# Sistema para Reconhecimento e Geração de Planos

Angelo E. M. Ciarlini
e-mail: angelo@inf.puc-rio.br

Antonio L. Furtado
e-mail: furtado@inf.puc-rio.br

PUC-RioInf.MCC 02/96 February, 1996

Abstract: Plan recognition is an important feature of database cooperative interfaces, since if a system knows what its user intends to do, it will be in a better position to help him. Planning gains importance when detected plans violate integrity constraints, because it can be used both to adapt a detected plan and to generate another one with the same effects. This paper presents a prototype where plan recognition and generation are integrated within a database context.

Keywords: plan recognition, planning, database, cooperative interfaces.

Resumo: O reconhecimento de planos é um aspecto importante nas interfaces cooperativas de bancos de dados, uma vez que, quanto melhor um sistema saiba o que o usuário pretende fazer, melhor pode ser sua cooperação. O planejamento torna-se importante quando planos detectados violam restrições de integridade, porque pode ser usado tanto para adaptar um plano detectado quanto para gerar um novo plano com os mesmos efeitos. Este trabalho apresenta um protótipo onde o reconhecimento e a geração de planos são integrados dentro de um contexto de banco de dados.

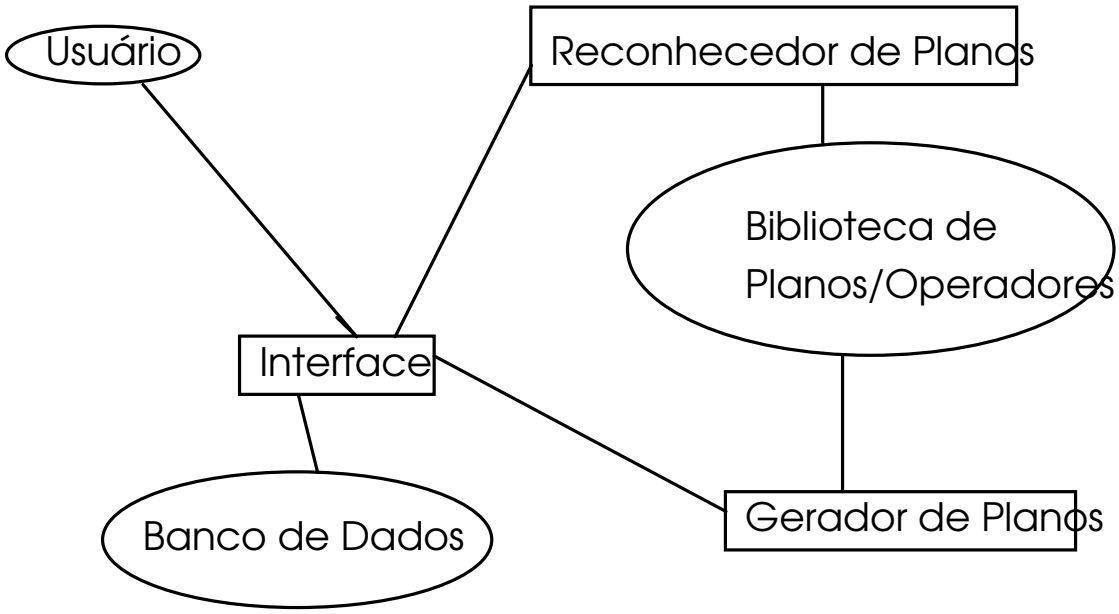
Palavras-chave: reconhecimento de planos, geração de planos, bancos de dados, interfaces cooperativas.

# 1. Introdução

Este documento tem a função de apresentar o Sistema para Reconhecimento e Geração de Planos (SRGP), indicando sua utilidade e descrevendo sua estrutura, funcionamento e modo de usar.

Os sistemas de reconhecimento de planos trabalham com bibliotecas de planos, criadas estaticamente, que, em princípio, contêm todas as possíveis intenções dos usuários. A interação do usuário com um sistema desses consiste em fornecer observações do que pretende fazer e, a partir disso, o sistema deve encontrar um plano na biblioteca que corresponda ao que ele deseja fazer e como deseja fazê-lo. No SRGP, admite-se que um plano reconhecido pode não ser aplicável, pois alguma precondição para a sua execução pode não ser satisfeita. Nesse caso, o sistema admite duas hipóteses: adaptar o plano reconhecido ou gerar um novo plano, uma vez que o objetivo do usuário já foi detectado.

O SRGP é constituído de três partes: um reconhecedor, um planejador e uma interface que faz a ligação entre os dois subsistemas e se comunica com o usuário. A figura a seguir demonstra a estrutura e funcionamento do SRGP.




Tanto o gerador quanto o reconhecedor de planos compartilham a mesma biblioteca de planos/operadores. Para o SRGP, os planos são operadores complexos compostos por outros operadores, formando assim uma hierarquia de composição.

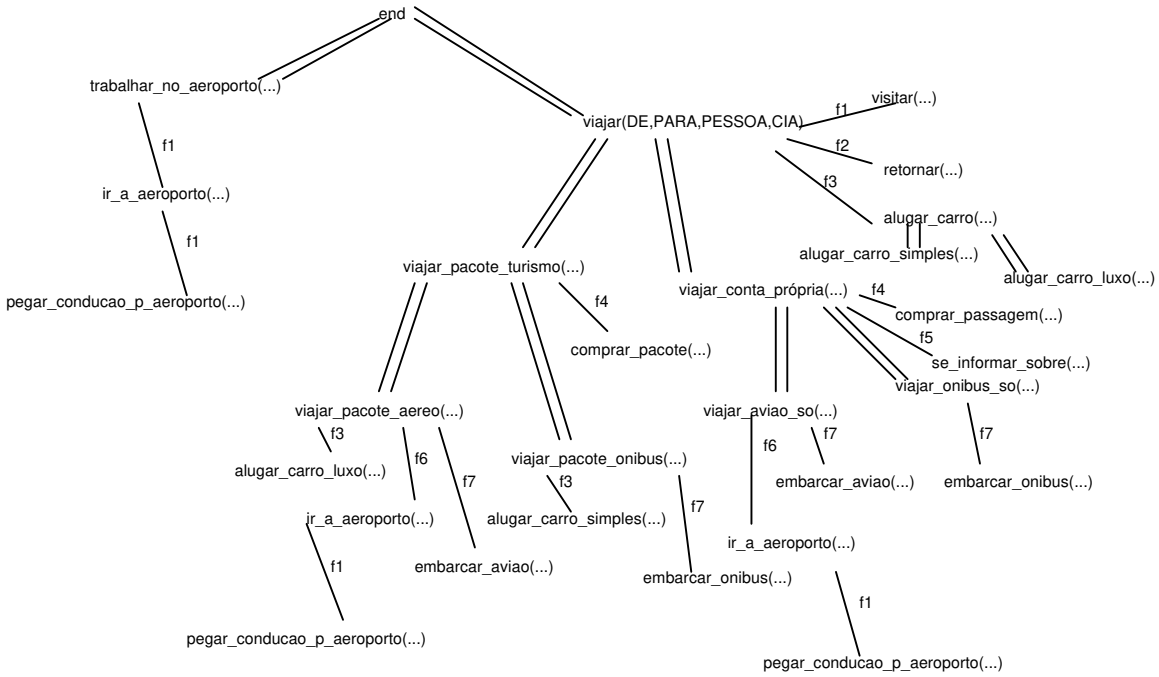
O usuário fornece, através da interface, uma lista de observações (operadores) correspondentes a ações que pretende executar para atingir um determinado objetivo, que consiste nos efeitos do plano a ser detectado. Com base nessas observações, é aplicado um algoritmo de reconhecimento de planos que consulta uma biblioteca criada previamente. Uma vez reconhecido um plano, a interface verifica na base de dados se tal plano pode ser executado. Se puder, o plano é informado ao usuário, caso contrário são indicadas, além do plano reconhecido, as precondições não satisfeitas. Nesse momento, o usuário tem a opção de solicitar que o gerador de planos seja acionado, seja para adaptar o plano reconhecido ou para gerar um novo plano com os mesmos efeitos.

Além da hierarquia de composição já citada, há duas outras hierarquias utilizadas no SRGP. A primeira delas é uma hierarquia de abstrações, segundo a qual os operadores herdam a composição de operadores mais abstratos. A hierarquia de abstrações é sempre uma floresta, ou seja, não é admitida herança múltipla. A hierarquia de composições, contudo, não é obrigatoriamente uma floresta pois um determinado operador pode ser componente de vários operadores distintos. A hierarquia restante refere-se a níveis sucessivos de criticalidade de precondições, determinando quais precondições devem ser resolvidas primeiro no processo de geração de planos.

