z)))]).

path(X,Z):-edge(X,Z),
           store_evid(path(X,Z),[((edge(X,Z),true),[])]).

edge(a,b).
edge(b,a).
```

Figura 4.1: Programa base para o cálculo de todos os caminhos de um grafo (à esquerda) e o mesmo programa mas transformado para ser capaz de gerar justificações para esses caminhos (à direita).

O programa da esquerda define novamente o cálculo dos caminhos possíveis entre os vértices de um grafo e no lado direito temos o mesmo programa mas transformado por forma a permitir a geração de justificações. Como podemos ver a principal diferença entre os programas reside na inclusão de chamadas ao predicado *store_evid/2* no final de cada cláusula do *path/2*. Este predicado é responsável por armazenar as justificações e garantir que só é guardada uma para cada caminho, de forma a evitar um grande consumo de memória e possíveis ciclos infinitos.

Supondo que fazemos a consulta *path(a,a)*, referente ao grafo representado pelos dois factos *edge/2* da Figura 4.1, o programa originaria, no final, as seguintes justificações:

path(a,a) - [((edge(a,b),true),[]),((path(b,a),true),ref(path(b,a)))]

path(b,a) - [((edge(b,a),true,[])]

A partir destes resultados podemos concluir que para ir de a para a temos de utilizar primeiro o arco *(a, b)* e em seguida o tuplo *((path(b,a),true),ref(path(b,a))* indica que temos ainda de utilizar o caminho entre b e a que está guardado na referência *ref(path(b,a))*. Seguindo a referência podemos constatar que o caminho entre b e a implica percorrer o arco *(b,a)*.

4.1.3 Justificação usando Tabulação com Operadores de Modo

A justificação usando a declaração de predicados tabelados com operadores de modo [7] pode ser considerada como um sub-caso da justificação online porque é feita ao mesmo

tempo que a avaliação da consulta. Esta implementação tem vantagens em relação à que apresentamos anteriormente pois é bastante mais elegante e mais simples de implementar do que utilizando um predicado como o *store_evid/2*.

Se compararmos o programa não transformado da Figura 4.1 com o da Figura 4.2 podemos ver que as únicas modificações efectuadas, por este método, foram incluir mais um argumento no predicado tabelado e modificar a declaração deste no topo do programa. No exemplo anterior, o predicado *store_evid/2* era o responsável por fazer o controle da geração das justificações. Este controle passa agora a ser feito pelos operadores + e - que apenas permitem que uma resposta referente a cada vértice fique na tabela.

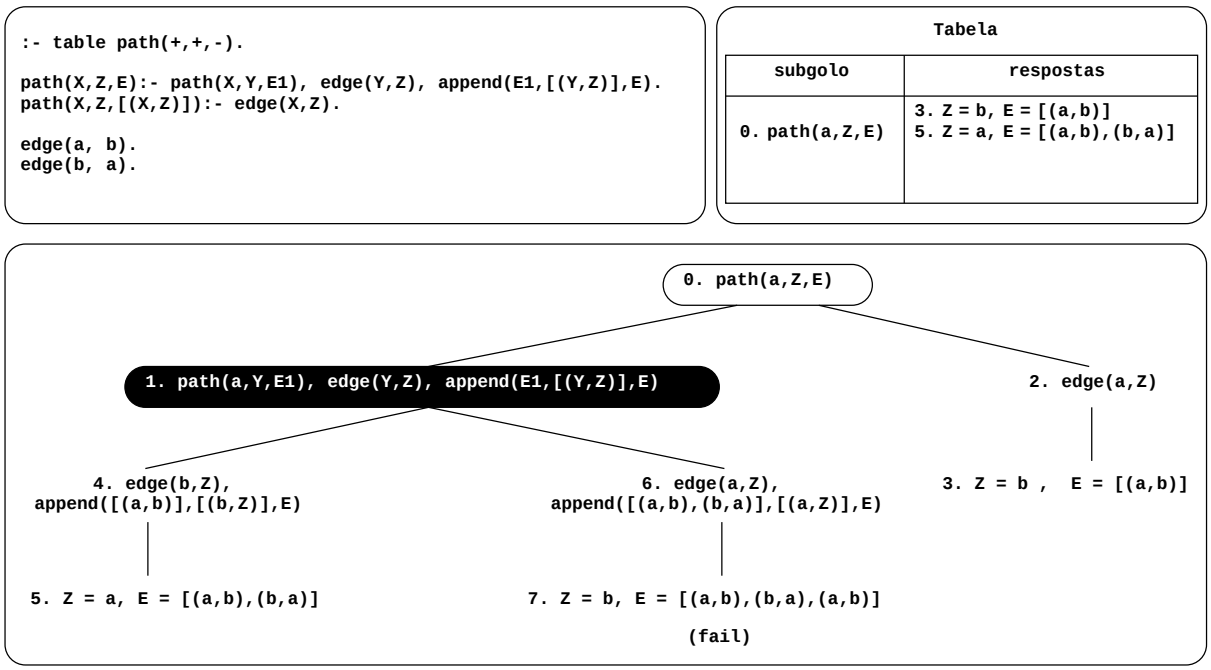

Figura 4.2: Exemplo da utilização do operador - na geração de justificações.

Vejamos a execução do programa representado na parte inferior da Figura 4.2. Se executarmos a consulta *path(a,Z,E)* (passo 0), ao ser seleccionada a primeira cláusula do programa, a computação é suspensa por dar origem a uma chamada repetida (passo 1). É então testada a segunda cláusula do predicado path/3 (passo 2), o que nos permite encontrar a resposta {*Z=b, E=[(a, b)]*} (passo 3), que inclui a justificação de que de *a* se chega a *b* usando o arco *(a,b)*. De seguida, reactivamos o nó 1 e ao consumirmos a resposta encontrada no passo anterior, iremos originar uma nova resposta (passo 5). Esta nova resposta indica-nos que de *a* podemos chegar a *a* utilizando os arcos *(a,b)* e *(b,a)*. Esta resposta será também consumida pelo nó 1 (passo 6), levando à descoberta da resposta {*Z=b, E=[(a,b),(b,a),(a,b)]*} (passo 7).

Consultada a tabela, a verificação da existência de uma variante só é feita sobre a variável Z (segundo argumento) uma vez que a variável E (terceiro argumento) é não indexada. Como já existe uma resposta referente ao vértice b, esta última é descartada. Desta forma conseguimos guardar apenas uma justificação para cada caminho encontrado e evitar ciclos infinitos.

4.2 Preferências

A Programação em Lógica é usada, muitas vezes, para resolver problemas de optimização. Normalmente quando escrevemos um algoritmo deste tipo, pode ser difícil definir uma solução óptima através de uma simples maximização ou minimização. O paradigma da Programação em Lógica por Preferências tenta solucionar este tipo de problemas dividindo a especificação do problema da definição de solução óptima. Esta abordagem surge como resultado do trabalho de Govindarajan et al. [6, 5] que concebeu um método simples, declarativo e eficiente para resolver problemas de optimização.

Para melhor percebermos a sua sintaxe vejamos o exemplo da Figura 4.3. O programa tem por objectivo calcular os caminhos de custo mínimo entre os vértices de um grafo. Caso exista mais do que um caminho com o mesmo custo, o desempate é feito seleccionando a resposta com a menor distância. A sintaxe desenvolvida por Govindarajan et al. divide o programa em duas partes: a primeira parte chamada *core* é constituída pelo programa em si (código na parte superior da Figura 4.3). As restantes cláusulas são chamadas cláusulas de preferência e são responsáveis por seleccionar as respostas óptimas para o problema (código na parte infererior da Figura 4.3). O símbolo $<$, utilizado nessas cláusulas, tem como significado *"é menos preferido do que"*, caso as condições à direita do símbolo :- se verifiquem.

```
path(X,Z,C,D):- edge(X,Z,C,D).
path(X,Z,C,D):- edge(X,Y,C1,D1), path(Y,Z,C2,D2), C is C1 + C2, D is D1 + D2.

edge(a,b,1,4).
edge(b,a,1,3).

path(X,Z,C1,D1) < path(X,Z,C2,D2):- C2<C1, !.
path(X,Z,C1,D1) < path(X,Z,C2,D2):- C2=C1, D2<D1.
```

Figura 4.3: Exemplo de um programa que utiliza preferências.

Conforme podemos ver, a sintaxe desenvolvida por Govindarajan et al. pode ser bastante útil na implementação de preferências, pois divide de uma forma simples e intuitiva a especificação das preferências do programa.

4.2.1 Preferências usando Tabulação com Operadores de Modo

Guo et al. propõem pequenas alterações ao nível da sintaxe desenvolvida por Govindarajan et al., bem como uma série de transformações de programa para que os programas possam ser interpretados recorrendo aos operadores de modo dos predicados tabelados [9, 10]. Passemos a considerar novamente o programa da Figura 4.3 para melhor percebermos as modificações introduzidas. Em relação à sintaxe, o programador precisa agora de incluir mais uma declaração do tipo *table path(+,+,<<<,<<<)* que indica que os dois primeiros argumentos são indexados e os seguintes, representados pelo símbolo <<<, vão estar sujeitos às cláusulas de preferência. As transformações feitas ao programa de modo a que este possa ser interpretado tal como pretendido são as seguintes:

- O primeiro operador <<< a surgir na declaração do predicado tabelado é substituído pelo operador *last* e os restantes operadores de preferência são substituídos pelo operador -, ou seja a declaração *table path(+,+,<<<,<<<)* é substituida por *table path(+,+,last,-)*.
- A cabeça dos predicados tabelados é modificada acrescentado a palavra New ao nome do predicado.

```
pathNew(X,Y,C,D):- edge(X,Z,C1,D1), path(Z,Y,C2,D2),
                   C is C1+C2, D is D1+D2.
pathNew(X,Y,C,D):- edge(X,Y,C,D).
```

- É criada uma nova cláusula com o nome do predicado tabelado, ficando no caso do nosso programa, com o seguinte aspecto:

```
path(X,Y,C,D):- pathNew(X,Y,C,D),
                (
                  path(X,Y,C1,D1)
                ->
                  path(X,Y,C1,D1) <<< path(X,Y,C,D)
                ;
                  true
                ).
```

A Figura 4.4 apresenta, no canto superior esquerdo, o programa completo depois de terem sido feitas as alterações descritas acima. Consideremos agora a primeira árvore

de execução que aparece na figura, onde efectuamos a consulta *path(a,b,C,D)* (passo 0). Na transformação de programa que realizamos, o predicado *path/4* apenas unifica com uma única cláusula (passo 1). De seguida, o subgolo *pathNew/4* da cláusula *path/4* é o responsável por descobrir um caminho entre os vértices *a* e *b*, vindo a encontrar um caminho com custo 1 e com distância 4 (passo 2). É, agora, altura de executar o *if then else* que tem por objectivo verificar se já existe alguma resposta na tabela referente àquele caminho através de uma chamada repetida a *path/4*. Caso exista alguma resposta, esta irá ser comparada com a resposta descoberta pelo *pathNew/4* usando as preferências definidas no programa. Neste caso a tabela ainda está vazia, por isso, a resposta é simplesmente considerada sem ser necessário executar nenhuma cláusula de preferência (passo 3).

Como ficaram hipóteses por considerar, após *backtraking* a execução volta ao passo 1. Desta vez a chamada a *pathNew/4* encontra um caminho entre os vértices *a* e *b* com custo 1 e com distância 3. Como já existe uma resposta na tabela, que tinha sido encontrada no passo 3, é altura de comparar ambas recorrendo às cláusulas de preferência (passo 5). A primeira cláusula de preferência especifica que a nova resposta é preferida caso tenha um custo menor do que a que estava na tabela, o que não se aplica neste caso uma vez que ambas têm custo 1. Por esse motivo é testada a segunda cláusula de preferência, que sucede, por a nova resposta ter um custo igual e uma distância menor do que a que estava na tabela. Consequentemente a chamada a *path(a,b,C,D)* sucede com a resposta *C=1, D=3* (passo 6). Como estamos a utilizar o operador last, que guarda apenas a última resposta referente aos argumentos indexados, a resposta encontrada anteriormente é substituída pela resposta encontrada agora no passo 6.

O programa que acabámos de ver apresenta no entanto algumas limitações pois quando efectuamos uma consulta, é necessário que todos os argumentos indexados estejam instanciados, como no exemplo *path(a,b,C,D)*. Na árvore de execução que se encontra na parte de baixo da Figura 4.4, podemos ver o que acontece quando fazemos uma consulta com todas as variáveis livres. No passo 1 o subgolo *pathNew/4* encontra, à semelhança do que aconteceu anteriormente, um caminho entre os vértices *a* e *b*. Seguidamente a tabela deveria ser consultada para obter as respostas referentes a este caminho (passo 2), contudo não é o que acontece. Se recordarmos aquilo que já vimos da tabulação, esta só verifica se há respostas na tabela caso se trate de uma chamada repetida. Neste caso, a chamada inicial foi *path(X,Z,C,D)* (passo 0) mas a chamada do passo 2 é *path(a,b,C1,D1)*. Como podemos verificar não se trata então da mesma chamada, como acontecia no exemplo anterior, o que leva a que a computação continue

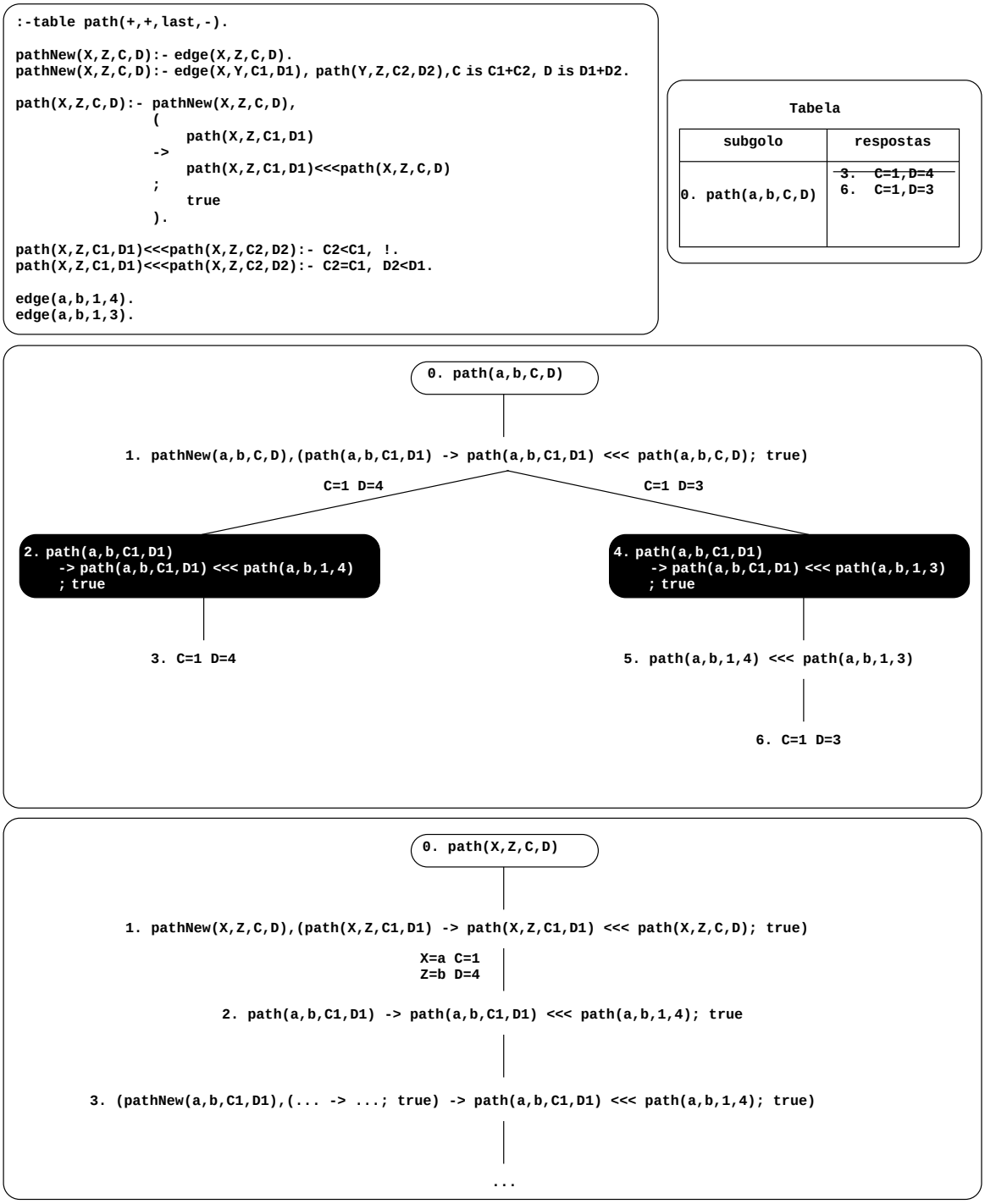

Figura 4.4: Exemplo da utilização do operador *last* na implementação de preferências para as consultas *path(a,b,C,D)* e *path(X,Z,C,D)*.

no passo 3 sem verificar se existem respostas na tabela. Esta limitação impede que sejam calculados os caminhos com os custos mínimos e as distâncias mínimas de e para todos os pontos do grafo.

Nesta tese propomos algumas alterações ao programa final, de forma a que seja possível realizar consultas em que as variáveis indexadas também estejam livres. Os dois primeiros pontos da transformação que vimos anteriormente são mantidos inalterados. As diferenças surgem no predicado *path/4* (terceira regra de transformação), nas cláusulas de preferências e na introdução de um novo predicado *pathPrefer/2*. O aspecto final do programa pode ser visto na Figura 4.5.

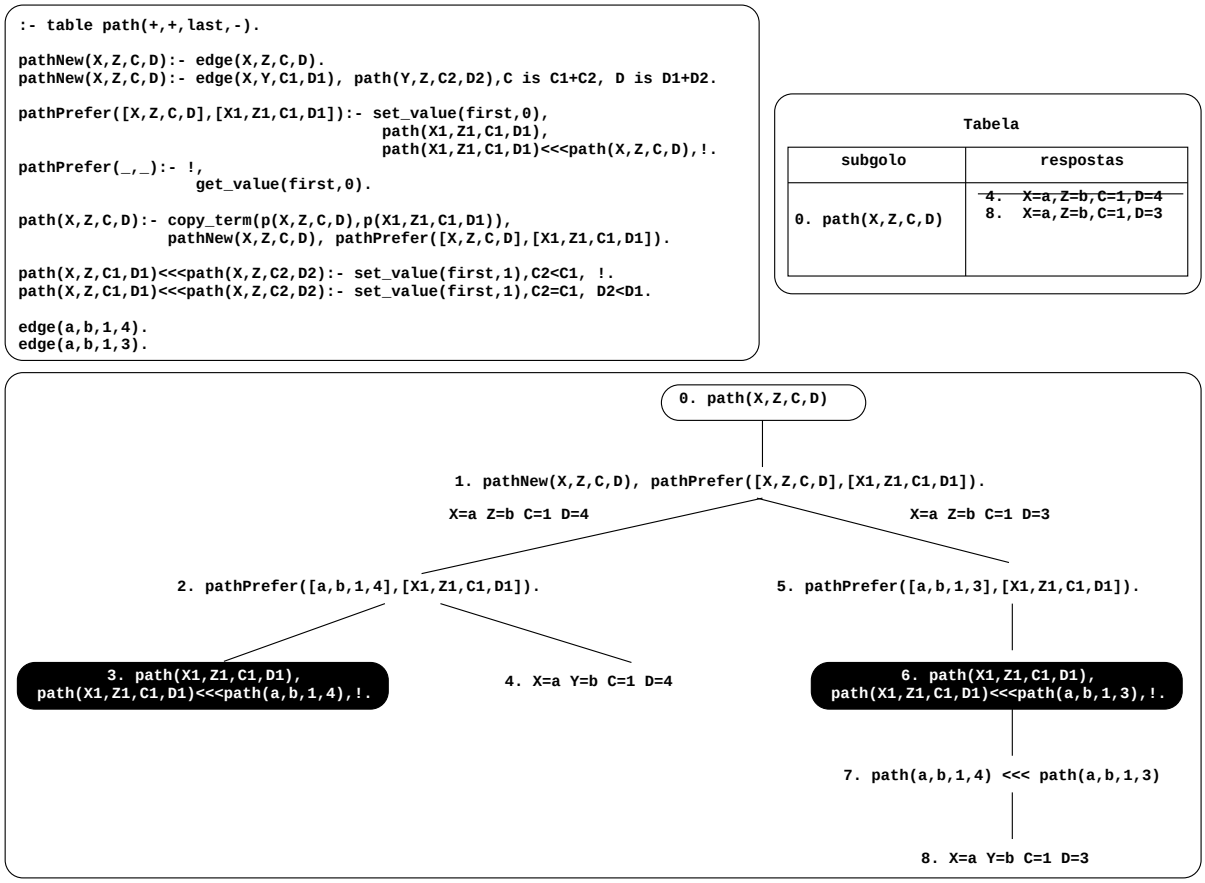

Figura 4.5: A nossa proposta de implementação de preferências em programas lógicos.

Para melhor percebermos as diferenças no funcionamento entre o programa anterior e o que propomos, vejamos a árvore de execução que se encontra na parte de baixo da Figura 4.5. Começamos por efectuar a consulta *path(X,Z,C,D)*. Esta unifica com a cláusula *path/4* onde são copiadas inicialmente as variáveis originais da chamada e, posteriormente, é chamado o subgolo *pathNew/4* com as variáveis originais. A

chamada ao subgolo origina a descoberta de um novo caminho entre os vértices *a* e *b* com custo 1 e distância 4. É agora altura de executar o predicado *pathPrefer/2* que recebe duas listas como argumentos: uma contendo o novo caminho descoberto e outra que contém as variáveis copiadas da chamada original (passo 2).

Na primeira cláusula do *pathPrefer/2* (passo 3), a variável global *first* é colocada a 0 (mais à frente perceberemos a utilidade deste passo) e depois é chamado o subgolo *path/4* com as variáveis da consulta do passo 0 que tinham sido copiadas. Esta estratégia permite evitar o problema com que nos deparamos no exemplo anterior. Desta forma, repetindo a chamada, a tabulação irá aceder à tabela e retirar uma por uma as respostas que lá se encontram. Neste caso, como não existe nenhuma resposta na tabela, a computação é suspensa neste ponto sendo testada a segunda cláusula do *pathPrefer/2*. Esta segunda cláusula começa por efectuar um corte, que tem como objectivo impedir que a suspenção da primeira cláusula seja reactivada, e em seguida é verificado se a variável *first* se encontra a 0. Note que a variável *first* é colocada a 1 somente nas cláusulas de preferência, o que significa que existem soluções na tabela. Como a primeira cláusula do *pathPrefer/2* realiza um corte no fim garante ainda que, ao estarmos a testar a segunda cláusula, é porque a primeira cláusula não chegou ao fim, o que pode ter ocorrido por duas situações. A primeira situação a considerar é a variável *first* estar a 0, o que significa que na tabela não existia nenhuma resposta compatível com os vértices que estamos a considerar, logo a nova resposta deve ser inserida. No caso da variável *first* ter valor 1 significa que havia pelo menos uma resposta compatível na tabela, contudo a nova resposta é pior do que a que lá estava, e por isso, não deve ser inserida.

No caso concreto do passo 4, a variável *first* está a 0 levando a que a resposta {*X=a, Z=b, C=1, D=4*} seja inserida na tabela. Como ainda há alternativas a considerar no subgolo *pathNew/4*, após *backtracking* é descoberto um novo caminho de *a* para *b* com custo 1 e com distância 3, e em seguida é chamado o predicado *pathPrefer/2* (passo 5). A primeira cláusula do predicado *pathPrefer/2* encontra uma resposta na tabela, que tinha sido descoberta no passo 4, e utilizando a segunda cláusula de preferência, conclui-se que a nova resposta é preferivel à anterior. O predicado sucede levando a que a nova resposta substitua a antiga na tabela (passo 8).

4.3 *Answer Subsumption*

A técnica da tabulação, como vimos, só considera uma resposta se na tabela não existir uma variante da mesma. Nesta tese já mostramos alguns operadores que nos permitem alterar este funcionamento de forma a dar ao programador mais controlo sobre esta operação permitindo-lhes, por exemplo, minimizar ou maximizar respostas ou implementar preferências, de modo a definir o seu próprio critério de optimização.

Em particular, as preferências podem ser vistas como que um sub-caso da técnica de *answer subsumption* da qual iremos falar a seguir. Esta técnica permite que uma nova resposta encontrada modifique um conjunto de respostas que já estavam na tabela em vez de estar limitada a uma só, como temos visto até agora. Apesar das grandes vantagens e do potencial que a técnica de *answer subsumption* possui, esta não tem sido objecto de grande investigação e, por essa razão não está presente na maior parte dos sistemas de Prolog com suporte para tabulação. De seguida iremos ver a abordagem do sistema XSB a este tipo de mecanismo e, mais à frente, apresentaremos a nossa proposta.

4.3.1 *Answer Subsumption* no XSB Prolog

O interpretador do XSB Prolog possui dois predicados, à disposição dos programadores, que implementam a técnica de *answer subsumption* [18]. Nesta secção abordamos as principais características de ambos. O predicado mais simples é o *filterReduce/4*. Na Figura 4.6 apresentamos, na parte de cima, o código da sua implementação e, na parte de baixo, um exemplo da sua utilização. O predicado *filterReduce/4* recebe como primeiro argumento a chamada juntamente com as variáveis indexadas sobre a qual será aplicada a *answer subsumption*, como segundo argumento recebe a função de preferência a ser aplicada, como terceiro argumento recebe a identidade e como quarto argumento a variável onde virá o resultado. Vejamos o seu funcionamento em mais detalhe no parágrafo que se segue.

Na Figura 4.6, as linhas 2 e 3 fazem chamadas a predicados internos do XSB que permitem ir buscar informações ao ponto de escolha corrente. A linha 4 faz uma cópia das variáveis da chamada, esta cópia serve para, mais tarde, aceder à tabela e consultar as respostas que lá se encontram. Os predicados das linhas 5 e 6 permitem separar as variáveis que são utilizadas para verificar se as respostas se tratam de uma variante das restantes. Na linha 7 é feita a chamada ao predicado sobre o qual será aplicada