O SRGP foi integralmente implementado em Prolog. O reconhecedor de planos adota o algoritmo especificado por Kautz em [Ka91]. O gerador de planos é uma adaptação em Prolog do ABTWEAK [YTW92], originalmente implementado em Lisp por Woods [Wo91]. O ABTWEAK estende as características de comprometimento mínimo de planejadores não-lineares, tais como o TWEAK [Ch87], com as de planejadores hierárquicos, como o ABSTRIPS [Sa74]. Como foi dito antes, os algoritmos de reconhecimento e geração foram adaptados de forma a compartilharem uma mesma biblioteca de planos/operadores. Além disso, o gerador passou a admitir que o planejamento fosse iniciado a partir de um plano da biblioteca a ser adaptado. O algoritmo de reconhecimento de Kautz foi estendido para admitir que os planos/operadores possuíssem, além de uma identificação, um número qualquer de variáveis, que permite parametrizá-los. Foi utilizado nesta implementação o ARITY PROLOG 6.0 [Ari88].

## 2. O Reconhecimento de Planos

A fim de ilustrar o reconhecimento de planos, é apresentado um exemplo de biblioteca com um contexto referente a viagens de avião ou de ônibus, por conta própria ou através de pacotes de turismo. A figura a seguir mostra a hierarquia de composição (arcos simples, com nome) e de abstração (arcos duplos) dos planos/operadores. A relação inversa da abstração é chamada de especialização.




O reconhecimento de planos, conforme especificado no algoritmo de Kautz, consiste em gerar grafos de explicação ("explanation graphs") das observações fornecidas. Esses grafos contêm, como nós, os planos/operadores e, como arcos, arcos de composição e de abstração. Existe um plano denominado *end* que é a abstração de todos os planos que podem ser reconhecidos como objetivo final do usuário. O grafo de explicação de uma observação descreve de que forma o operador correspondente a essa observação pode ser enquadrado como parte de algum(ns) plano(s) que seja uma especialização do plano *end*.

No exemplo, se for fornecido, como observação, um operador *comprar_passagem*, o sistema verificará que esse operador é componente do operador *viajar_conta_própria* que é uma especialização de *viajar* que por sua vez é uma especialização de *end*. Como *viajar_conta_própria* possui as especializações viajar_avião_só e viajar_ônibus_só, esses planos/operadores farão parte do grafo e serão os planos mais especializados, os quais podem ser reconhecidos como possíveis objetivos do usuário.

Os grafos são montados examinando-se as hierarquias de composição e abstração. O processo de construção dos grafos é recursivo e implica em verificar para um dado operador quais são os planos/operadores que o têm como componente, como abstração ou como especialização. Para que a explicação se complete, é necessário que exista pelo menos um caminho, através de arcos de composição e de abstração, entre o operador correspondente à observação e o plano *end*.

Após se montar um grafo para cada observação, os grafos são unificados, igualando-se os nós do tipo *end*, propagando-se a igualdade e reduzindo-se o número de alternativas. O grafo final possui todas as especializações de *end* onde todos as observações fornecidas se encaixam, ou seja, todas elas são operadores a serem executados.

Em relação ao algoritmo definido por Kautz, o programa apresenta a diferença de não admitir associar aos operadores restrições temporais ou valores tais como nomes próprios. As restrições temporais que estabelecem um intervalo de tempo para a execução de um operador foram suprimidas para simplificar a implementação nesta primeira versão, uma vez que era necessário integrar o reconhecimento com a geração de planos, que não trabalha com intervalos. Os valores (nomes próprios) não precisaram mais ser definidos como parte dos operadores em virtude de se ter adotado uma parametrização na sua identificação. Essa parametrização foi adotada por questões de compatibilidade com a forma como são definidos os operadores no gerador de planos.

## 2.1 Quem Usa um Operador

Na montagem dos grafos, uma das tarefas fundamentais é descobrir o conjunto de operadores que podem ter um certo operador como seu componente, fato este denominado um **uso**. Formalmente, um uso é uma tripla **<Ec,r,Ei>**, onde **Ec** é o componente, **Ei** é o usuário e **r** é o atributo de **Ei** que tem **Ec** como valor. Diz-se que um uso **<Eac,r,Eai>** abstrai **<Ec,r,Ei>** exatamente quando **Eac** abstrai **Ec** e **Eai** abstrai **Ei**.

Para descobrir todos os usos possíveis, é necessário olhar não apenas as definições explícitas de que um operador é componente de outro, mas também considerar as heranças que provocam usos implícitos. Dessa forma, Kautz estabelece que, sendo **Explicit** o conjunto de usos explícitos, pode-se definir o conjunto **Uses** das definições de composição implícitas e explícitas como a união de **Explicit** com o conjunto de triplas **<Ec,r, Ei>** tal que

(1) há um operador **Eu** tal que **<Ec,r,Eu>** $\in$ **Explicit**;

(2) há um uso **<Eac,r,Eau>** $\in$ **Explicit** que abstrai **<Ec,r,Eu> ;**

(3) **Eau** abstrai **Ei**;

(4) não haja **Ej** que especialize, seja igual ou abstraia **Ei** tal que exista um **<Ed,r,Ej>** ∈ **Explicit** que especialize **<Eac,r,Eau>**; e

(5) não haja **Eai** que abstraia **Ei** para o qual as condições (3) e (4) valham (com **Eai** no lugar de **Ei**).

Para explicar o funcionamento dessa definição, pode-se retornar ao exemplo das viagens de turismo. O operador *viajar* possui como especialização, além de *viajar_conta_própria*, o operador *viajar_pacote_turismo* que, por sua vez, possui como especializações *viajar_pacote_aereo* e *viajar_pacote_ônibus*. O atributo *f3* de *viajar* é *alugar_carro*. Nas especializações *viajar_pacote_aereo* e *viajar_pacote_ônibus* há redefinições do valor do atributo *f3* que recebeu, respectivamente, os valores *alugar_carro_luxo* e *alugar_carro_simples*, que são especializações de *alugar_carro*. As especializações de *viajar_conta_própria* (*viajar_avião_só* e *viajar_ônibus_só*) não possuem redefinição do atributo *f3* e simplesmente herdam o valor de *viajar_conta_própria* que por sua vez o herda de *viajar*. Imagine-se que se deseja saber os possíveis usos de *alugar_carro_luxo*. As definições explícitas indicam apenas o uso referente a *viajar_pacote_aereo*. Para se descobrir os demais, torna-se necessário examinar quais operadores atendem às condições acima.

A primeira condição estabelece a necessidade de existir um operador **Eu** que seja explicitamente usuário do operador **Ec**, no caso *alugar_carro_luxo*. Sendo assim, **Eu** pode ser associado ao operador *viajar_pacote_aereo*.

A segunda condição fala de um uso explícito **<Eac, r, Eau>** que abstraia <*alugar_carro_luxo*, *f3*, *viajar_pacote_aereo*>. Pode-se verificar que <*alugar_carro*, *f3*, *viajar*> atende a essa condição.

A terceira condição estabelece que o usuário **Ei** deve ser uma especialização de **Eau** (*viajar*). Os operadores *viajar_conta_própria*, *viajar_ônibus_só*, *viajar_avião_só*, *viajar_pacote_turismo*, *viajar_pacote_aereo* e *viajar_pacote_ônibus* atendem a essa condição e portanto são candidatos a usuários implícitos de *alugar_carro_luxo.*

A quarta condição estabelece que não haja nenhuma redefinição do atributo *f3* nos candidatos que atendem à terceira condição, nem em abstrações ou especializações suas. Sendo assim, *viajar_pacote_turismo*, *viajar_pacote_aereo* e *viajar_pacote_ônibus* podem ser descartados.

A quinta condição descarta *viajar_ônibus_só* e *viajar_avião_só*, uma vez que as condições terceira e quarta são satisfeitas para *viajar_conta_própria*. Sendo assim, além do uso explícito <*alugar_carro_luxo*,*f3*,*viajar_pacote_aereo*>, há o uso ímplicito

<*alugar_carro_luxo*,*f3*,*viajar_conta_própria*>. Isso representa a idéia de que "quem aluga um carro de luxo pode estar viajando por um pacote aéreo ou viajando por conta própria".

A verificação dos possíveis usos de um operador é feita, pelo programa, dinamicamente, por um predicado que traduz as regras acima para o Prolog.

## 2.2. A Construção de um Grafo de Explicação

O algoritmo recursivo que constrói um grafo para uma dada observação recebe como entrada um operador a ser explicado (**Ec**) e uma lista de componentes ( duplas *(atributo, valor)*) que esse operador deve ter (**parameters**). Logicamente, na primeira chamada, essa lista é vazia. Além disso, também são fornecidas uma lista dos tipos de operadores visitados até o momento na movimentação ao longo da hierarquia de abstrações (**visited**) e as direções ao longo dessa hierarquia a serem seguidas nas novas chamadas (**direction** - com o valor **up**, admitindo abstrações, o valor **down**, admitindo especializações, ou com ambos os valores). O retorno fornecido é o número do nodo correspondente ao operador e uma nova lista de tipos de operadores visitados. O grafo é continuamente modificado ao longo das chamadas sucessivas. Como a implementação foi feita em Prolog, que não possui variáveis globais, o predicado que implementa tal algoritmo possui grafos de entrada e saída.