```
1    filterReduce(Call,Op,Id,Res) :-
       '_$savecp'(Breg),
       breg_retskel(Breg,4,Skel,Cs),
       copy_term(Skel,Oskel),
5      get_opt_non_opt(Oskel,Osargs,Oopt),
       get_non_opt(Skel,Sargs),
       call(Call,Nvar),
       (
              get_returns(Cs,Oskel,Leaf),
10            variant(Sargs,Osargs)
       ->
              call(Op,Oopt,Nvar,Res),
              Res \== Oopt,
              delete_return(Cs,Leaf)
15     ;
              call(Op,Id,Nvar,Res)
       ).
```

```
pathNew(X,Y,C) :- path(X,Z,C1),edge(Z,Y,C2),C is C1 + C2.
pathNew(X,Y,C) :- edge(X,Y,C).


min(One,Two,Min):- One > Two -> Min = Two ; Min = One.


path(X,Y,C) :- filterReduce(pathNew(X,Y),min,infinity,C).
```

Figura 4.6: O predicado *filterReduce/4*: código da sua implementação no XSB Prolog (em cima) e exemplo da sua utilização (em baixo).

a *answer subsumption* para que esta encontre uma nova resposta. Seguidamente na linha 9 o predicado *get_returns/3* acede à tabela e retira, por *backtracking*, todas as respostas que lá se encontram até que uma suceda. Na linha seguinte é verificado se a resposta que estava na tabela é variante da nova que foi encontrada na linha 7. Em caso de sucesso das linhas anteriores (linhas 9 e 10) é testada na linha 12 a cláusula de preferência definida pelo utilizador sobre os elementos sujeitos à preferência. Na linha 13 é verificado se a resposta óptima é igual à que se encontrava na tabela, pois, nesse caso, não é necessário inseri-la. A linha 14 só é executada se a resposta nova for a preferida. Neste caso é necessário apagar a que estava na tabela para que esta a substitua.

Como se trata de um *if then else*, a condição da linha 16 só é executada caso alguma das condições das linhas 9 e 10 falhe, ou seja, caso não haja nenhuma resposta na tabela que seja variante da nova, sendo a resposta nova sujeita à cláusula de preferência juntamente com a identidade.

Para além do predicado *filterReduce/4*, o XSB possui um outro predicado de *answer subsumption*, o *bagReduce/4*. A principal diferença entre ambos os predicados prende-se com o facto do predicado *bagReduce/4* usar a unificação em vez da variância para verificar se as respostas são compatíveis. Para melhor percebermos as implicações práticas vejamos um exemplo: se estivéssemos a considerar as respostas {*X=1, Y=A*}

e {*X=B, Y=1*}, com o predicado *filterReduce/4* elas não seriam consideradas compatíveis ao passo que com o *bagReduce/4* elas são consideradas compatíveis.

O modo de utilização deste predicado é, contudo, muito semelhante ao anterior. Vejamos em detalhe o código da sua implementação juntamente com um exemplo da sua utilização na Figura 4.7. Tal como no *filterReduce/4*, as linhas 2 e 3 são chamadas a predicados internos do XSB que permitem extrair informação do ponto de escolha corrente. Em seguida, na linha 4, é feita a cópia dos termos da chamada original e é aqui que surgem as primeiras diferenças em relação ao *filterReduce/4*, pois esta cópia é feita de maneira a preservar os valores de algumas das variáveis que não estão livres. Esta técnica tem vantagens que serão explicadas mais à frente.

```
1    bagReduce(Call,Res,Op,Id) :-
       '_$savecp'(Breg),
       breg_retskel(Breg,4,Skel,Cs),
       copy_term(p(Call,Res,Skel),p(Call,Ovar,Oskel)),
5      Call(Nvar),
       (
            get_returns(Cs,Oskel,Leaf)
         ->
            Op(Ovar,Nvar,Res),
10          Res \== Ovar,
            delete_return(Cs,Leaf)
         ;
            Op(Id,Nvar,Res)
       ).
```

```
pathNew(X,Y,C) :- path(X,Z,C1),edge(Z,Y,C2),C is C1 + C2.
pathNew(X,Y,C) :- edge(X,Y,C).


min(One,Two,Min):- One > Two -> Min = Two ; Min = One.


path(X,Y,C) :- bagReduce(pathNew(X,Y),C,min,infinity).
```

Figura 4.7: O predicado *bagReduce/4*: código da sua implementação no XSB Prolog (em cima) e exemplo da sua utilização (em baixo).

Seguidamente, na linha 5, é feita a chamada sobre a qual iremos aplicar a *answer subsumption*, a fim de encontrar novas repostas. Embora a ideia seja a mesma, a forma como é feita a chamada é diferente da que vimos no predicado *filterReduce/4*. A chamada pode ser feita desta forma porque o predicado em causa está implementado recorrendo à sintaxe Hilog [25]. Esta sintaxe que o XSB implementa permite utilizar subgolos como se fossem funções de modo a que seja possível passar-lhes argumentos. Em particular, no exemplo em causa faz com que a chamada tradicional *path(A,B,C)* possa ser feita como *path(A,B)(C)*. Nas linhas 6 até à linha 14 está definido um *if then else*. A condição testada no *if* verifica se existem na tabela respostas compatíveis com a encontrada na linha 5. No predicado *filterReduce/4* após a chamada ao *get_returns/3* era necessário verificar se a resposta obtida era compatível (variante) com a nova.

Neste caso tal deixa de ser necessário. Isto deve-se ao facto da cópia que foi feita na linha 4 preservar o valor de alguns argumentos. Deste modo o predicado *get_returns/3* encarrega-se de fazer esse controlo. A partir deste ponto a implementação dos dois predicados é praticamente igual, havendo apenas uma pequena diferença na maneira como é chamada a função de preferência. No predicado *filterReduce/4* a chamada é feita na forma *call(Op,Id,Nvar,Res)* e no *bagReduce/4* é feita como *Op(Id,Nvar,Res)*. Esta diferença não tem implicações práticas tratando-se de uma questão de semântica pois, mais uma vez, está a ser usada a sintaxe Hilog.

Estes dois predicados têm, contudo, algumas limitações. Vejamos o exemplo da Figura 4.8. Supondo que os factos *edge/3* representam caminhos entre dois pontos e o seu respectivo custo, queremos encontrar todos os caminhos possíveis entre dois pontos ficando somente com os de menor custo. Para tal é necessário definir a função de preferência e fazer a chamada ao predicado *bagReduce/4*. Este predicado recebe como primeiro argumento o predicado sobre o qual será aplicada a *answer subsumption*, neste caso *edge(X,Y)*, em que X e Y são as variáveis sobre as quais é verificada a compatibilidade entre respostas (variáveis indexadas), e como restantes argumentos a variável onde virá o resultado optimizado, a função de preferência e a identidade.

```
edge(a,b,3).
edge(a,b,2).
edge(a,c,2).
edge(a,X,1).


min(One,Two,Min):- One > Two -> Min = Two ; Min = One.


path(X,Y,C) :- bagReduce(edge(X,Y),C,min,infinity).
```

Figura 4.8: Exemplo das limitações do predicado bagReduce/4.

Antes de olharmos para as respostas que o XSB daria, vamos tentar perceber quais as respostas esperadas. Os primeiros dois factos são referentes ao mesmo caminho pelo que na tabela ficará, por agora, a resposta $\{a,b,2\}$ por ter o menor custo. Seguidamente é considerado o facto *edge(a,c,2)* que, por não existir na tabela nenhuma resposta compatível, é inserido automaticamente. Por último falta considerar o facto genérico *edge(a,X,1)*. Este facto é compatível com ambas as respostas contidas na tabela uma vez que unifica com elas e tem um custo menor pelo que seria de esperar que as respostas anteriores, $\{a,b,2\}$ e $\{a,c,2\}$, fossem substituidas e alteradas para $\{a,b,1\}$ e $\{a,c,1\}$. Contudo não é isso que acontece. O XSB só altera a resposta $\{a,b,2\}$ para $\{a,b,1\}$ permanecendo a outra inalterada. Este comportamento tem origem na forma como o *bagReduce/4* está implementado, mais precisamente na parte do *if then else* (linhas 6 a 14 na Figura 4.7). Vejamos mais concretamente de que forma é

implementada esta instrução em Prolog. A sintaxe é:

A(X) -> B(X) ; C(X)

que pode ser lido como *"caso A(X) seja verdade executa B(X) senão executa C(X)"*. Na prática o Prolog implementa o *if then else* da seguinte forma:

P(X) :- A(X), !, B(X).

P(X) :- C(X).

O pormenor que gera o problema é o cut (!) que corta as hipóteses que ficam em aberto em A(X). Voltando ao *bagReduce/4*, quando a função *get_returns/3* devolve uma resposta que sucede só essa é que será alterada, sendo as restantes descartadas. Daí no nosso exemplo só ter sido alterada a resposta {*a,b,2*} para {*a,b,1*}, tendo a resposta {*a,c,2*} permanecido inalterada.

No início desta secção, referimos que um programa com *answer subsumption* deveria ser capaz de considerar várias respostas compatíveis que estivessem na tabela em vez de apenas uma, como acontecia com as preferências. No entanto, tal como acabamos de ver, nenhum dos dois predicados do XSB Prolog tem essa capacidade. No entanto, em relação às preferências, ambos os predicados do XSB Prolog, conseguem a partir de uma resposta nova e outra que esteja na tabela gerar uma terceira. Esta nova funcionalidade permite, por exemplo, implementar contadores que vão utilizando a nova resposta que é gerada como um acumulador guardando-a na tabela.

4.3.2 *Answer subsumption* usando o Operador *last*

A nossa implementação de *answer subsumption* tem duas diferenças fundamentais em relação às duas abordagens feitas pelo XSB. A primeira é utilizar o operador *last* para forçar a actualização das respostas nas tabelas e a segunda é a capacidade de considerar todas as respostas que são compatíveis e não apenas a primeira.

A ideia base é em tudo semelhante à que vimos para o XSB. O nosso predicado chama-se *filter/3*, e o seu código pode ser visto na Figura 4.9. O predicado *filter/3* recebe como argumentos a chamada sobre a qual será feita a *answer subsumption* juntamente com as variáveis indexadas, o nome do predicado de preferência e, por

último, a variável onde virá a solução óptima. Passemos agora a ver com algum detalhe a nossa implementação. Primeiro explicaremos o predicado *filter/3* e posteriormente os predicados auxiliares que podem ser vistos na Figura 4.11. Para melhor ilustrar o funcionamento desta ferramenta utilizaremos novamente o programa que tem por objectivo encontrar os caminhos mínimos entre os diversos pontos do grafo.

```
1   filter(Call,Op,Result):-
     Call=..[Pred|ArgsList],
     copy_term(ArgsList,CopyArgsList),
     copy_term(Result,CopyResult),
5    copy_term(Result,ExtendedResult),
     call(Call,ExtendedResult),
     NewCall=..[Pred|CopyArgsList],
     filter_findall(filter(NewCall,Op,CopyResult),ArgsList,CopyArgsList,CopyResult,AnswersList),
     (
10     AnswersList \= []
     ->
       filter_check_insert_update([ExtendedResult|ArgsList],AnswersList,ExtendedResult,Op,Result),
     ;
       Result=ExtendedResult
15   ).
```

```
pathNew(X,Y,C) :- edge(X,Y,C).
pathNew(X,Y,C) :- path(X,Z,C1),edge(Z,Y,C2),C is C1 + C2.

min(One,Two,Min):- Two < One, Min=Two.