A primeira ação executada é a inclusão de **Ec** em **visited**. Após isso, verifica-se se um nodo correspondente ao mesmo tipo de **Ec** já faz parte do grafo. Em caso afirmativo, o número do nodo é retornado. Caso contrário, um novo nodo é criado para conter o operador fornecido. No caso do exemplo das viagens de turismo, na primeira chamada, ao ser fornecido o operador *alugar_carro_luxo*, a lista **visited** está vazia porque nenhuma chamada foi feita ainda. Como o operador não pode ser encontrado em **visited**, um novo nodo é criado para inseri-lo no grafo. Esse teste é necessário porque pode já ter sido criado um nodo correspondente ao operador fornecido.

Quando o operador fornecido é o *end*, então basta retornar o número do nodo criado, senão a explicação deve continuar através de chamadas recursivas.

Para cada uso **<Ec,r,Eu>** do operador fornecido, se esse uso não abstrai ou especializa algum uso de algum dos membros de **visited**, então é feita uma chamada, onde o operador a ser explicado é **Eu**. A lista de componentes (**parameters**) tem como elemento apenas a dupla **<r,N>**, onde **N** é o número do nodo criado para **Ec**, a lista de nodos visitados é vazia e a lista de direções possíveis admite movimentos tanto para cima quanto para baixo ao longo da hierarquia de abstrações. No caso do exemplo, é feita uma chamada para o usuário

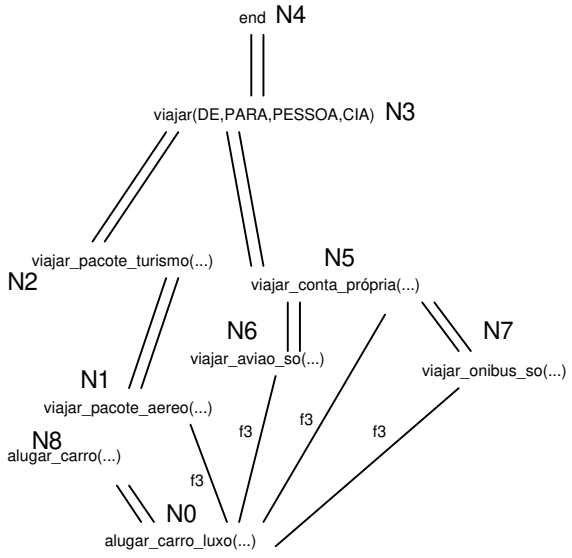
*viajar_pacote_aereo* e outra para o usuário *viajar_conta_própria*. O teste em relação a **visited** faz com que não seja incluído no grafo, no decorrer das futuras chamadas, o uso <*viajar*,*f3*,*alugar_carro*>, uma vez que já foi incluído o uso mais especializado <*viajar_pacote_aereo*, *f3*, *alugar_carro_luxo*>.

Se a movimentação para baixo ao longo da hierarquia de abstrações foi ordenada, então é feita recursivamente uma chamada para cada especialização direta **Esc** de **Ec**. Nessas novas chamadas, **parameters** tem o mesmo valor correspondente à chamada feita para **Ec**, **visited** é continuamente atualizado e a direção admitida nas movimentações é apenas para baixo. No caso do exemplo, na chamada para a explicação de *viajar_conta_própria*, essa parte do código é responsável pelas chamadas recursivas referentes aos operadores *viajar_avião_só* e *viajar_ônibus_só*.

Do mesmo modo, se a movimentação para cima foi ordenada, é feita uma chamada recursiva para a abstração direta **Eac** de **Ec**. Nessa chamada, a lista de componentes (**parameters**) é restrita aos atributos definidos para **Eac** ou operadores acima na hierarquia de abstrações. É ordenada a movimentação apenas para cima na hierarquia de abstrações e o valor de **visited** é também atualizado. Após a chamada, o arco de abstração **<M,N>** é incluído no grafo, onde **M** e **N** são, respectivamente, os números dos nodos correspondentes a **Eac** e **Ec**. No exemplo, na chamada para a explicação do operador *viajar_pacote_aereo*, essa parte do código se encarrega de fazer uma chamada para a explicação de *viajar_pacote_turismo*.

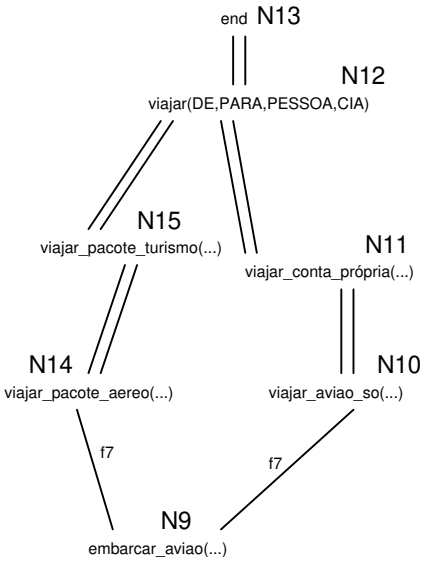
Por fim, o número **N** do nodo correspondente a **Ec** e o novo valor de **visited** são retornados.

O grafo de explicação abaixo é o resultado da explicação de *alugar_carro_luxo*. Os números dos nodos correspondem à seqüência de chamadas recursivas.



## 2.3. O Casamento dos Grafos

Quando existe mais de um operador observado, torna-se necessário unificar os grafos correspondentes a operadores distintos. Para isso é aplicado um algoritmo de casamento entre dois grafos que constrói um novo grafo eliminando as especializações alternativas de *end* que não são completamente compatíveis. O algoritmo de casamento é aplicado seguidas vezes até se obter um grafo que contenha todas as observações. Para ilustrar esse algoritmo, pode-se imaginar que, além de uma observação *alugar_carro_luxo*, haja uma observação *embarcar_avião*. Nesse caso, o algoritmo da subseção 2.2 forneceria o seguinte grafo a ser casado com o grafo de *alugar_carro_luxo*:



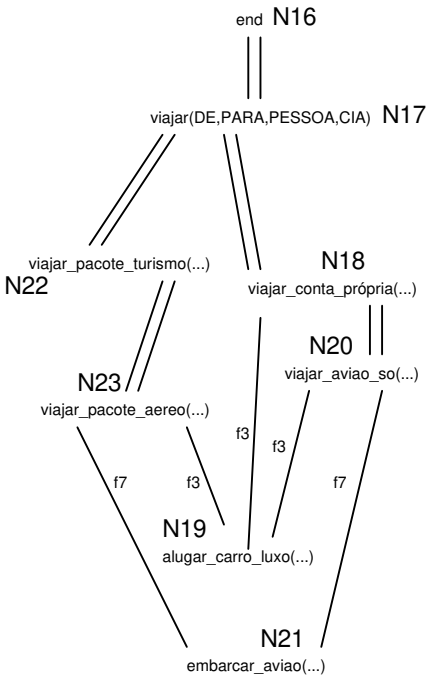

O algoritmo de casamento é também recursivo e utiliza uma tabela onde, para cada par de de números de nodos **n1** e **n2** já casado, é armazenado o número do nodo resultante **n3**. O algoritmo é chamado inicialmente para os nodos correspondentes aos operadores *end* dos dois grafos, no caso do exemplo, os nodos N4 e N13.

Primeiramente, verifica-se se os nodos já foram casados, consultando-se a tabela. Nesse caso, o valor consultado é retornado, caso contrário o algoritmo prossegue. Descobre-se dentre os dois nodos **n1** e **n2** qual deles corresponde a um tipo de operador mais especializado para que seja esse o tipo **typen3** do nodo **n3** a ser criado no grafo resultante. Se o tipo dos nodos for incompatível, o casamento falha. No caso do exemplo, não ocorre a necessidade de casamento entre dois nodos com nível de especialização diferentes. Isso pode ocorrer quando, dentre as observações, existem dois operadores, onde um operador é a especialização de outro. Um exemplo desse caso poderia ser o casamento de dois grafos onde um tenha sido construído para a observação *alugar_carro* e outro para a observação *alugar_carro_luxo*.

Já se sabendo o tipo **typen3**, verifica-se para cada atributo **r** desse tipo se os nodos que estão sendo casados possuem valor. Se ambos possuem, o algoritmo de casamento é chamado recursivamente para os mesmos nodos. Se apenas um dos nodos possui valor para esse atributo, a sua estrutura é copiada no novo grafo. Se nenhum deles possui, então o atributo não faz parte do novo grafo. O número do nodo fornecido como resultado, seja do casamento dos valores contidos nos dois grafos, seja pela cópia da estrutura correspondente ao valor em um grafo, é então inserido como valor do atributo **r** em questão para o nodo **n3**. No caso do exemplo, não ocorre nenhuma necessidade de casamento dos atributos porque o único arco de composição de um grafo (*f3* no grafo de *alugar_carro_luxo*) refere-se a um atributo diferente do referido pelo único arco de composição do outro grafo (*f7* no grafo de *embarcar_avião*). Portanto, as estruturas correspondentes a esses atributos são simplesmente copiadas no grafo resultante.