path(X,Y,C) :- filter(pathNew(X,Y),min,C).
```

Figura 4.9: Código da implementação do predicado *filter/3* e um exemplo da sua utilização.

Passemos agora a ver passo a passo o funcionamento do *filter/3*. Para isso iremos simular o que aconteceria caso tivesse sido feita a chamada *filter(pathNew(1,Y),min,C)*. Primeiramente é utilizado o operador *=..* do Prolog que separa o nome do predicado da chamada das variáveis indexadas. No nosso exemplo isso corresponderia a dividir a chamada *pathNew(1,Y)* em *pathNew* e na lista *[1,Y]*. A seguir são feitos três *copy_term/2*. A utilidade destas cópias será explicada mais à frente. Na linha 6, recorrendo ao predicado *built-in call/2*, é feita a chamada ao predicado *pathNew/3*. Utilizando o exemplo que temos vindo a considerar a chamada seria *call(pathNew(1,Y),ExtendedResult)*.

A chamada ao predicado *pathNew/3* origina a descoberta de um novo caminho. Consideremos, por exemplo, que foi descoberto o caminho $\{Y = 2,\ ExtendedResult = 4\}$. Na linha 7, recorrendo à cópia que fizemos na linha 2, é recriado o estado inicial da variável *Call* que no nosso exemplo era *pathNew(1,Y)*. Este passo é fundamental para que o predicado *filter_findall/5* da linha 8 funcione. O *filter_findall/5* tem por objectivo ir buscar à tabela todas as respostas que unifiquem, nos elementos indexados, com a nova resposta que foi encontrada na linha 6. Torna-se então necessário recriar a chamada

original do *filter/3* uma vez que sem esta não podemos saber a que sub-golo da tabela devemos aceder para ir buscar as respostas sobre as quais devemos testar a unificação. Os argumentos que são passados são então a réplica da chamada original do *filter/3* - neste caso *filter(pathNew(1,CopyY),min,CopyResult)* - , a lista de variáveis originais *[1,2]*, a cópia da lista de variáveis *[1,CopyY]*, o *CopyResult* que neste momento é uma variável livre e outra variável livre, *AnswerList*, onde obteremos a lista de respostas.

O predicado *filter_findall/5* tem grande importância na implementação do nosso programa. Para melhor percebermos o seu funcionamento vejamos que tipo de respostas poderiam ser retornadas para o exemplo que temos vindo a considerar. Consideremos, por exemplo, as respostas *(1,Y,4)*, *(X,2,3)*, *(1,2,2)* e *(X,Y,7)*. Tal como podemos ver todas as respostas unificam com os elementos indexados da resposta *(1,2,4)* obtida na linha 6, ou seja, com os elementos 1 e 2.

Quando vimos a forma como o XSB implementava os seus predicados de *answer subsumption* verificámos que as suas limitações advinham do facto de o *if then else* ser efectuado sobre o predicado *get_returns/3*, o equivalente ao nosso *filter_findall/5*. Por esse motivo optamos por guardar todos os resultados numa lista passando o *if* a verificar se esta tem elementos ou não (linha 10). O caso mais simples é quando a lista se encontra vazia. Nesse caso a resposta encontrada na linha 6 é por definição a resposta mínima, sendo por isso inserida na tabela directamente (linha 14). No caso de haver repostas na lista, como no exemplo que estamos a considerar, passa a ser chamado o predicado *filter_check_insert_update/5* (linha 12). Este predicado recebe como argumentos uma lista que contem a nova resposta encontrada na linha 6, outra lista contendo as respostas retornadas pelo predicado *filter_findall/5*, o elemento não indexado da nova resposta encontrada, o nome do predicado de preferência e uma variável livre onde virão as respostas que unificam. Se reparamos com atenção, o elemento não indexado da chamada aparece duas vezes, sendo que a primeira é na lista que contém a reposta encontrada no passo 8 e a segunda será útil para guardar a melhor resposta para um caso específico que veremos mais à frente. O predicado *filter_check_insert_update/5* tem como função comparar a nova solução encontrada com as que encontram na lista de respostas produzida pelo predicado *filter_findall/5* recorrendo para isso às cláusulas de preferência definidas pelo utilizador. Nos parágrafos que se seguem iremos explorar as multiplicidades de casos com que este predicado se pode deparar. A Figura 4.10 apresenta alguns desses casos.

A Figura 4.10 apresenta do lado esquerdo a resposta que vamos considerar, no centro a lista de respostas retornadas pelo predicado *filter_findall/5* e no lado direito o aspecto da tabela face à nova resposta. No passo 0 da figura é apresentado um

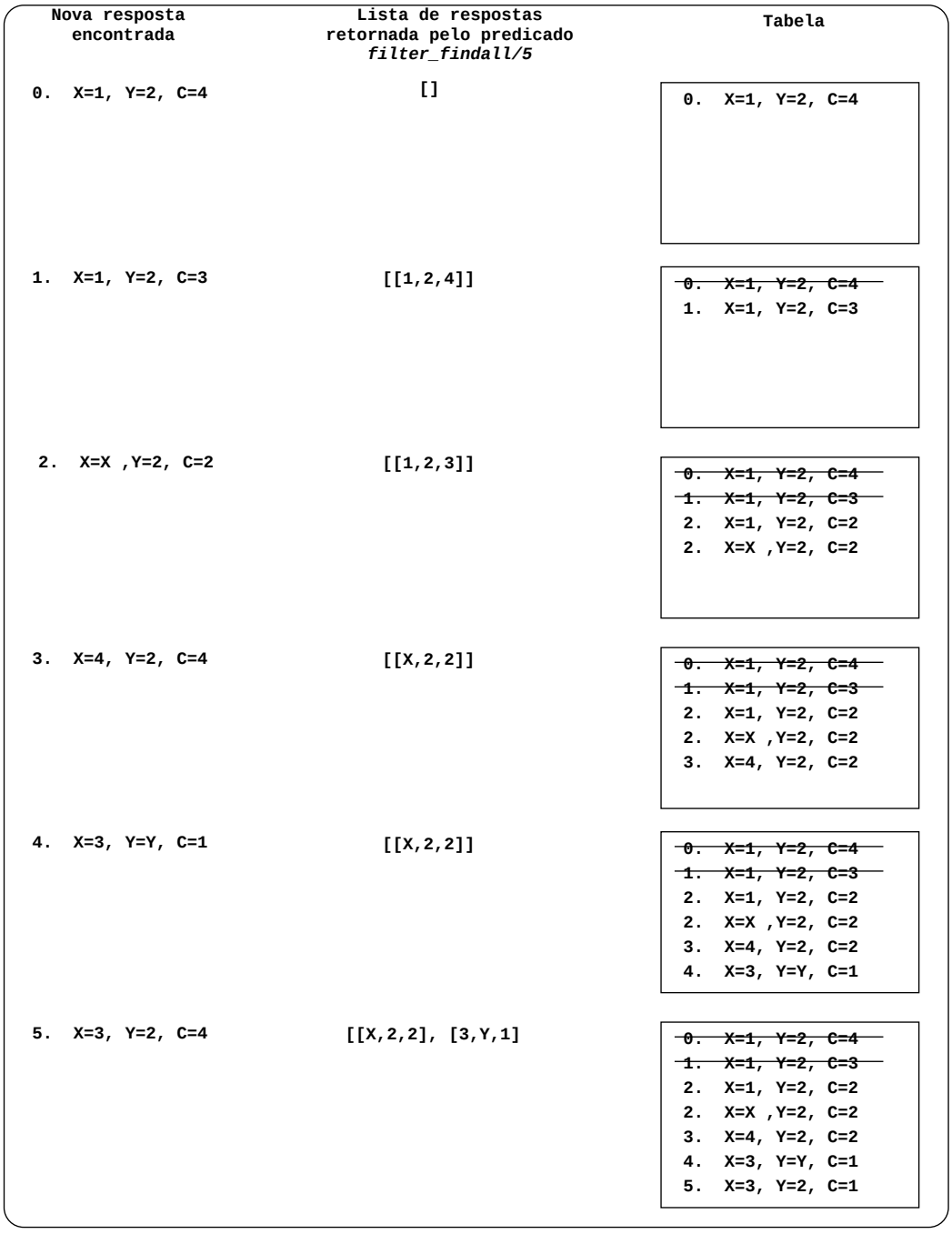

Figura 4.10: Exemplos da inserção de respostas pelo predicado *filter_check_insert_update/5*.

caso bastante simples. Não há respostas na tabela que unifiquem com a resposta *(1,2,4)* pelo que a resposta é inserida directamente sem passar pelo predicado *filter_check_insert_update/5*. Seguidamente é considerada, no passo 1 da figura, a resposta (1,2,3). A resposta do passo anterior unifica com esta resposta sendo por isso contemplada na lista de respostas. Utilizando as cláusulas de preferência chegamos à conclusão que a resposta *(1,2,4)* que está na tabela é pior que a que foi encontrada neste passo. Isso origina a que a resposta *(1,2,3)* seja inserida na tabela substituindo a anterior. Um caso um pouco mais complicado é apresentado no passo 2. A resposta encontrada *(X,2,2)* tem uma variável livre: a variável *X*. Na lista de respostas temos a resposta *(1,2,3)* que por ter um custo maior terá de ser actualizada para (1,2,2). A resposta (X,2,2) também terá de ser inserida na tabela uma vez que poderá vir a ser útil mais à frente durante a execução do programa. O caso inverso deste é apresentado no passo 3 da figura. Agora a resposta que contém variáveis está na tabela. A resposta encontrada *(4,2,4)* será modificada para *(4,2,2)* por aplicação da resposta *(X,2,2)*.

No quarto passo apresentamos um caso em que as respostas unificam mas não podem ser consideradas compatíveis. Encontramos a resposta *(3,Y,1)* que unifica com a resposta *(X,2,2)* que está na tabela. No entanto, o predicado *filter_check_insert_update/5* não as deve considerar compatíveis. Supondo que eram consideradas compatíveis, a tabela iria ficar com as respostas *(3,Y,1)* e a resposta *(X,2,2)* seria alterada para *(X,2,1)*. Se mais à frente durante a computação surgisse a resposta *(4,2,1)* ela seria alterada para *(4,2,1)*. Mas olhando para as respostas iniciais não podemos concluir isso, uma vez que a resposta *(3,Y,1)* não unifica com *(1,2,4)*. Por esse motivo o nosso programa tem de ser capaz de impedir que estas respostas sejam consideradas compatíveis e proceder somente à inserção da resposta *(3,Y,1)*.

Nos casos que estivemos a considerar até agora só existia no máximo uma resposta na tabela que unificasse com a nova. No quinto passo apresentamos um caso em que isso não acontece. Se fosse agora descoberta a resposta *(3,2,4)* teríamos que considerar as respostas *(X,2,2)* e *(3,Y,1)*. Nestes casos é necessário ter algum cuidado pois a aplicação das duas respostas que unificam iria gerar respectivamente as respostas *(3,2,2)* e *(3,2,1)*. Ambas são melhores do que a nova que encontrámos mas, como seria de esperar, não devem ser as duas inseridas na tabela. Por esse motivo o nosso programa tem de ser capaz de inserir somente a melhor resposta, ou seja, a resposta *(3,2,1)*.

Agora que já vimos em pormenor a implementação do predicado *filter/3* vejamos a implementação dos predicados auxiliares *filter_findall/5* e *filter_check_insert_update/5* presentes na Figura 4.11. O predicado *filter_findall/5*, tal como referimos anterior-

mente, é responsável por aceder à tabela e retornar o conjunto das respostas que unificam com a nova resposta encontrada. O predicado *filter_findall/5* utiliza o predicado *built-in findall/3* aplicado ao predicado *filter_find/3*. O *findall/3* permite-nos encontrar todas as soluções para uma determinada chamada. Neste caso optámos por fazer sobre o predicado *filter_find/3*, em vez de fazer sobre o predicado *filter/3* com o objectivo de melhorar o desempenho do programa. A função *filter_find/3* irá fazer uma chamada repetida ao predicado *filter/3* que, por ser um predicado tabelado retorna as respostas que estão na tabela. De seguida com recurso ao predicado *copy_term/2* fazemos um teste que nos permite verificar se a nova resposta e a que está na tabela unificam. O facto de este teste ser feito aqui permite que a quantidade de soluções retornadas seja muito menor, uma vez que só as respostas que unifiquem com a nova resposta encontrada vão ser retornadas. Quanto menor for a lista de retorno, menos operações de acesso serão feitas poupando acessos a memória e conseguindo um melhor desempenho.

Por fim, falta-nos ver a implementação do predicado *filter_check_insert_update/5*. Tal como vimos anteriormente, este predicado tem como função comparar a nova solução encontrada com as que estão na lista que o predicado *filter_findall/5* produziu. O predicado *filter_check_insert_update/5* é constituído por 4 cláusulas. Iremos começar por explicar a segunda por nos permitir um melhor encadeamento das ideias que estão na base deste predicado. Esta cláusula começa por fazer uso do predicado auxiliar *is_more_generic/2* com o intuito de perceber se a nova resposta encontrada é mais genérica, no que respeita aos argumentos indexados, do que a que está à cabeça da lista. O exemplo 2 da Figura 4.10 representa essa situação. A resposta encontrada é considerada mais genérica do que a que estava na tabela por ter variáveis livres e a outra não. Este predicado também sucede se ambas as respostas forem iguais como por exemplo, no caso dos elementos indexados *(1,2)* e *(1,2)* ou *(1,X)* e *(1,Y)*. Se o predicado *is_more_generic/2* sucede é aplicado um corte (!) na computação para evitar que as outras cláusulas também sejam testadas para este caso concreto. Depois é necessário perceber por que motivo o predicado *is_more_generic/2* sucedeu, e para isso, recorremos ao predicado *is_equal/2*. Este predicado permite-nos saber se as respostas são realmente iguais. Caso sejam é verificado, recorrendo às cláusulas de preferência, se a nova resposta encontrada é melhor do que a que estava na tabela. Se for, a computação continua recursivamente sobre as restantes respostas da lista produzida pelo predicado *filter_findall/5* senão a computação falha, uma vez que deixa de ser necessário percorrer o resto da lista. Os procedimentos que estivemos a ver para o caso em que as respostas são iguais ainda escondem um detalhe

```
filter_findall(Filter,ArgsList,CopyArgsList,CopyResult,AnswerList):-
         findall([CopyResult|CopyArgsList],filter_find(Filter,ArgsList,CopyArgsList),AnswerList),
         !.


filter_find(Filter,ArgsList,CopyArgsList):-
         call(Filter),
         copy_term(CopyArgsList,ArgsList).


filter_check_insert_update(_,[],BestResult,_,BestResult) :- !.