Se **n1** possui especializações no primeiro grafo ou **n2** as possui no segundo grafo, torna-se necessário fazer o casamento dessas especializações. Note-se que basta que um deles possua especializações para que isso seja necessário. Se um dos nodos possui especializações e o outro não, então dever-se-á tentar casar as alternativas do que as possui com o outro nodo. As alternativas de casamento que são bem sucedidas fazem com que os arcos de abstração **<n3,A3>**, onde **A3** é o resultado do casamento das alternativas, sejam inseridas no grafo resultante. Se nenhum casamento de alternativas for possível, então o casamento de **n1** com **n2** falha. No exemplo, essa parte do código é que descarta a alternativa de *viajar_conta_própria* denominada *viajar_ônibus_só*, pois ela está contida apenas no primeiro grafo.

Por fim, o valor **n3** é retornado como resultado do casamento de **n1** e **n2**. O grafo a seguir mostra o resultado do casamento dos grafos do exemplo.




# 3. A Interpretação dos Resultados do Reconhecimento

Após a obtenção de um grafo que explica um conjunto de observações fornecido pelo usuário, tem-se como tarefa identificar nesse grafo quais são os planos que correspondem a intenções finais do usuário. Admite-se, para isso, que o usuário deseja executar **planos concretos**, ou seja, especializações do operador *end* que, por sua vez, não possuem especializações. No grafo resultante apresentado na subseção 2.3, são planos concretos apenas os operadores *viajar_avião_só* e *viajar_pacote_aereo*.

Uma vez identificado um plano concreto, torna-se necessário instanciar, coerentemente com o reconhecimento, todas as variáveis contidas na sua definição, pois essas variáveis podem estar presentes tanto nas precondições a serem testadas quanto nos efeitos da aplicação do plano. Deve-se notar que as variáveis contidas no termo que identifica o plano já são instanciadas quando se recupera, na biblioteca de planos, a "forma" correspondente ao plano, porém as instanciações correspondentes a variáveis contidas apenas nos componentes precisam ser recuperadas, consultando-se o grafo. A necessidade desse processo de instanciação pode ser ilustrada com o exemplo do plano concreto reconhecido *viajar_pacote_aereo.* Ele é identificado pelo termo *viajar_pacote_aereo(DE,PARA,PESSOA,CIA)* e possui as precondições *ter_muito_dinheiro(PESSOA)*, *haver_conducao_p_aeroporto*, *estar_em(PESSOA,DE)* e *possui_lugar(VOO).* A variável *VOO* não faz parte do termo que identifica *viajar_pacote_aereo*, mas é citada no seu componente *f7*

(*embarcar_aviao(DE,PARA,PESSOA,CIA,VOO)).* Quando se recupera a "forma" de *viajar_pacote_aereo*, a variável *VOO* permanece livre, mas ela precisa ser instanciada com o mesmo valor usado na observação *embarcar_avião* para que a precondição seja corretamente testada. Por isso, é necessário reexaminar o gráfico gerado como resultado do reconhecimento.

Após a instanciação coerente de todas as variáveis, testam-se as precondições. Para que uma precodição seja considerada válida é necessário que o seu valor seja um fato, básico ou dedutível, do banco de dados. Se todas as precondições são válidas, o plano pode ser executado e o que resta fazer é comunicar o resultado ao usuário. Se alguma precondição falha, então o gerador de planos pode ser útil para se chegar a uma solução satisfatória.

A primeira hipótese de utilização do gerador é tentar adaptar o plano reconhecido, de forma que com a inclusão de alguns operadores e/ou restrições os efeitos primários do plano reconhecido possam ser obtidos. A segunda hipótese é gerar um plano absolutamente novo que tem como objetivo gerar os efeitos primários do plano reconhecido. Para que a primeira hipótese pudesse ser adotada, tornou-se necessário que os planos concretos da biblioteca contivessem todas as informações correspondentes a um plano ABTWEAK. Tanto na adaptação quanto na geração de novos planos, a entrada do gerador é preparada coletando-se todos os fatos contidos no banco de dados, para constituírem o estado inicial para o planejamento. A escolha entre geração e adaptação é deixada sempre ao usuário.

# 4. A Geração de Planos

Como já foi mencionado, a geração de planos é feita através de uma versão adaptada do planejador hierárquico não-linear ABTWEAK. A adaptação consistiu em fazer com que o planejamento pudesse se iniciar tendo como ponto de partida um plano da biblioteca a ser adaptado. Foram feitas também modificações na forma de consulta ao conteúdo dos operadores da biblioteca, pois a estrutura desses operadores passou a conter tanto definições necessárias para o reconhecimento quanto para a geração de planos.

Um plano ABTWEAK é definido por operadores que não precisam estar completamente ordenados em uma seqüência. Durante o planejamento, são definidas relações de ordem apenas quando se torna necessário. Sendo assim, é possível que, para a satisfação de uma meta, o ABTWEAK forneça apenas uma ordem parcial. Além dos operadores e das relações de ordem, fazem parte do plano restrições que obrigam que duas variáveis referenciem um mesmo objeto (correferência) ou que não referenciem um mesmo objeto (impossibilidade de correferência).

Para exemplificar a definição de ordem parcial e de restrições de referência das variáveis, pode-se retornar ao exemplo da seção 2. Os operadores *comprar_passagem*, *ir_a_aeroporto* e *embarcar_avião* são componentes do plano *viajar_pacote_aereo.* Pode não existir nenhuma restrição de ordem relacionando *ir_a_aeroporto* com *comprar_passagem*, mas deve-se especificar que ambos ocorram antes do operador *embarcar_avião*. Como exemplo de restrição de impossibilidade de correferência de variáveis a um objeto, pode-se citar a restrição de que, no plano *viajar_pacote_aereo(DE,PARA,PESSOA,CIA),* as variáveis *DE* e *PARA* tenham valores diferentes.

A hierarquia de precondições é implementada através da atribuição de níveis de criticalidade a cada precondição dos operadores. O planejamento procura fazer com que todas as precondições de nível mais alto sejam satisfeitas, antes de serem consideradas precondições de um nível inferior. O processo termina quando são satisfeitas as precondições do nível zero. Para que esse processo possa ser efetuado, cada plano possui uma indicação do nível de criticalidade já resolvido por ele.

Como exemplo de hierarquia de precondições, pode-se estabelecer que as precondições do tipo *estar_em(PESSOA,LUGAR)* tenham prioridade sobre as demais. Sendo assim, é feito um planejamento completo relativo à satisfação desse tipo de precondição, antes que os demais tipos de precondições tenham que ser considerados.

A verificação de satisfação de um plano é feita, como no TWEAK, através do "Modal Truth Criterion" (MTC), só que atendo-se apenas às precondições com criticalidade acima do nível do plano que está sendo testado. O MTC tem a virtude de testar a validade de um plano em um tempo polinomial como uma função do número de operadores. Basicamente, o MTC consiste em verificar se, para cada precondição de um operador contido no plano, existe um operador que obrigatoriamente a estabelece ("establisher") e se ela não pode ser removida por um outro operador ("clobberer").

Para que uma precondição seja satisfeita, o algoritmo gera todas as possíveis soluções para resolvê-la, utilizando operadores já presentes no plano ou incluindo novos operadores como possíveis "establishers". Quando um operador é colocado como "establisher" de uma certa precondição de um operador ("user"), são incluídas, na alternativa de plano que está sendo gerado, restrições temporais ("establisher" antes do "user") e de correferência a variáveis.

Para uma dada alternativa de "establisher", contudo, pode existir um "clobberer" gerando um conflito. Na eliminação de conflitos existem várias alternativas de resolução:

1. Fazer com que o "clobberer" venha antes do "establisher".

2. Impor que o "clobberer" venha depois do "user".

3. Impor, através de restrições que impedem a correferência de variáveis, que o efeito gerado pelo "establisher" seja igual à negação do efeito do "clobberer" com o qual surgiu o conflito.

Durante o processo de planejamento, são geradas todas as alternativas para que uma dada precondição seja satisfeita, incluindo todas as formas de resolução de conflitos em cada passo. Essas alternativas dão origem a vários planos sucessores do plano original, que são incluídos em uma fila de prioridades, ordenada segundo o custo total estimado para se atingir a solução. Para se fazer essa estimativa, é necessária uma função heurística, que atribui um custo menor para a conclusão de planos com nível de criticalidade mais baixo, ou seja, que já tiveram as precondições com criticalidade mais alta resolvidas.

# 5. Estruturas de Dados

Para os usuários do Sistema, torna-se importante entender as estruturas de dados utilizadas na definição de operadores, de planos e da lista de criticalidades.

## 5.1. Estrutura dos Planos/Operadores da Biblioteca

Os operadores são definidos por uma lista Prolog com os seguintes elementos, na ordem:

- Identificador do operador, que, no caso das cópias dos operadores, contidas nos planos, possui um valor único, diferente de todos os identificadores de outros operadores do plano.

- O termo de identificação do operador que é um “functor” PROLOG contendo o nome e os parâmetros do operador na forma *nome(PAR1,PAR2,PAR3,...,PARn).*

- Uma lista de precondições do operador. As precondições afirmativas são definidas sob a forma de “functores”, contendo o nome e os parâmetros (variáveis ou constantes). Caso a precondição seja uma negação, ela é envolvida pelo functor “not”, da seguinte forma: *not(nome(PAR1,PAR2,...,PARn))*.

- Uma lista de efeitos gerados pelo operador, que possui a mesma estrutura da lista de precondições.

- O custo do operador.

- Uma lista de efeitos primários, gerados pelo operador, que possui a mesma estrutura da lista de precondições. Os efeitos primários são utilizados quando o uso deles está ativado (ver subseção 8.5). Nesse caso, a busca por um operador, que satisfaça uma certa condição, é feita olhando-se, em cada operador, essa lista e não a lista de todos os efeitos gerados por ele. Quando um plano é reconhecido, atribui-se como sua finalidade, tanto na geração de novo plano quanto na adaptação do plano reconhecido, a geração dos efeitos contidos nesta lista.

- Uma lista contendo, no caso de operadores complexos (planos), as duplas *(atributo, nome_do_componente(PAR1,PAR2,...,PARn))*.

- Uma lista contendo duplas de variáveis que não podem referenciar um mesmo objeto.

- Uma lista de restrições temporais, que estabelece a precedência entre os operadores componentes. Cada restrição é uma dupla contendo os atributos dos dois componentes.

A seguir é mostrado o exemplo da estrutura de um operador.

```
operator([ viajar_pacote_aereo,
                viajar_pacote_aereo(DE,PARA,PESSOA,CIA),
        [
                ter_muito_dinheiro(PESSOA),
                haver_conducao_p_aeroporto,
                estar_em(PESSOA,DE),
                possui_lugar(VOO)
        ],
        [
                conheceu(PESSOA,PARA),
                viajou_aviao(PESSOA),
                e_cliente(PESSOA,CIA)
        ],
        10,
        [
                conheceu(PESSOA,PARA),
                viajou_aviao(PESSOA),
                e_cliente(PESSOA,CIA)
        ],
        [
                (f3,alugar_carro_luxo(PESSOA)),
                (f6,ir_a_aeroporto(PESSOA,DE)),
                (f7,embarcar_aviao(DE,PARA,PESSOA,CIA,VOO))],

        [(DE,PARA),(DE,PESSOA),(DE,CIA),(PARA,PESSOA),
          (PARA,CIA), (PESSOA,CIA)],

        [(f6,f7),(f4,f7),(f7,f1),,(f1,f2),,(f3,f1)]

]).
```

## 5.2. Estrutura dos Planos ABTWEAK

Os planos, durante a geração de planos, assumem a estrutura de uma lista Prolog com os seguintes campos, na ordem:

- Custo do plano até o momento.
- Nível de criticalidade.
- Lista de submetas de níveis superiores a serem protegidas , quando o usuário estabelece a utilização de propriedade monotônica (ver subseção 8.1). Para cada submeta é armazenada uma lista de três elementos contendo o identificador do operador que a tem como precondição ("user"), a precondição e a lista de possíveis operadores que a estabelecem ("establishers"). A proteção consiste em garantir que um dos "establishers" continuará estabelecendo a precondição para o "user".
- Lista de operadores do plano.
- Lista de duplas de identificadores de operadores, relacionando a ordem de precedência.
- Listas de duplas de variáveis/constantes que não podem referenciar um mesmo objeto.
- Contadores do número de operadores definidos para o plano em cada nível da hierarquia. É uma lista de duplas, onde o primeiro elemento corresponde ao nível e o segundo ao respectivo número de operadores.
- Indicação se o plano está inválido, que é marcada quando alguma restrição sobre a ordem dos operadores ou sobre a impossibilidade de correferência de variáveis a um mesmo objeto é violada. Possui o valor "true" quando isso ocorre.

## 5.3. A Lista de Criticalidades

A lista de criticalidades é utilizada para definir a criticalidade das precondições. Ela é uma lista que contém tantos elementos quantos forem os K níveis, começando pela definição das precondições de nível K-1 e terminando pelas de nível zero. Cada elemento é uma lista cujo primeiro elemento é o nível que ele está definido e os demais são as precondições

associadas a esse nível. Na definição de uma precondição, utiliza-se a mesma sintaxe adotada nas listas de precondições e efeitos dos operadores. Os parâmetros das precondições, para efeito de definição da criticalidade, podem ser constantes ou o símbolo '$', que denota um valor não especificado . A seguir temos o exemplo da estrutura de uma lista de criticalidade.

```
[
    [2,is_block,('$'),clear(c),not(clear(c)),not(on(c,'$')),on(c,'$')],
    [1,clear(b),not(clear(b)),not(on(b,'$')),on(b,'$')],
    [0,clear('$'),not(clear('$')),not(on('$','$'),on('$','$')]
]
```

# 6. A Especificação do Contexto

A especificação dos operadores/planos que compõem a biblioteca, da função heurística, da lista de criticalidades e dos fatos a serem admitidos como o contexto atual do banco de dados deve ser feita através de um arquivo a ser consultado através do interpretador do ARITY PROLOG.

Os operadores são definidos através de um predicado unário de nome **operator**, que contém como argumento a definição do operador. Deve existir uma cláusula para cada operador.

A função heurística deve ser definida através de um predicado binário que recebe no primeiro argumento o plano e devolve no segundo argumento o seu custo estimado. Esse predicado deve ter o nome de **heuristic_function**.

A lista de criticalidades deve ser definida através do predicado unário **critical_list** que a contém como argumento.

Os fatos do banco de dados são definidos através do predicado unário **db** que contém o fato como seu argumento.

A hierarquia de abstrações entre os operadores é definida através do predicado binário **is_a**. O primeiro argumento é o operador especializado e o segundo a abstração. Os operadores são representados nesse predicado com o termo que o identifica, ou seja, na forma *nome(ARG1,ARG2,...)*. A presença dos argumentos na definição da herança é importante para a instanciação correta das variáveis durante o reconhecimento de planos.

O número máximo de operadores a serem admitidos em um plano deve ser definido através do predicado unário **solution_limit** e o número máximo de expansões de planos em sucessores através do predicado unário **expand_bound**.

# 7. A Utilização do Sistema

Para se iniciar a execução do sistema devem ser executados os seguintes passos:

- Consultar o arquivo de inicialização "init.ari", executando-se o comando:

  **[-init].**

- Consultar o arquivo de especificação do contexto, através do comando:

  **[-'NOME DO ARQUIVO']**

Executar o comando **load**.

Executar a rotina principal através do comando

**main(L_OBS)**, onde

L_OBS é a lista de observações a serem utililizadas no reconhecimento do plano. Cada operador deve ser especificado sob a forma: *nome(ARG1, ARG2, ...)*, onde os argumentos podem ser variáveis ou, mais comumente, constantes.

Se o SRGP reconhecer algum plano que se adeqüe às observações, então testa se as precondições para esse plano são válidas. Se forem, o sistema indica o plano reconhecido. Caso alguma precondição não seja válida, o SRGP comunica o plano reconhecido e as precondições não satisfeitas e pergunta se o usuário deseja que o plano seja adaptado. Caso ele não deseje isso, pode ainda pedir que um novo plano seja gerado, tendo como objetivo atingir os mesmos efeitos primários do plano reconhecido.

Após a geração ou adaptação de um plano, os resultados são apresentados ao usuário na seguinte forma:

- A lista de restrições temporais aparece contendo duplas de identificadores de operadores. Os operadores que já faziam parte do plano reconhecido têm como identificador o atributo do qual eles eram valor no plano reconhecido. Os operadores incluídos pelo ABTWEAK têm como identificador um número inteiro.

- Aparece então, para cada identificador, o termo que denota a respectiva aplicação de um operador.

- O operador com identificador **i** corresponde ao operador que, por convenção, gera todo o contexto atual.

- O operador com identificador **g** corresponde a um operador que, também por convenção, contém como precondições os objetivos a serem atingidos.

Como o SRGP possibilita o reconhecimento de vários planos alternativos, novas soluções podem ser solicitadas forçando-se o "bactracking", ou seja, digitando-se um ponto e vírgula após a apresentação de um resultado.