filter_check_insert_update([ExtendedResult|ArgsList],[[TableResult|TableArgsList]|TailAnswersList],
                           BestResult,Op,Result):-
         is_more_generic(ArgsList,TableArgsList),
         !,
         (
           is_equal(ArgsList,TableArgsList),
           !,
           call(Op,TableResult,ExtendedResult,NewBestResult),
           filter_check_insert_update([ExtendedResult|ArgsList],TailAnswersList,NewBestResult,Op,Result)
         ;
           call(Op,TableResult,ExtendedResult,Result),
           ArgsList=TableArgsList,
         ;
           filter_check_insert_update([ExtendedResult|ArgsList],TailAnswersList,BestResult,Op,Result)
         ).

filter_check_insert_update([ExtendedResult|ArgsList],[[TableResult|TableArgsList]|TailAnswersList],
                           BestResult,Op,Result):-
         is_more_generic(TableArgsList,ArgsList),
         !,
         update_best_result(BestResult,TableResult,Op,NewBestResult),
         filter_check_insert_update([ExtendedResult|ArgsList],TailAnswersList,NewBestResult,Op,Result).

filter_check_insert_update(NewAnswer,[_|TailAnswersList],BestResult,Op,Result):-
         filter_check_insert_update(NewAnswer,TailAnswersList,BestResult,Op,Result).


is_equal(A,B):-
         \+ \+
         (
         numbervars(A,0,_),
         numbervars(B,0,_),
         A=B
         ).


is_more_generic(A,B):-
         \+ \+
         (
         numbervars(B,0,_),
         A=B
         ).


update_best_result(BestResult,TableResult,Op,NewBestResult):-
        (
           call(Op,BestResult,TableResult,NewBestResult)
        ->
           true
        ;
           NewBestResult = BestResult
        ).
```

Figura 4.11: Predicados auxiliares da implementação do predicado *filter/3*.

importante. É nesta zona do código que, caso o programa necessite, é criada um terceira resposta a partir da nova e da que estava na tabela. Para isso é passado à cláusula de preferência a variável *NewBestResult*. Esta variável guardará o resultado da aplicação da cláusula de preferência às duas respostas. Posteriormente quando o predicado *filter_check_insert_update/5* for chamado recursivamente ser-lhe-á passado este resultado. Sendo mais tarde inserida a resposta através da primeira cláusula do *filter_check_insert_update/5*.

Outra alternativa é o predicado *is_equal/2* falhar. Neste caso a resposta encontrada é mesmo mais genérica, e por isso, caso seja melhor, deve alterar a resposta que está na tabela. Para além disso a resposta mais genérica que foi encontrada também deve ser inserida. Contudo esta inserção não é efectuada nesta cláusula mas sim na primeira cláusula deste predicado que será usada quando a lista chegar ao fim. Finalmente, a terceira alternativa da segunda cláusula chama-se recursivamente a fim de garantir que os restantes elementos da lista também são avaliados.

A terceira cláusula do predicado *filter_check_insert_update/5* verifica se a resposta que está na tabela é mais genérica do que a que foi encontrada. O exemplo 3 da Figura 4.10 exemplifica esta situação. A seguir a essa verificação é feito um corte (!) para impedir que o caso que estamos a considerar seja considerado pela restante cláusula deste predicado. Posteriormente é utilizado o predicado auxiliar *update_best_result/5*. A função deste predicado auxiliar é guardar (no argumento que guarda a melhor resposta e que é passado recursivamente) a melhor resposta e mais genérica para que esta seja aplicada quando já não houver mais respostas que unifiquem a considerar. Esta cláusula também chama recursivamente o predicado *filter_check_insert_update/5* à semelhança do que vimos anteriormente. A quarta cláusula permite lidar com o caso que vimos no quarto exemplo da Figura 4.10. A única coisa que esta cláusula faz é ignorar o elemento à cabeça da lista voltando a chamar-se recursivamente.

4.3.3 Outros Exemplos Práticos

Agora que discutimos em detalhe a implementação do predicado *filter/3* é altura de vermos mais alguns exemplos da sua utilidade prática. A Figura 4.12 apresenta outros quatro programas que recorrem ao nosso predicado para implementar *answer subsumption*. Para melhor compreendermos as suas funcionalidades utilizaremos como exemplo os grafos presentes na Figura 4.13.

No primeiro exemplo, o programa tem como objectivo calcular o custo e o número total

```
pathNew(X,Z,[C,1]):- edge(X,Z,C).
pathNew(X,Z,[C,N]):- edge(X,Y,C1), path(Y,Z,[C2,N]), C is C1*N+C2.


path(X,Z,C):- filter(pathNew(X,Z),sum,C).


sum([OldC,OldN],[NewC,NewN],[C,N]):- C is OldC+NewC, N is OldN+NewN.
```

```
pathNew(X,Z,[C,1]):- edge(X,Z,C).
pathNew(X,Z,[C,N]):- edge(X,Y,C1), path(Y,Z,[C2,N1]), C is C1+C2, N is N1+1.


path(X,Z,C):- filter(pathNew(X,Z),min,C).


min([OldC,OldN],[NewC,NewN],[NewC,NewN]):- NewC < OldC.
min([OldC,OldN],[NewC,NewN],[NewC,NewN]):- NewC = OldC, NewN < OldN.
```

```
pathNew(X,Z,[C,1,[X,Z]]):- edge(X,Z,C).
pathNew(X,Z,[C,N,L]):- edge(X,Y,C1), path(Y,Z,[C2,N1,L1]), C is C1+C2,
                       N is N1+1, append([X],L1,L).


path(X,Z,C):- filter(pathNew(X,Z),min,C).


min([OldC,OldN,_],[NewC,NewN,L],[NewC,NewN,L]):- NewC < OldC.
min([OldC,OldN,_],[NewC,NewN,L],[NewC,NewN,L]):- NewC = OldC, NewN < OldN.
```

```
pathNew(X,Z,[C,1,[X,Z],[X,Z]]) :- edge(X,Z,C).
pathNew(X,Z,[C,N,L,L]) :- edge(X,Y,C1),path(Y,Z,[C2,N1,L1,_]),C is C1+C2,
                          N is N1 + 1, append([X],L1,L).


path(X,Z,C):- filter(pathNew(X,Z),min,C).