# 8. Parâmetros de Utilização do Gerador de Planos

O sistema oferece a possibilidade ao usuário de alterar o processamento do planejador através de alguns parâmetros. Esses parâmetros podem ser ativados através do predicado unário **set** e removidos através do predicado, também unário, **remove**. Para saber se um parâmetro está ativado basta executar um comando com o seu nome, uma vez que eles são colocados no banco de dados como fatos Prolog.

Os parâmetros e suas características são descritos a seguir.

## 8.1. mp_mode

Este parâmetro indica que se deve adotar a propriedade monotônica, ou seja, cada precondição estabelecida em níveis superiores da hierarquia deve continuar sendo estabelecida, quando se resolve um nível inferior, por um dos possíveis "establishers" que a estabeleciam no nível superior.

Essa propriedade tende a acelerar o funcionamento do planejador quando a hierarquia de precondições é bem definida. Por outro lado, quando a hierarquia não é boa, a eficiência diminui e não se tem garantia de que o planejamento chegue a um resultado.

## 8.2. drp_mode

Este parâmetro indica a adoção do modo "downard refinement property". Quando ele está ativado e se resolve o problema em um certo nível de abstração, segundo algum plano, a lista de prioridades, que contém os outros planos alternativos de solução no mesmo nível, é

colocada em uma pilha. A pesquisa passa a se concentrar apenas nos sucessores do plano resolvido nesse nível de abstração. Apenas quando se verifica a inexistência de soluções nos níveis inferiores, a lista dos planos alternativos é desempilhada para que a pesquisa continue.

### 8.3. abstract_goal_mode

Este parâmetro indica que, na busca de precondições a serem satisfeitas, deve-se aplicar abstração também às precondições contidas na meta principal ("GOAL"). Como essa situação é a normal, o Sistema é carregado inicialmente com este parâmetro ativado.

### 8.4. random_det_mode

Este parâmetro indica que a escolha de uma precondição não satisfeita para ser resolvida, em um certo nível de abstração, deve-se dar de forma aleatória. Quando não está ativado, procura-se a precondição seguindo a ordem inversa da inclusão de operadores no plano.

### 8.5. use_primary_effects_p

Este parâmetro indica que a busca de operadores que satisfaçam uma certa precondição deve escolher um operador que, além de estabelecê-la, a indique em sua lista de efeitos primários.

## 9. Um Exemplo Completo

A fim de ilustrar sua utilização, o Sistema é acompanhado de um arquivo chamado **EXEMPLO.ARI** que contém as definições referentes ao exemplo da seção 2, que trata de um universo de viagens de turismo.

No contexto especificado na seção 2 admitem-se como planos concretos: *trabalhar_no_aeroporto*, *viajar_pacote_aereo***,** *viajar_pacote_onibus***,** *viajar_aviao_so* e *viajar_onibus_so*.

Deve-se notar que o atributo *f3* de viajar, que possui o valor *alugar_carro* é redefinido em *viajar_pacote_aereo* e *viajar_pacote_onibus*. Dessa forma, indica-se que quem viaja por conta própria pode escolher qualquer tipo de carro para alugar, enquanto que quem viaja por pacote aéreo é obrigado a alugar um carro de luxo e quem viaja por pacote de ônibus é obrigado a alugar um carro simples.

Na próxima subseção, é apresentado o conteúdo do arquivo com a definição do contexto descrito na seção 2. Nas demais subseções, são mostradas telas correspondentes a diferentes situações na utilização do SRGP para o contexto descrito.

## 9.1. O Arquivo EXEMPLO.ARI

O conteúdo do arquivo **EXEMPLO.ARI** é mostrado a seguir. Deve-se notar, na definição dos planos concretos *viajar_pacote_aereo* e *viajar_pacote_onibus,* que eles possuem precondições, efeitos, efeitos primários e especificações sobre restrições temporais e correferências. Esses dados são necessários tanto para se verificar a aplicabilidade do plano quanto para adaptá-lo através do ABTWEAK. Os planos concretos *trabalhar_no_aeroporto*, *viajar_aviao_so* e *viajar_onibus_so* não têm precondições especificadas, por isso, quando reconhecidos, são sempre "aplicáveis".

```
/*  DEFINICOES HIERARQUICAS PARA RECONHECIMENTO DE PLANOS */

is_a(viajar(_,_,_,_), end).

is_a(viajar_conta_propria(DE,PARA,PESSOA,CIA), viajar(DE,PARA,PESSOA,CIA)).

is_a(viajar_pacote_turismo(DE,PARA,PESSOA,CIA), viajar(DE,PARA,PESSOA,CIA)).

is_a(viajar_pacote_aereo(DE,PARA,PESSOA,CIA), viajar_pacote_turismo(DE,PARA,PESSOA,CIA)).

is_a(viajar_pacote_onibus(DE,PARA,PESSOA,CIA), viajar_pacote_turismo(DE,PARA,PESSOA,CIA)).

is_a(viajar_aviao_so(DE,PARA,PESSOA,CIA), viajar_conta_propria(DE,PARA,PESSOA,CIA)).

is_a(viajar_onibus_so(DE,PARA,PESSOA,CIA), viajar_conta_propria(DE,PARA,PESSOA,CIA)).

is_a(trabalhar_no_aeroporto(_),end).

is_a(alugar_carro_simples(PESSOA), alugar_carro(PESSOA)).

is_a(alugar_carro_luxo(PESSOA), alugar_carro(PESSOA)).


/* DEFINICOES DOS OPERADORES */

operator([ viajar,
           viajar(DE,PARA,PESSOA,CIA),
           [],
           [],
           10,
           [],
           [(f1,visitar(PARA,PESSOA)),
            (f2,retornar(PARA,DE,PESSOA)),
            (f3,alugar_carro(PESSOA))
           ],
           [],
           []
         ]).


operator([ viajar_pacote_turismo,
           viajar_pacote_turismo(DE,PARA,PESSOA,CIA),
           [],
           [],
           10,
           [],
           [(f4,comprar_pacote(DE,PARA,PESSOA,CIA))],
           [],
           []
         ]).


operator([ viajar_conta_propria,
           viajar_conta_propria(DE,PARA,PESSOA,CIA),
           [],
           [],
           10,
```

```
            [],
            [(f4,comprar_passagem(DE,PARA,PESSOA,CIA)),
             (f5,se_informar_sobre(PESSOA,PARA))],
            [],
            []
          ]).


operator([ viajar_aviao_so,
           viajar_aviao_so(DE,PARA,PESSOA,CIA),
           [],
           [],
           10,
           [
            conheceu(PESSOA,PARA),
             viajou_aviao(PESSOA)
         ],
           [(f6,ir_a_aeroporto(PESSOA,DE)),
            (f7,embarcar_aviao(DE,PARA,PESSOA,CIA,VOO))],
           [],
           []
         ]).


operator([ viajar_onibus_so,
           viajar_onibus_so(DE,PARA,PESSOA,CIA),
           [],
           [],
           10,
           [
             conheceu(PESSOA,PARA),
             viajou_onibus(PESSOA)
         ],
           [(f7,embarcar_onibus(DE,PARA,PESSOA,CIA))],
           [],
           []
         ]).


operator([ viajar_pacote_aereo,
           viajar_pacote_aereo(DE,PARA,PESSOA,CIA),
           [
             ter_muito_dinheiro(PESSOA),
             haver_conducao_p_aeroporto,
             estar_em(PESSOA,DE),
             possui_lugar(VOO)
           ],
           [
             conheceu(PESSOA,PARA),
             viajou_aviao(PESSOA),
             e_cliente(PESSOA,CIA)
           ],
           10,
           [
             conheceu(PESSOA,PARA),
             viajou_aviao(PESSOA),
             e_cliente(PESSOA,CIA)
           ],
           [
            (f3,alugar_carro_luxo(PESSOA)),
            (f6,ir_a_aeroporto(PESSOA,DE)),
            (f7,embarcar_aviao(DE,PARA,PESSOA,CIA,VOO))],
           [(DE,PARA),(DE,PESSOA),(DE,CIA),(PARA,PESSOA),
            (PARA,CIA),(PESSOA,CIA)],
           [(f6,f7),(f4,f7),(f7,f1),(f1,f2),(f3,f1)]
         ]).


operator([ viajar_pacote_onibus,
           viajar_pacote_onibus(DE,PARA,PESSOA,CIA),
           [
             ter_um_pouco_de_dinheiro(PESSOA),
             estar_em(PESSOA,DE)
         ],
           [
             conheceu(PESSOA,PARA),
             viajou_onibus(PESSOA),
             e_cliente(PESSOA,CIA)
         ],
           10,
           [
             conheceu(PESSOA,PARA),
             viajou_onibus(PESSOA),
             e_cliente(PESSOA,CIA)
         ],
           [
            (f3,alugar_carro_simples(PESSOA)),
            (f7,embarcar_onibus(DE,PARA,PESSOA,CIA))],
           [(DE,PARA),(DE,PESSOA),(DE,CIA),(PARA,PESSOA),
            (PARA,CIA),(PESSOA,CIA)],
```

```
            [(f4,f7),(f7,f1),(f1,f2),(f3,f1)]
          ]).


operator([ trabalhar_no_aeroporto,
           trabalhar_no_aeroporto(PESSOA),
           [],
           [],
           10,
           [],
           [
           (f10,ir_a_aeroporto(PESSOA,DE))
           ],
           [],
           []
          ]).


operator([ ir_a_aeroporto,
           ir_a_aeroporto(PESSOA,DE),
           [
             estar_em(PESSOA,DE),
             haver_conducao_p_aeroporto
          ],
           [
             estar_no_aeroporto(PESSOA,DE)
           ],
           10,
           [],
           [
           (f1,pegar_conducao_p_aeroporto(PESSOA,DE))
           ],
           [],
           []
          ]).

operator([ embarcar_aviao,
           embarcar_aviao(DE,PARA,PESSOA,CIA,VOO),
           [
              estar_em(PESSOA,DE),
              estar_no_aeroporto(PESSOA,DE),
              ter_embarque_autorizado(DE,PARA,PESSOA,CIA),
              possui_lugar(VOO)
          ],
           [
              not(estar_em(PESSOA,DE)),
              not(estar_no_aeroporto(PESSOA,DE)),
              not(ter_embarque_autorizado(DE,PARA,PESSOA,CIA)),
              estar_em(PESSOA,PARA),
              viajou_aviao(PESSOA),
              estar_viajando(PESSOA,DE)
          ],
           10,
           [],
           [],
           [],
           []
          ]).


operator([ embarcar_onibus,
           embarcar_onibus(DE,PARA,PESSOA,CIA),
           [
             estar_em(PESSOA,DE),
              ter_embarque_autorizado(DE,PARA,PESSOA,CIA)
          ],
           [
              not(estar_em(PESSOA,DE)),
              not(ter_embarque_autorizado(DE,PARA,PESSOA,CIA)),
              estar_em(PESSOA,PARA),
              viajou_onibus(PESSOA),
              estar_viajando(PESSOA,DE)
          ],
           10,
           [],
           [],
           [],
           []
          ]).


operator([ alugar_carro_simples,
           alugar_carro_simples(PESSOA),
           [
             ter_um_pouco_de_dinheiro(PESSOA)
          ],
           [
             ter_meio_de_transporte(PESSOA)
```