min([OldC,OldN,OldL1,OldL2],[NewC,NewN,NewL1,NewL2],[C,N,L1,L2]):-
    (NewC < OldC -> C = NewC, L1= NewL1, IsOld=1 ; C = OldC, L1 = OldL1),
    (NewN < OldN -> N = NewN, L2= NewL2; N = OldN, L2 = OldL2, IsOld==1).
```

Figura 4.12: Exemplos da utilização do predicado *filter/3*.

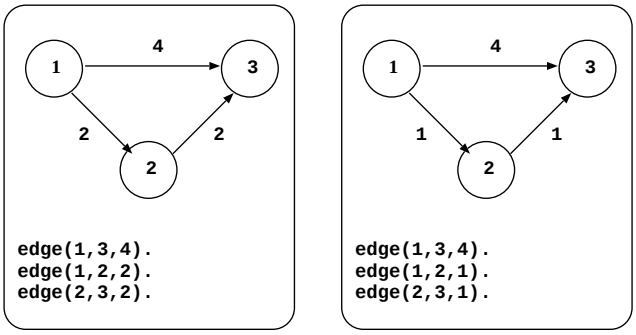

Figura 4.13: Grafos auxiliares para os exemplos da Figura 4.13.

de caminhos entre dois vértices de um grafo. Este programa utiliza a funcionalidade que permite produzir e combinar respostas a partir de duas outras respostas. A nossa função de preferência é responsável por somar o custo e o número de caminhos de cada resposta nova com os respectivos valores das respostas da tabela. Supondo que aplicávamos o nosso programa ao grafo do lado esquerdo da Figura 4.13 e que efectuávamos a consulta *path(1,3,C)* obteríamos a seguinte resposta: *C = [8,2]*. Isto significa que existem dois caminhos entre 1 e 3, o primeiro utilizando o caminho directo entre 1 e 3 e um outro utilizando o caminho entre 1 e 2 e entre 2 e 3. Ambos os caminhos têm custo 4 daí na resposta aparecer 8 que é a soma dos dois custos.

O segundo programa exemplo é bastante parecido com o que consideramos para explicar o funcionamento do predicado *filter/3* e que permite seleccionar o caminho com menor custo. Neste caso, se o programa encontrar dois caminhos com o mesmo custo, selecciona aquele que tem menor distância. Por esse motivo são necessárias duas cláusulas de preferência: a primeira selecciona a resposta com menor custo e em caso de empate a segunda cláusula seleccionará a que tem menor distância. Voltando a recorrer ao grafo do lado esquerdo da Figura 4.13 e efectuando a consulta *path(1,3,C)* obtemos a seguinte resposta *C=[4,1]*. Conforme tínhamos visto para chegar de 1 a 3 existem dois caminhos com custo quatro sendo que a menor distância é 1. Conforme podemos ver o nosso programa foi capaz de seleccionar a resposta com menor distância preferindo a resposta com distância 1 em vez da que tem distância 2.

O terceiro programa exemplo é semelhante ao que acabamos de ver, uma vez que a resposta é seleccionada com base no menor custo e o desempate é feito igualmente com base na distância. Para além disso também queremos guardar uma justificação representada pela lista L. Para isso, a lista L é recebida como argumento no predicado de preferência uma vez que é um elemento não indexado. Voltando a executar a mesma consulta e usando o grafo do lado esquerdo da figura obteríamos a resposta

C = [4,1,[1,3]]. A resposta, como seria de esperar, é idêntica à que obtivemos no programa anterior, sendo o novo elemento a lista que apresenta uma justificação para o resultado.

O quarto e último exemplo é um misto dos dois anteriores. Neste caso queremos que a resposta que fique na tabela reúna o menor custo e a menor distância numa só resposta. Para além disso deverá guardar a justificação tanto para o custo como para a distância. Este programa também fará uso da funcionalidade que nos permite gerar uma terceira resposta a partir de duas respostas anteriores. Para exemplificar o funcionamento deste programa iremos usar o grafo do lado direito da figura. Executando a consulta que temos vindo a efectuar para estes exemplos chegávamos ao resultado *C = [2,1,[1,2,3],[1,3]].* A resposta indica que o caminho com menor custo é percorrido usando o caminho de 1 para 2 e depois de 2 para 3. Ao passo que o caminho com menor distância é atingido utilizando o caminho directo de 1 para 3. As cláusulas de preferência deste nosso programa apresentam um pormenor bastante importante. Se observarmos com atenção a cláusula de preferência neste exemplo podemos ver que existe mais uma variável no corpo da cláusula. Esta variável permite evitar que, caso a resposta gerada seja igual à que está na tabela, esta seja inserida. Caso tal não fosse evitado, o programa poderia entrar em ciclo infinito uma vez que a cláusula estaria a ser sucessivamente inserida e considerada. Este problema advém da forma como está implementado o operador *last*. Sempre que uma resposta sucede, este substitui imediatamente a anterior que tenha as mesmas variáveis indexadas. Portanto se estiver sempre a inserir a mesma resposta ela irá substituir-se indefinidamente a ela própria.

4.4 Resumo

Neste capítulo demonstrámos a utilidade prática dos operadores de modo que apresentamos nas secções anteriores. Vimos que os diversos operadores podem ser utilizados para implementar mecanismos de Justificação, Preferências e de *Answer Subsumption*.

Capítulo 5

Implementação

Neste capítulo abordamos a implementação dos operadores de modo dos predicados tabelados e discutimos em pormenor as modificações feitas ao sistema YapTab de forma a suportar este novo mecanismo.

5.1 Declaração de Predicados Tabelados

Nos capítulos anteriores vimos uma nova forma de declarar predicados tabelados que nos permite ter um controlo mais refinado sobre a inserção de respostas na tabela. Para conseguirmos utilizar este mecanismo vimos que era necessário declarar os predicados tabelados da seguinte forma:

:- table $p(a_1,...,a_n)$.

em que p é o functor e os argumentos $a_1,...,a_n$ são os operadores +, -, @, *min*, *max* ou *last* que apresentamos nos capítulos anteriores. Para que o YapTab seja capaz de receber estes argumentos foi necessário proceder a algumas modificações na forma como a declaração *:- table* é tratada. Com estas alterações, o YapTab passa a receber o nome e a aridade do predicado tabelado juntamente com a lista contendo os operadores definidos pelo utilizador para o predicado. Com esta informação é construído um vector de modos que guarda em cada posição a informação sobre o operador de modo e o índice do argumento a ele associado. No entanto, nesse vector, os argumentos podem aparecer numa ordem distinta daquela que o utilizador definiu aquando da declaração do predicado tabelado. A alteração na ordem pela qual aparecem os argumentos é

propositada e apresenta vantagens que serão explicadas mais à frente neste capítulo. A ordem pela qual os operadores são inseridos no vector de modos é a seguinte:

- operadores de indexação (representado pelo símbolo +);
- operadores de agregação *max* e *min*;
- operador de agregação @ (operador que nos permite guardar todas as respostas para aquele argumento);
- operador *last*;
- operador de argumento não indexado (representado pelo símbolo -)

Vejamos um exemplo prático para melhor percebermos de que forma é construído o vector de modos. Na Figura 5.1 podemos ver o vector construído para o predicado tabelado *p/4* com a declaração *p(-,min,+,+)*. Tal como seria de esperar, o vector tem tantas posições como a aridade do predicado tabelado. O operador +, que tem maior precedência pela nossa definição, ocorre duas vezes na declaração do predicado e é por isso colocado nas duas primeiras posições do vector juntamente com os índices onde ocorre, posições 2 e 3. O operador que devemos considerar a seguir é o operador *min* presente na posição 1 da declaração. Por fim, colocamos no vector o operador -, de menor precedência e que aparece na posição 0 da declaração.

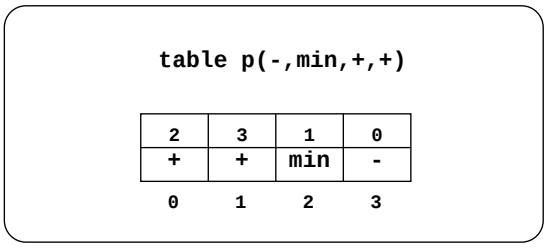

Figura 5.1: Vector de modos construído para a declaração *p(-,min,+,+)*.

Para além da construção deste vector, foi ainda necessário adicionar à estrutura de dados *table entry* o apontador respectivo para que mais tarde fosse possível aceder a esta informação. Na próxima secção iremos abordar o papel deste vector na inserção de subgolos nas tabelas.

5.2 Inserção de Subgolos nas Tabelas

Durante a execução de um programa tabelado, quando encontramos subgolos tabelados é necessário inseri-los na *trie de subgolos* respectiva, caso se trate da sua primeira ocorrência. Utilizando o novo sistema de declaração de predicados tabelados o processo é bastante semelhante. As diferenças ocorrem na ordem pela qual os argumentos são inseridos na *trie*. Na Figura 5.2 podemos ver na parte esquerda a representação de uma *trie* com os elementos inseridos pela ordem de inserção normal e do lado direito a mesma *trie*, mas com os elementos inseridos pela ordem representada no vector de modos da Figura 5.1.

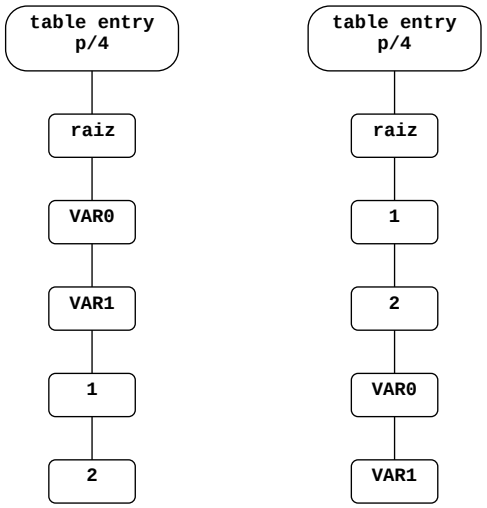

Figura 5.2: Aspecto de duas *tries* para o subgolo *p(A,B,1,2)*. A *trie* do lado esquerdo foi construída inserindo os elementos pela ordem que aparecem no subgolo enquanto que a do lado direito foi construída pela ordem do vector de modos da Figura 5.1.

O subgolo que estamos a considerar é o *p(A,B,1,2)*. No caso tradicional, os elementos são inseridos na *trie* pela ordem em que aparecem no subgolo. No novo sistema são inseridos pela ordem em que aparecem no vector de modos. Para tal, começamos por consultar a posição zero do vector de modo a obter a indicação do índice desse argumento no subgolo, neste caso índice 2 que corresponde à constante 1. Percorrendo as restantes posições do vector inserimos os restantes argumentos pela seguinte ordem: a constante 2, a variável B e por fim a variável A. Neste processo a informação do operador de modo não tem qualquer utilidade.

Depois da inserção estar concluída é construído um novo vector que mantém a mesma ordem dos argumentos do subgolo do vector de modos. Este novo vector continua a guardar a informação sobre o operador a utilizar mas substitui a informação sobre

o índice do argumento pela informação sobre o número de variáveis existentes nesse argumento. Vejamos o exemplo da chamada *p(A,B,1,2)* na Figura 5.3 para melhor compreendermos de que forma este vector é construído. As primeiras duas posições do vector, as posições 0 e 1, têm ambas o número de variáveis a 0 uma vez que se referem a duas constantes. Os restantes dois elementos dizem respeito a variáveis e por essa razão cada uma tem um 1 na posição do número de variáveis. Depois de construído, este vector pode ser acedido através da *subgoal frame* respectiva e será particularmente útil na inserção de respostas nas tabelas, tal como discutido na próxima secção.

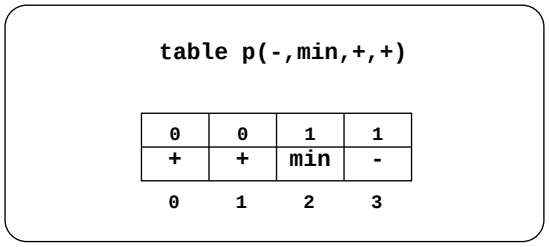

Figura 5.3: Vector de variáveis construído para a chamada *p(A,B,1,2)* coma declaração *p(-,min,+,+)*.

5.3 Inserção de Respostas nas Tabelas

No novo sistema, as respostas encontradas durante a execução de um programa tabelado são inseridas igualmente na *trie de respostas* respectiva, como de resto já acontecia com a tabulação tradicional. Contudo, neste novo sistema, esse processo é um pouco diferente. Nos parágrafos que se seguem explicamos em detalhe as novas operações realizadas.

Numa *trie de respostas*, as respostas são somente representadas pelas substituições das variáveis presentes no subgolo. Quando consideramos cada uma delas individualmente é necessário saber que operador devemos aplicar uma vez que pode não haver uma correspondência directa entre as variáveis e os operadores. Para fazer esse controle utilizamos a informação que se encontra no vector de variáveis da *subgoal frame* sobre o número de variáveis presentes em cada argumento do subgolo. Tomemos como exemplo o predicado *p/4*, que temos estado a ver, e vamos considerar que encontramos a resposta $\{A=5,\ B=7\}$ para o subgolo *p(A,B,1,2)*. As substituições são consideradas pela mesma ordem com que as variáveis que substituem foram inseridas na *trie* de subgolos. Por esse motivo a primeira substituição a considerar é a $\{B=7\}$ e só depois a $\{A=5\}$. Para sabermos que operador devemos considerar é necessário consultar

o vector com o número de variáveis construído anteriormente. Descobrimos que o operador a ser utilizado é o *min* uma vez que é o primeiro operador a ter variáveis. Neste caso concreto é no entanto irrelevante saber que operador devemos utilizar uma vez que não existem mais respostas na tabela. Logo, a constante 7 é inserida sem ser necessário qualquer tipo de operação de comparação. Resta agora considerar a constante 5. Uma vez que o operador *min* só tinha uma variável, não pode ser este o operador a aplicar. Na próxima posição do vector, o operador - também tem uma variável o que indica que é este o operador que deve ser aplicado. Mais uma vez, por não haver mais respostas na tabela, esta informação torna-se irrelevante sendo a constante 5 inserida sem necessidade de fazer qualquer tipo de comparação.

Neste exemplo os operadores não tiveram um papel muito relevante. Vejamos agora o que acontecia caso fosse encontrada posteriormente a resposta $\{A{=}9,\ B{=}6\}$. O primeiro elemento a considerar seria a substituição $\{B{=}6\}$. O processo para definir qual o operador a utilizar é o mesmo que vimos anteriormente. Desta vez, como já existe uma resposta na tabela, quando vamos inserir a constante 6, o operador *min* irá forçar a que as constantes 6 e 7 (a resposta que já constava na tabela) sejam comparadas. Quando comparamos ambas as resposta concluímos que a nova, por ser menor, é melhor que a anterior. Tal facto leva a que a resposta antiga tenha de ser invalidada para que mais tarde no decorrer da computação não seja considerada. O processo de invalidação consiste em remover todos os nós intermédios e colocar uma etiqueta no nó folha que marca a resposta como invalidada. A razão pela qual mantemos o nó folha é explicada na próximo secção. Depois da invalidação estar completa são inseridos os elementos seguintes da nova resposta uma vez que não é necessário compará-los com mais nenhum. O aspecto final desta *trie* está representado na Figura 5.4. Conforme podemos ver, o nó folha da resposta invalidada é mantido e o encadeamento entre o nó folha da resposta anterior e o nó folha da nova resposta é também efectuado.

O processo de invalidação tem um papel fundamental na implementação dos operadores como o *min*, *max* e *last*. Este tipo de operadores força a que haja uma actualização dos valores representados na tabela. Para que tal seja possível é necessário marcar as respostas anteriores como inválidas de maneira a que não sejam consideradas em computações subsequentes. A necessidade de mudar a ordem dos argumentos, tal como discutido nas secções anteriores, está directamente relacionada com o mecanismo de invalidação. Uma vez que a inserção nas *tries* é feita de cima para baixo, a ordem que definimos permite que a invalidação seja feita da forma mais simples e eficiente possível, isto é, quando uma parte da resposta tem que ser invalidada a partir de um

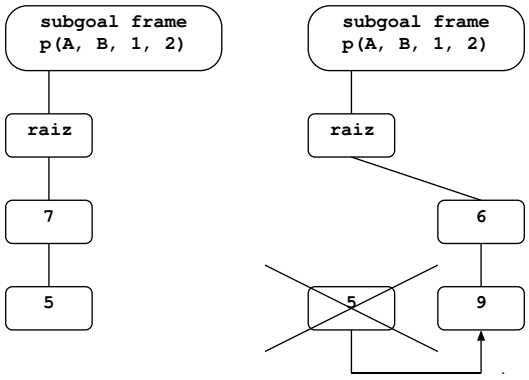

Figura 5.4: Invalidação de uma resposta na *trie* de respostas.

determinado nó, isso deve ser feito de modo a todos os ramos que estejam para baixo desse nó possam ser eliminados deixando intactos os nós de cima. Vejamos o exemplo da Figura 5.5 que nos apresenta duas *tries* de respostas para o subgolo *p(A,B,C,D)* referente ao predicado *p/4* tabelado como *p(-,min,+,+)*. Na *trie* da esquerda os elementos são inseridos sem alteração de ordem, enquanto na da direita os elementos são inseridos pela nova ordem definida pelos operadores de modo. A resposta que constava da *trie* era a {*A=1, B=2, C=7, D=8*} e a resposta que entretanto foi encontrada é a {*A=5, B=1, C=7, D=8*}. Como a segunda resposta é melhor, é necessário que a primeira seja invalidada. Conforme podemos ver, utilizando o nosso método (do lado direito), quando encontramos a nova resposta começamos por inserir os elementos indexados 7 e 8. Seguidamente vamos considerar a inserção do elemento 1. Verificamos que este é menor do que o que está na tabela e procedemos à invalidação dos nós que estão para baixo deste. Caso seguíssemos o modo de inserção normal de respostas, começaríamos por inserir primeiro o elemento 5 e só posteriormente consideraríamos o elemento 1. Para detectarmos que a nova resposta era melhor do que a que estava na tabela seria necessário navegar na *trie* e para conseguirmos invalidar a outra resposta seria necessário ajustar os nós anteriores da *trie*, tornando todo este processo bastante mais demorado e também mais complexo.

5.4 Pontos de Escolha

Conforme vimos anteriormente, os pontos de escolha têm como principal função guardar as alternativas que ficam em aberto durante a execução de um programa (a representação esquemática de um ponto de escolha pode ser vista na Figura 2.2). Na tabulação, a estrutura dos pontos de escolha é estendida para guardar mais in-

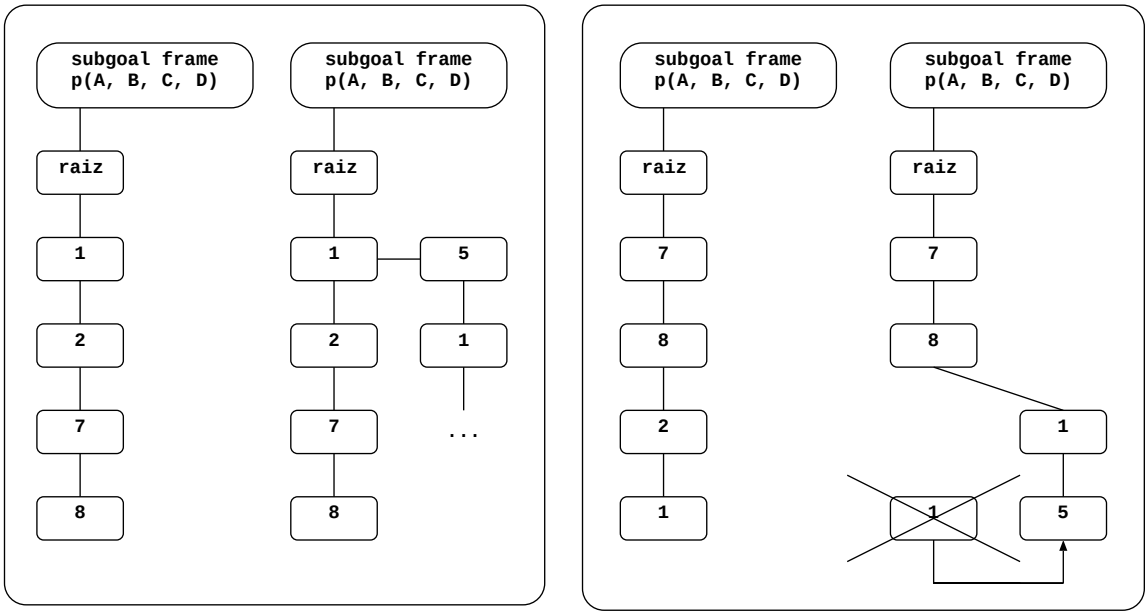

Figura 5.5: Como a mudança da ordem de inserção dos elementos das respostas pode facilitar o mecanismo de invalidação.

formação. Os pontos de escolha de predicados tabelados podem tomar duas formas: a de *geradores* ou a de *consumidores* consoante sejam, respectivamente, referentes a subgolos que apareceram pela primeira vez durante a execução do programa ou referentes a chamadas repetidas. Os pontos de escolha geradores possuem um campo extra que referência a *subgoal frame* respectiva, ao passo que os pontos de escolha consumidores referênciam uma outra estrutura de dados chamada *dependency frame*. Esta estrutura permite ao ponto de escolha consumidor guardar diversas informações importantes para a recuperação do estado da computação, e entre elas destaca-se o apontador para a última resposta consumida. Este apontador merecerá, da nossa parte, uma atenção especial no exemplo que iremos ver de seguida onde mostramos como os pontos de escolha se relacionam com a tabulação e com a forma de declarar predicados tabelados.

O exemplo da Figura 5.6 apresenta do lado superior esquerdo o código do programa, do lado superior direito a árvore de execução do programa e em baixo uma representação da pilha dos pontos de escolha e a *trie* de respostas para o subgolo *a(X)*. Iremos, de seguida, fazer a descrição passo a passo do programa focando a atenção na criação dos pontos de escolha. Começamos por efectuar a consulta *a(X)* (passo 0). Por se tratar de um predicado tabelado e por ser a primeira vez que esta chamada aparece é criado um ponto de escolha gerador. A consulta unifica com a primeira cláusula do predicado

a(X) que origina a chamada ao predicado *b(X)* (passo1) que, por sua vez, origina uma chamada a *a(Y)* (passo 2). Por se tratar de um chamada repetida é colocado um ponto de escolha consumidor na pilha de pontos de escolha. Para além disso, o nó 2 da computação é suspenso voltando a computação ao ponto de escolha gerador que tinha uma hipótese em aberto: o facto *a(2)*. A exploração deste leva à descoberta da resposta *X=2* (passo 3). Neste momento estamos em condições de reactivar o nó 2. Este nó irá agora consumir a nova resposta ficando com o apontador da última resposta consumida a apontar para a resposta 2. O consumo da resposta origina a chamada ao predicado *c(X)* (passo 4). A avaliação deste predicado chama novamente uma variante de *a(X)*, o subgolo *a(Z)*, o que leva, conforme vimos anteriormente, a que seja criado um novo ponto de escolha consumidor e a que seja consumida a resposta 2. Após a computação de *c(X)* é descoberta a resposta *X=1*. Como o predicado *a/1* está tabelado com o operador *min*, só a resposta mínima será guardada, o que irá levar a que a resposta anterior *X=2* seja invalidada. No entanto, conforme podemos reparar na parte inferior da Figura 5.6, o primeiro ponto de escolha consumidor continua a apontar para a resposta invalidada e é por isso que os nós folha da *trie* de respostas nunca são eliminados. Se tal acontecesse, o nó consumidor *a(Y)* não seria capaz de seguir o encadeamento de respostas nem de consumir a nova resposta que entretanto foi encontrada.

5.5 Resumo

Neste capítulo explicámos de que forma são implementados os operadores de declaração de modo dos predicados tabelados no YapTab. Abordámos o novo sistema de inserção de subgolos e respostas na tabela que nos permite invalidar respostas de uma forma simples e eficiente. Vimos que a invalidação de respostas nos permite actualizar os valores das respostas nas *tries* e permite que os pontos de escolha mantenham o mecanismo de consumo de respostas seguindo o encadeamento dos nós folha.

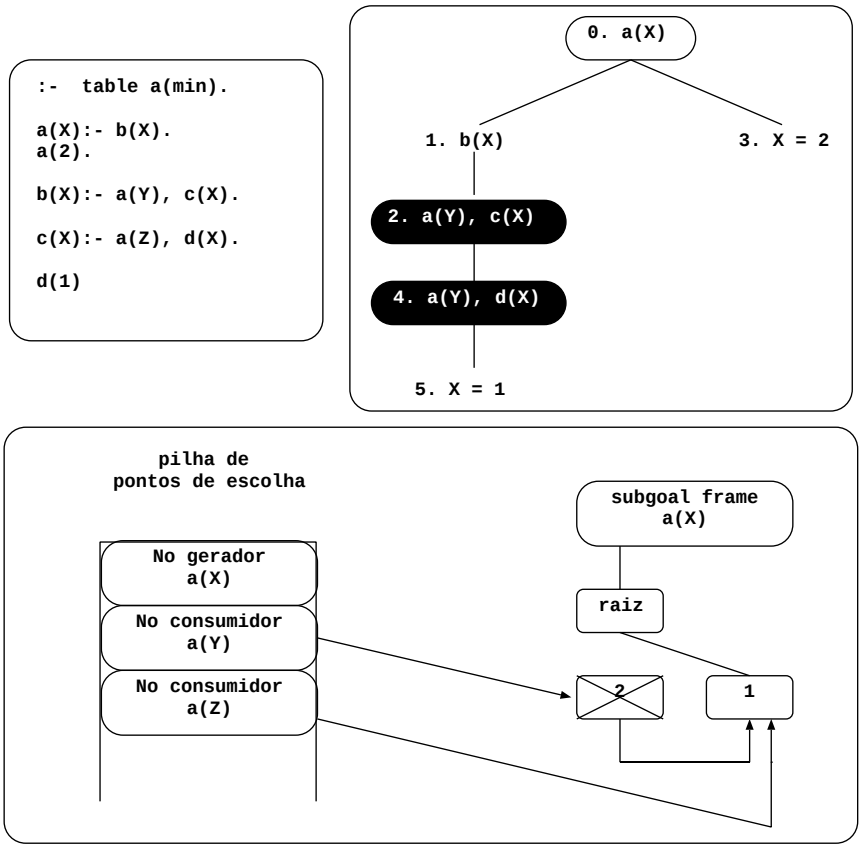

Figura 5.6: Importância de manter o nó folha de uma resposta invalidada.

Capítulo 6

Avaliação

Neste capítulo apresentamos uma análise do desempenho dos diferentes mecanismos propostos nesta tese. Para tal utilizamos programas que têm por objectivo calcular o caminho mínimo de um grafo recorrendo aos diversos mecanismos e apresentamos os seus tempos de execução comparando-os com os dos mecanismos do XSB Prolog. Terminamos justificando os resultados obtidos.

6.1 Conjunto de Programas

Para analisar o desempenho dos diversos mecanismos propostos nesta tese utilizamos programas que calculam o caminho mínimo de um grafo. Estes programas foram implementados recorrendo aos três mecanismos apresentados nesta tese: utilizando o operador *min*, o mecanismo de preferências e a *Answer Subsumption*. Implementamos também programas semelhantes recorrendo aos predicados do XSB Prolog *filterReduce/4* e *bagReduce/4* para os podermos comparar com os resultados obtidos pelas nossas propostas. O código da implementação dos diversos programas resultantes da utilização dos vários mecanismos é apresentado em seguida:

- Programa construído utilizando o operador *min*:

```
:- table path(+,+,min).

path(X,Y,E):- path(X,Z,E1), edge(Z,Y,E2), E is E1 + E2.
path(X,Y,E):- edge(X,Y,E).
```

- Programa construído utilizando o mecanismo de preferências:

```
:- table path(+,+,last).

pathNew(X,Y,C):- path(X,Z,C1), edge(Z,Y,C2), C is C1 + C2.
pathNew(X,Y,C):- edge(X,Y,C).

pathPrefer([X,Y,C],[X1,Y1,C1]):- set_value(first,0),path(X1,Y1,C1),
                                 path(X1,Y1,C1) <<< path(X,Y,C),!.
pathPrefer(_,_):- !, get_value(first,0).

path(X,Y,C):- copy_term(X,X1), copy_term(Y,Y1), copy_term(C,C1),
              pathNew(X,Y,C), pathPrefer([X,Y,C],[X1,Y1,C1]).

path(X,Y,C1) <<< path(X,Y,C2) :- set_value(first,1), C2 < C1.
```

- Programa construído utilizando o predicado de *answer subsumption filter/3*:

```
pathNew(X,Y,C):- path(X,Z,C1), edge(Z,Y,C2), C is C1 + C2.
pathNew(X,Y,C):- edge(X,Y,C).

min(One,Two,Min):- Two < One, Min=Two.

path(X, Y, C):- filter(pathNew(X,Y),min,C).
```

- Programa construído utilizando o predicado de answer subsumption do XSB *filterReduce/4*:

```
pathNew(X,Y,C):- path(X,Z,C1), edge(Z,Y,C2), C is C1 + C2.
pathNew(X,Y,C):- edge(X,Y,C).

min(One,Two,Min):- One > Two -> Min = Two ; Min = One.

path(X,Y,C):- bagReduce(pathNew(X,Y),C,min,infinity).
```

- Programa construído utilizando o predicado de answer subsumption do XSB *bagReduce/4*:

```
pathNew(X,Y,C):- path(X,Z,C1), edge(Z,Y,C2), C is C1 + C2.
pathNew(X,Y,C):- edge(X,Y,C).

min(One,Two,Min):- One > Two -> Min = Two ; Min = One.

path(X,Y,C):- filterReduce(pathNew(X,Y),C,min,infinity).
```

Nos programas acima, a cláusula de recursividade aparece em primeiro lugar sendo que o predicado responsável por essa recursividade se encontra do lado esquerdo. Para a realização destes testes consideramos todas as combinações possíveis de recursividade. De seguida apresentamos todas as combinações que foram consideradas para o exemplo em que utilizamos o operador *min*, nos outros mecanismos procedemos de forma análoga.

- Recursão à esquerda com a cláusula recursiva em primeiro lugar (*path_left_first*):

```
:- table path(+,+,min).

path(X,Y,E):- path(X,Z,E1), edge(Z,Y,E2), E is E1 + E2.
path(X,Y,E):- edge(X,Y,E).
```

- Recursão à esquerda com a cláusula recursiva em último lugar (*path_left_last*):

```
:- table path(+,+,min).

path(X,Y,E):- edge(X,Y,E).
path(X,Y,E):- path(X,Z,E1), edge(Z,Y,E2), E is E1 + E2.
```

- Recursão à direita com a cláusula recursiva em primeiro lugar (*path_right_first*):

```
:- table path(+,+,min).

path(X,Y,E):- edge(X,Z,E1), path(Z,Y,E2), E is E1 + E2.
path(X,Y,E):- edge(X,Y,E).
```

- Recursão à direita com a cláusula recursiva em último lugar (*path_right_last*):

```
:- table path(+,+,min).