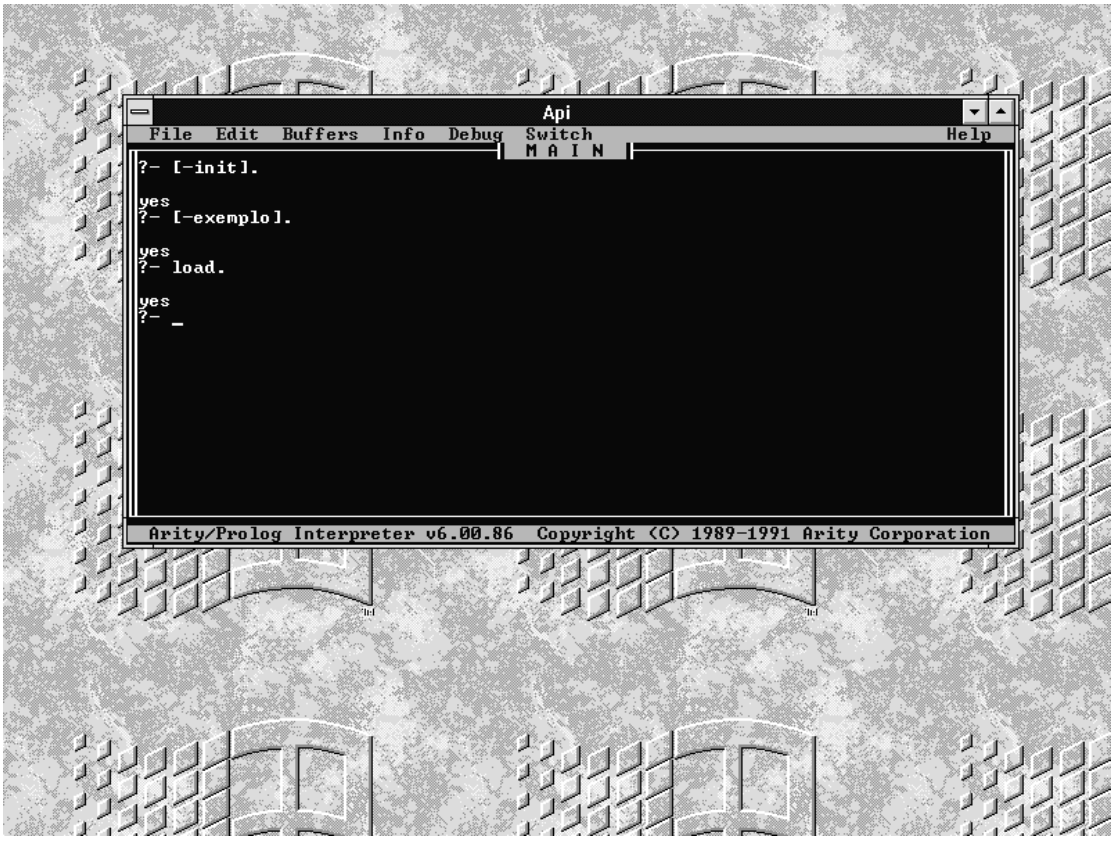
```
           ],
            10,
            [],
            [],
            [],
            []
          ]).

operator([ alugar_carro_luxo,
            alugar_carro_luxo(PESSOA),
            [
              ter_muito_dinheiro(PESSOA)
            ],
            [
              ter_meio_de_transporte(PESSOA)
            ],
            10,
            [],
            [],
            [],
            []
          ]).


operator([ se_informar_sobre,
            se_informar_sobre(PESSOA,PARA),
            [],
            [
              ter_inform_sobre(PESSOA,PARA)
            ],
            10,
            [],
            [],
            [],
            []
          ]).

operator([ comprar_pacote,
            comprar_pacote(DE,PARA,PESSOA,CIA),
            [
              ter_um_pouco_de_dinheiro(PESSOA)
            ],
            [
               ter_embarque_autorizado(DE,PARA,PESSOA,CIA),
               ter_inform_sobre(PESSOA,PARA),
               e_cliente(PESSOA,CIA)
            ],
            10,
            [],
            [],
            [],
            []
          ]).

operator([ comprar_passagem,
            comprar_passagem(DE,PARA,PESSOA,CIA),
            [
              ter_um_pouco_de_dinheiro(PESSOA)
            ],
            [
               ter_embarque_autorizado(DE,PARA,PESSOA,CIA)
            ],
            10,
            [],
            [],
            [],
            []
          ])

operator([ pegar_conducao_p_aeroporto,
            pegar_conducao_p_aeroporto(PESSOA,DE),
            [
              estar_em(PESSOA,DE),
              haver_conducao_p_aeroporto
            ],
            [
              estar_no_aeroporto(PESSOA,DE)
            ],
            10,
            [],
            [],
            [],
            []
          ]).



operator([ visitar,
            visitar(PARA,PESSOA),
            [
```

```
               estar_em(PESSOA,PARA),
               ter_inform_sobre(PESSOA,PARA),
               ter_meio_de_transporte(PESSOA)
            ],
            [
               conheceu(PESSOA,PARA)
            ],
            10,
            [],
            [],
            [],
            []
         ]).


operator([ retornar,
            retornar(PARA,DE,PESSOA),
            [
              estar_em(PESSOA,PARA),
               estar_viajando(PESSOA,DE)
            ],
            [
               not(estar_em(PESSOA,PARA)),
               estar_em(PESSOA,DE),
               not(estar_viajando(PESSOA,DE))
            ],
            10,
            [],
            [],
            [],
            []
         ]).

operator([ viajar_de_carona,
            viajar_de_carona(DE,PARA,PESSOA),
            [
               estar_em(PESSOA,DE),
               ter_carona(DE,PARA)
            ],
            [
               not(estar_em(PESSOA,DE)),
               estar_em(PESSOA,PARA)
            ],
            10,
            [],
            [],
            [],
            []
         ]).




operator([ fazer_qualquer_viagem_aviao,
            fazer_qualquer_viagem_aviao(PESSOA),
            [
               ter_um_pouco_de_dinheiro(PESSOA)
            ],
            [
               viajou_aviao(PESSOA)
            ],
            10,
            [],
            [],
            [],
            []
         ]).




/* DEFINICAO DA LISTA DE CRITICALIDADES */

critical_list([
                  [1,estar_no_aeroporto('$','$'),
                   estar_em('$','$')
                  ],
                  [0,conheceu('$','$'),
                     ter_muito_dinheiro('$'),
                     ter_um_pouco_de_dinheiro('$'),
                     ter_carona('$','$'),
                     viajou_aviao('$'),
                     viajou_onibus('$'),
                     haver_conducao_p_aeroporto,
                     e_cliente('$','$'),
                     ter_embarque_autorizado('$','$','$','$'),
                     estar_viajando('$','$'),
```

```
                    ter_meio_de_transporte('$'),
                    ter_inform_sobre('$','$'),
                    possui_lugar('$')
                ]
            ]).

/* DEFINICAO DA FUNCAO HEURISTICA */

heuristic_function(STATE,VAL):-
      plan_kval(STATE,KVAL),
      ith([-10,0],KVAL,VAL).

/*  " BANCO DE DADOS "   */

db(haver_conducao_p_aeroporto).
db(ter_muito_dinheiro(jose)).
db(ter_um_pouco_de_dinheiro(X)):-db(ter_muito_dinheiro(X)).
db(estar_em(jose,vitoria)).
db(ter_carona(vitoria,rio)).
db(possui_lugar(_)).


/*  PARAMETROS DE OPERACAO DO GERADOR DE PLANOS */

expand_bound(100).
solution_limit(100).
```