path(X,Y,E):- edge(X,Y,E).
path(X,Y,E):- edge(X,Z,E1), path(Z,Y,E2), E is E1 + E2.
```

Os grafos utilizados com os programas de teste têm quatro configurações distintas: *btree*, *cycle*, *pyramid* e *grid*. Uma representação esquemática destas configurações pode ser vista na Figura 6.1. A profundidade utilizada para realizar as medições dos tempos de execução foi de 12 para a *btree*, 100 para o *cycle*, 100 para a *pyramid* e 12 para a *grid*.

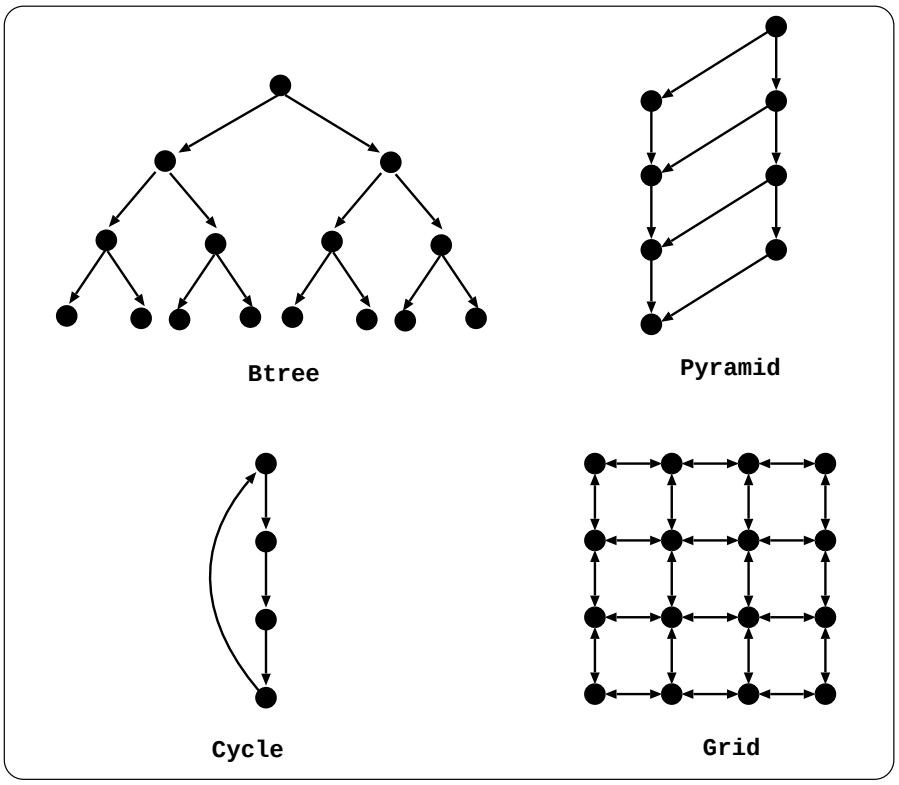

Figura 6.1: Exemplos de configurações com profundidade 4 dos grafos utilizados nos programas de teste.

6.2 Avaliação do Desempenho

De forma a avaliar o desempenho medimos os tempos de execução em milissegundos para os diversos programas juntamente com as diversas possibilidades de recursão

Tabela 6.1: Tempos de execução, em milisegundos, dos diversos mecanismos na execução de programas que calculam o caminho mínimo de um grafo.

Programa		**YapTab**			**XSB**	
		min	***pref***	***filter***	***filterReduce***	***bagReduce***
path_left_first	*btree_12*	10	96.540	129.000	587.920	110
	cycle_100	10	5.810	7.810	34.269	30
	grid_12	20	101.480	148.150	548.070	260
	pyramid_100	10	17.360	23.060	96.680	60
path_left_last	*btree_12*	20	96.290	128.040	578.190	110
	cycle_100	10	5.850	7.850	34.480	19
	grid_12	20	101.500	141.550	509.470	260
	pyramid_100	10	16.950	22.920	95.070	60
path_right_first	*btree_12*	40	96.780	128.560	579.290	220
	cycle_100	10	5.800	7.850	35.010	40
	grid_12	40	145.770	183.290	691.010	710
	pyramid_100	10	17.050	23.650	98.979	100
path_right_last	*btree_12*	40	98.130	131.730	580.860	210
	cycle_100	10	5.900	7.920	34.860	49
	grid_12	40	136.630	182.540	693.590	700
	pyramid_100	10	17.340	23.210	99.270	110

e grafos que referimos na secção anterior. Todos os testes foram executados no mesmo ambiente: um computador com um processador Intel(R) Core(TM)2 Quad CPU Q9550 @ 2.83GHz com 4 Gb de RAM a correr o sistema operativo Mandriva 2009.1 de 32 bits. Podemos ver na Tabela 6.1 os resultados obtidos para uma média de 3 execuções para cada teste.

Conforme podemos verificar facilmente, o operador *min* apresenta os melhores resultados neste conjunto de testes. Tal facto é facilmente explicado por este estar implementado a baixo nível na máquina de execução em oposição aos restantes mecanismos que estão implementados parcialmente ou mesmo totalmente em Prolog.

Relativamente ao mecanismo das preferências, observando os resultados da tabela, verificamos que os tempos obtidos, comparativamente com o operador *min*, são geralmente bastante piores. Isto acontece porque este mecanismo, para além de estar escrito em Prolog apresenta a possibilidade do utilizador poder definir os seus próprios

critérios de preferência em vez de estar limitado aos operadores de modo *min* e *max* da tabulação. Tais funcionalidades apresentam um custo acrescido durante a execução do programa.

Quando comparados, o mecanismo de preferências e o de *answer subsumption*, apresentam resultados dentro da mesma ordem de grandeza, sendo que o predicado *filter/3* é cerca de 1.4 vezes pior, em média, do que as preferências. Esta diferença é facilmente explicada, em primeiro lugar, pelo grande número de operações que têm que ser feitas no predicado *filter/3* de forma a implementar o mecanismo de *answer subsumption* e, em segundo lugar, pelo facto do predicado ter de percorrer todas as respostas que estão na tabela ao invés de considerar somente a primeira resposta compatível, como acontece com o mecanismo preferências.

Se nos debruçarmos agora sobre os predicados do XSB Prolog, podemos ver que os predicados *filterReduce/4* e o *bagReduce/4* apresentam tempos muito diferentes um do outro. O predicado *bagReduce/4* consegue resultados bastante melhores devido a uma implementação mais eficiente. A diferença principal reside na forma como o predicado *get_returns/3*, que é comum aos dois, é chamado. Este predicado está implementado a baixo nível e é responsável por retornar as respostas que estão na tabela. No caso do *filterReduce/4* a chamada é feita de forma a retornar todas as respostas, sendo a verificação de compatibilidade posteriormente feita ao nível do Prolog, ao passo que no *bagReduce/4*, só são retornadas desde logo as respostas compatíveis sendo essa verificação feita a baixo nível pelo próprio predicado *get_returns/3*.

No entanto se repararmos com atenção, tanto o nosso predicado *filter/3* como o predicado *filterReduce/4* do XSB percorrem todas as respostas que estão na tabela e fazem testes de compatibilidade ao nível do Prolog. Por este motivo, à partida, não seria de esperar que o predicado *filterReduce/4* fosse 4.2 vezes, em média, pior do que o predicado *filter/3*. A razão pela qual isto acontece reside numa subtileza da implementação. No predicado *filterReduce/4*, a lista de respostas é construída primeiro e só depois é percorrida elemento a elemento para verificar a compatibilidade de respostas, ao passo que no predicado *filter/3* só são colocados na lista os elementos compatíveis. O custo extra de inserir mais elementos na lista do que os que são necessários é o suficiente para que em grafos de alguma dimensão haja diferenças de tempos tão significativas.

6.3 Resumo

Neste capítulo analisámos os tempos de execução obtidos pelos nossos mecanismos e pelos disponíveis no XSB Prolog e explicámos os motivos das diferenças observadas nesses resultados.

Capítulo 7

Conclusão

Este capítulo sumariza o trabalho desenvolvido nesta tese. Começaremos por apresentar as principais contribuições deste trabalho e em seguida sugerimos alguns problemas que podem ser solucionados num trabalho futuro.

7.1 Principais Contribuições

Esta tese tinha como principais objectivos: (1) implementar os operadores de modo da tabulação no sistema YapTab e (2) explorar as áreas onde estes podem ter aplicação. Tal como vimos, o Yap Prolog já possuía um sistema de tabulação denominado YapTab. O nosso primeiro objectivo foi então dotar este sistema de mecanismos que permitissem suportar os operadores de modo da tabulação. O Yap Prolog passa, desta forma, a ser um dos poucos sistemas a incluir esta funcionalidade. Os outros dois sistemas que conhecemos que implementam os operadores de modo da tabulação são o TALS [8], o sistema de tabulação do ALS Prolog, e o B-Prolog.

As outras duas contribuições desta tese estão relacionadas com a utilização dos operadores de modo. Guo et al. já tinham demonstrado que os operadores de modo poderiam ser utilizados para solucionar problemas de preferências [10], sugerindo para isso alterações à sintaxe inicial proposta por Govindarajan et al. [6, 5] e criando uma série de regras de transformação de programas que permitissem aos sistemas de tabulação que possuem operadores de modo interpretá-los. A abordagem de Guo et al. apresenta no entanto um problema: não é possível que a chamada ao predicado sobre a qual queremos aplicar as preferências tenha as variáveis indexadas livres. Nesta

tese nós propomos um novo predicado, baseado na abordagem de Guo et al., capaz de ultrapassar esta limitação.

Nesta tese, demonstramos também que estes operadores podem ser utilizados numa área de investigação denominada de *answer subsumption*. Já existem diversos mecanismos implementados de *answer subsumption*, contudo nenhum usa operadores de modo. Referimos dois que são utilizados pelo XSB Prolog. o *filterReduce/4* e o *bagReduce/4*. A principal diferença entre os dois é que o primeiro usa a variância para verificar a compatibilidade das respostas e o outro usa a unificação. Mas ambos sofrem da mesma limitação: só conseguem considerar uma resposta compatível por cada nova resposta encontrada. Nesta tese, nós propomos um predicado denominado *filter/3* que usa a variância para aferir se as respostas são compatíveis e permite considerar um grupo de respostas por cada nova resposta encontrada.

7.2 Trabalho Futuro

Como trabalho futuro propomos três tópicos. O primeiro é referente à usabilidade do predicado de *answer subsumption filter/3*. Conforme vimos, o predicado *filter/3* faz uso do operador *last* que substitui sempre a resposta que está na tabela por uma mais recente que entretanto apareça referente às mesmas variáveis indexadas. Nos exemplos práticos da utilização do nosso predicado na secção 4.3.3, vimos que para o último exemplo era necessário controlar se a resposta que íamos inserir era exactamente igual à que lá estava e que esse controle tinha de ser feito pelo programador. No entanto a nível da usabilidade esta não é uma boa solução pois obriga a que o programador tenha algum conhecimento da implementação do nosso predicado. Um possível trabalho a realizar no futuro seria justamente passar esse controle para baixo nível sendo feito pelo procedimento de inserção das respostas nas *tries*.

Outro tópico que poderia ser objecto de alguma atenção é o modo como acedemos à tabela para ir buscar respostas compatíveis. O que fazemos neste momento é utilizar uma chamada repetida do predicado tabelado, para retornar todas as respostas que estão na tabela referentes aquele subgolo, para posteriormente verificarmos aquelas que são compatíveis. Seria interessante ter um predicado *built-in* que fizesse tal controlo a nível da máquina de execução, retornando apenas as respostas compatíveis, como acontece no XSB Prolog. Isso traria melhorias a nível da performance da nossa aplicação de preferências e do nosso predicado de *answer subsumption*.

Por último, seria importante poder obter uma maior *feedback* da utilização dos nossos predicados, por exemplo, por uma maior experimentação na resolução de problemas reais.

Referências

[1] H. Aït-Kaci. *Warren's Abstract Machine – A Tutorial Reconstruction*. The MIT Press, 1991.

[2] M. Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine*. PhD thesis, The Royal Institute of Technology, 1990.

[3] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un Système de Sommunication Homme–Machine en Francais. Technical report cri 72-18, Groupe Intelligence Artificielle, Universitè Aix-Marseille II, 1973.

[4] E. Fredkin. Trie Memory. *Communications of the ACM*, 3:490–499, 1962.

[5] K. Govindarajan. Optimization and Relaxation in Logic Languages. Technical report, Department of Computer Science, SUNY-Buffalo, 1997.

[6] K. Govindarajan, B. Jayaraman, and Mantha S. Preference Logic Programming. In *12th International Conference on Logic Programming*, pages 731–745. MIT Press, 1995.

[7] Hai-Feng Guo and G. Gupta. Simplifying Dynamic Programming via Mode-directed Tabling. *Software Practice Experience*, 38(1):75–94, 2008.

[8] Hai-Feng Guo and Gopal Gupta. A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In *International Conference on Logic Programming*, pages 181–196. Springer-Verlag, 2001.

[9] Hai-Feng Guo and B. Jayaraman. Mode-directed Preferences for Logic Programs. In *2005 ACM symposium on Applied computing*, pages 1414–1418. ACM, 2005.

[10] Hai-Feng Guo, B. Jayaraman, G. Gupta, and M. Liu. Optimization with Mode-Directed Preferences. In *7th ACM SIGPLAN International Conference*

on Principles and Practice of Declarative Programming, pages 242–251. ACM, 2005.

[11] G. Gupta, K. Ali, M. Carlsson, and M. V. Hermenegildo. Parallel Execution of Prolog Programs: A Survey. Research report, Laboratory for Logic, Databases and Advanced Programming, New Mexico State University, 1997.

[12] R. Kowalski. Predicate Logic as a Programming Language. In *Information Processing*, pages 569–574. North-Holland, 1974.

[13] L. Larmore and B. Schieber. On-Line Dynamic Programming with Applications to the Prediction of RNA Secondary Structure. In *First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 503–512. Society for Industrial and Applied Mathematics, 1990.

[14] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[15] D. Michie. Memo Functions and Machine Learning. *Nature*, 218:19–22, 1968.

[16] G. Pemmasani, Hai-Feng Guo, Y. Dong, C. R. Ramakrishnan, and I. V. Ramakrishnan. Online Justification for Tabled Logic Programs. In *7th International Symposium on Functional and Logic Programming*, pages 24–38. Springer-Verlag, 2004.

[17] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Access Mechanisms for Tabled Logic Programs. *Journal of Logic Programming*, 38(1):31–54, 1999.

[18] F. Riguzzi and T. Swift. Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions. In *Technical Communications of the 26th International Conference on Logic Programming*, volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 162–171. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[19] J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[20] R. Rocha, F. Silva, and V. Santos Costa. A Tabling Engine for the Yap Prolog System. In *APPIA-GULP-PRODE Joint Conference on Declarative Programming*, 2000.

[21] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, 1990.

[22] A. Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying Proofs Using Memo Tables. In *2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 178–189. ACM, 2000.

[23] D. H. D. Warren. *Applied Logic – Its Use and Implementation as a Programming Tool.* PhD thesis, Edinburgh University, 1977.

[24] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[25] D. S. Warren, T. Swift, K. Sagonas, J. Freire, P. Rao, B. Cui, E. Johnson, L. de Castro, R. F. Marques, D. Saha, and S. Dawson and. Kifer. The XSB System Version 3.2 Volume 1: Programmer's Manual, 2009.