## 9.2. O Carregamento do Sistema e do Arquivo EXEMPLO.ARI

A tela a seguir mostra os comandos correspondentes ao carregamento do Sistema e ao carregamento do arquivo com o contexto.



## 9.3. O Reconhecimento de um Plano Aplicável

Neste exemplo, o usuário entra com as observações: *ir_a_aeroporto(jose,vitoria)*, *comprar_pacote(vitoria,sao_paulo,jose,cia1)* e *alugar_carro_luxo(jose)*. É identificado o plano concreto *viajar_pacote_aereo(vitoria,sao_paulo,jose,cia1)*, que é aplicável porque as precondições *ter_muito_dinheiro(jose)*, *estar_em(jose,vitoria)* e *haver_condução_p_aeroporto* são fatos do banco de dados. A tela a seguir mostra a situação.

## 9.4 A Adaptação de um Plano

Nesta nova situação, alteram-se as observações para *ir_a_aeroporto(jose,rio)*, *comprar_pacote(rio,sao_paulo,jose,cia1)* e *alugar_carro_luxo(jose).* O plano concreto detectado é *viajar_pacote_aereo(rio,sao_paulo,jose,cia1)*, porém esse plano não é aplicável porque a precondição *estar_em(jose,rio)* não é satisfeita, uma vez que *jose* se encontra em *vitoria*. O usuário, nesse caso, optou por adaptar o plano reconhecido, o que gerou um novo plano que, além dos componentes do plano reconhecido, tem um operador *viajar_de_carona(jose,vitoria,rio)* antes do operador *ir_a_aeroporto(jose,rio)*. As duas telas a seguir mostram a situação.

Api
File Edit Buffers Info Debug Switch Help
M A I N
Plano Abtweak adaptado:
Relacoes de Ordem
[(3 , f6),(i , 3),(f2 , g),(i , f4),(i , f3),(f6 , f7),(f4 , f7),(f7 , f1),(f1
, f2),(f3 , f1)]
Identificacao dos Operadores
3:viajar_de_carona(vitoria,rio,jose)
f3:alugar_carro_luxo(jose)
f6:ir_a_aeroporto(jose,rio)
f7:embarcar_aviao(rio,sao_paulo,jose,cia1,_7AC4)
f4:comprar_pacote(rio,sao_paulo,jose,cia1)
f1:visitar(sao_paulo,jose)
f2:retornar(sao_paulo,rio,jose)
->;
Fim dos resultados para o plano.
no
?-
Arity/Prolog Interpreter v6.00.86 Copyright (C) 1989-1991 Arity Corporation

## 9.5 A Geração de um Novo Plano

Agora, para a mesma situação apresentada na subseção 9.4, o usuário opta por criar um plano novo que gere os mesmos efeitos do plano reconhecido, ou seja, *conheceu(jose,sao_paulo)*, *viajou_de_aviao(jose)* e *e_cliente(jose,cia1)*. O resultado é a geração de um plano com o único operador *viajar_pacote_aereo(vitoria,sao_paulo,jose,cia1)*, que corresponde a um vôo partindo da própria cidade em que a pessoa se encontra. Esse operador produz os mesmos efeitos do plano detectado e é aplicável. É interessante notar que, nesse caso, o plano gerado parece bem mais adequado que o plano adaptado da subseção anterior.

Api
File Edit Buffers Info Debug Switch Help
M A I N
As precondicoes a seguir nao estao satisfeitas:
[estar_em(jose,rio)]
O plano detectado tem os seguintes efeitos primarios:
[conheceu(jose,sao_paulo),viajou_aviao(jose),e_cliente(jose,cia1)]
Tenta adaptar o plano ? (s/n)n
Tenta gerar um novo plano ? (s/n)s
Plano Abtweak gerado:
Relacoes de Ordem
[(i , 0),(0 , g)]
Identificacao dos Operadores
0:viajar_pacote_aereo(vitoria,sao_paulo,jose,cia1)
->;
Fim dos resultados para o plano.
no
?-
Arity/Prolog Interpreter v6.00.86 Copyright (C) 1989-1991 Arity Corporation

## 9.6 Modificando-se os Custos dos Operadores

Para se ter uma idéia da influência dos custos dos operadores no planejamento, o usuário pode editar o arquivo EXEMPLO.ARI e diminuir de 10 para 1 os custos dos operadores *fazer_qualquer_viagem_aviao(PESSOA*) e *viajar_pacote_onibus(DE,PARA,PESSOA,CIA).* Repete-se então o procedimento da subseção anterior. O resultado gerado passa então a privilegiar a viagem por um pacote de ônibus que gera os mesmos efeitos da viagem aérea, exceto o efeito *viajou_aviao(jose)*. Para gerar esse último efeito, foi inserido no plano o operador *fazer_qualquer_viagem_avião(jose)*. Note-se que não há nenhuma precedência entre os dois operadores.

# 10. Trabalhos Futuros

O SRGP, nesta versão, tem uma série de pontos a serem aperfeiçoados e abre a possibilidade de se estudar uma série de pontos importantes.

Um primeiro aperfeiçoamento refere-se à performance do gerador de planos. Foi mantida a estrutura original do programa em Lisp, que continha o defeito de chamar duas vezes a rotina que implementa o MTC quando, na realidade, com uma reestruturação, apenas uma chamada passaria a ser necessária. Como essa rotina tem um custo alto na execução, essa mudança deve trazer uma economia significativa.

Também em relação à performance, torna-se necessário criar mecanismos para que o estado corrente do banco de dados tenha um tratamento especial no gerador de planos, evitando-se a necessidade de se criar um operador fictício que insere todo o contexto atual. Para qualquer banco de dados com um conjunto um pouco maior de fatos, torna-se inviável coletar todos eles e inseri-los como efeitos de um operador de inicialização.

Dentre os pontos importantes que podem ser estudados com o auxílio do SRGP temos os seguintes:

- o exame das possíveis interações entre a hierarquia de operadores adotada pelo reconhecedor e a hierarquia de precondições do gerador;
- a melhor forma de atribuir custos aos operadores no planejamento;
- o armazenamento de planos gerados ou adaptados pelo sistema como parte da biblioteca de planos/operadores;
- o tratamento de observações que não são apenas conjunções de operadores, mas que incluam disjunções e restrições temporais;
- a interligação do planejamento com bancos de dados ativos;
- a integração prática do Sistema a ambientes cooperativos; e
- a aplicação do Sistema a bancos de dados textuais.

## REFERÊNCIAS BIBLIOGRÁFICAS


[Ari88] Arity Corporation. "Arity/Prolog Language Reference Manual". Concord, MA, EUA. 1988.

[Ch87] Chapman, D. "Planning for conjuctive goals". In Artificial Inteligence, 32:333-377. 1987.

[Ka91] Kautz, H.A. “Formal theory of plan recognition”. In Reasoning about Plans - Allen, J.F.; Kautz, H.A.; Telavin, R.N.; Tenemberg, J.D. (eds.), Morgan Kaufmann, Los Altos, CA, EUA, Cap. 2. 1991.

[Sa74] Sacerdoti, E. "Planning in a hierarchy of abstraction spaces". In Artificial Intelligence, 5:115-135. 1974.

[YTW92] Yang, Q; Tenenberg, J.; Woods, S. "Abstraction in nonlinear planning". Technical Report n° CS-92-32, University of Waterloo, Canadá.1992.

[Wo91] Woods, S. G. "An Implementation and evaluation of a hierarchical non-linear planner”. Master's thesis, Computer Science Department, University of Waterloo, Canadá. 1991.