PLANEJAMENTO HIERÁRQUICO NO CÁLCULO DE SITUAÇÕES COM GOLOG

Edward M. I. Iamamoto

Resumo

Grande parte do trabalho prático em sistemas de planejamento em Inteligência Artificial nos últimos quinze anos têm sido baseado em planejamento hierárquico (Sacerdoti, 1974). O planejamento hierárquico baseia-se no conhecimento específico de problemas para remediar a intratabilidade computacional dos problemas de planejamento clássico. Esse conhecimento está na forma da decomposição de tarefas, isto é, ao planejador é dado um conjunto de métodos que dizem como uma tarefa de alto nível pode ser decomposta em tarefas de baixo nível. O propósito dessa monografia é abordar o planejamento hierárquico e como ele pode ser trabalhado como programação de alto nível no cálculo de situações baseado em GOLOG.

1. Introdução

A maior parte dos trabalhos práticos em sistemas de planejamento de Inteligência Artificial durante os últimos quinze anos têm sido baseados em planejadores hierárquicos. Todos esses trabalhos têm sido implementados em linguagens tradicionais procedurais (C, por exemplo), voltadas mais para a manipulação de estruturas de dados armazenadas na memória dos computadores, jumps (if, while, for) e execução seqüêncial de instruções. Apesar de esforços para trabalhar com liguagens de nível mais alto e orientadas a objetos, essas linguagens estão mais associadas à forma como os computadores executam instruções do que à forma de pensar humana.

O raciocínio lógico-dedutivo é uma poderosa ferramenta para apresentar provas e tirar conclusões baseadas em fatos e regras e está mais próximo da forma de pensar dos humanos. Um formalismo lógico sobre a forma como as ações transformam o mundo é necessário para raciocinar logicamente sobre a tarefa de planejamento; uma das propostas foi o *cálculo de situações*. Teorias lógicas podem ser expressas em programas lógicos escritos em linguagens lógicas como Prolog. De fato, planejadores dedutivos podem ser implementados em linguagens lógicas utilizando-se o cálculo de situações (Reiter, 2000).

O cálculo de situações original não define ações complexas com tomadas de decisão (if, while, etc.) nem procedimentos. Então, como associar um algoritmo de planejamento hierárquico ao rigor lógico do cálculo de situações? Isso pode ser feito através da linguagem GOLOG. GOLOG é uma linguagem que implementa ações complexas e procedimentos sobre o cálculo de situações. Com GOLOG podemos programar um algoritmo de planejamento hierárquico como se estivéssemos trabalhando com uma linguagem procedural; mas, na realidade, esse programa é composto de sentenças do cálculo de situações. Daí, podemos continuar raciocinando logicamente sobre o planejamento hierárquico.

Nos próximos capítulos, iremos apresentar brevemente o cálculo de situações, como GOLOG implementa ações complexas e as características do planejamento hierárquico que o tornam um candidato para o uso do GOLOG.

2. Cálculo de Situações

Neste capítulo apresentamos o cálculo de situações (McCarthy, 1963; McCarthy e Hayes, 1969). Na seção 2.1 descrevemos os conceitos de ação, fluente e situação. Na seção 2.2, mostramos como as situações estão associadas aos estados do mundo. Na seção 2.3 mostramos como as ações são axiomatizadas para ter efeitos e pré-condições. Por fim, ilustramos o problema da persistência (frame problem) no cálculo de situações, que surge quando formalizamos mundos dinâmicos; e. mostraremos como a *circunscrição* pode resolver esse problema. O entendimento do cálculo de situações será importante para conhecer a linguagem GOLOG que estudaremos mais a frente.

2.1 - Ações, situações e fluentes.

O cálculo de situações é uma linguagem de lógica de primeira ordem criada para representação e modelagem de mundos dinâmicos. Um mundo que usaremos como referência para nossos exemplos é o *mundo dos blocos*, pois este é familiar à comunidade de Inteligência Artificial, sendo encontrado em muitos outros textos. Basicamente, o mundo dos blocos trata-se de blocos sobre uma mesa, onde tais blocos podem ser empilhados uns sobre os outros (e também podem ser desempilhados).

O cálculo de situações estabelece que as mudanças em um mundo dinâmico ocorrem através da execução de *ações*. A tabela 1 descreve as ações do mundo dos blocos de acordo com o cálculo de situações.

Ação	Descrição
stack(X,Y)	Empilha um bloco X sobre um bloco Y
unstack(X,Y)	Desempilha um bloco X de sobre um bloco Y, colocando-o sobre a mesa
move(X,Y,Z)	Move o bloco X, sobre o bloco Y, para cima do bloco Z

Tabela 1. Descrição de ações para o mundo dos blocos.

O cálculo de situações usa uma constante s_0 para representar uma *situação inicial*, aquela em que nenhuma ação foi executada. Também definimos uma função binária *do*(α, *s*) que denota a situação sucessora de *s* resultante da realização da ação α (alguns autores utilizam a notação result(α, *s*) no lugar de *do*(α, *s*)). Um exemplo de situação resultante da execução de uma ação é *do*(*stack*(*a*, *b*), s_0) que representa a situação após a execução da ação *stack*(a, b) a partir da situação s_0. Outro exemplo de situação resultante, *do*(*move*(*a*, *b*, *c*), *do*(*stack*(*a*, *b*), s_0) é a situação após a execução das ações *stack*(*a*, *b*) e *move*(*a*, *b*, *c*) uma após a outra, notem o uso recursivo da função *do*. Uma característica do uso de *do* é que uma situação carrega consigo o histórico de ações que levaram até ela. No exemplo de

do(*move*(*a*, *b*, *c*), *do*(*stack*(*a*, *b*), s_0), o histórico é composto pelas ações ações *stack*(*a*, *b*) e *move*(*a*, *b*, *c*).

Os valores das relações e funções podem variar de uma situação para outra em mundos dinâmicos. Relações que variam de acordo com a situação são chamados de *fluentes relacionais*. A tabela 2 descreve os fluentes relacionais para o mundo dos blocos.

Fluente	Descrição
clear(X)	O bloco X está livre, sem outro bloco sobre si
ontable(X)	O bloco X está sobre a mesa
on(X,Y)	O bloco X está sobre o bloco Y

Tabela 2. Fluentes relacionais para o mundo do blocos.

Similarmente, funções cujo valor varia de acordo com a situação são chamadas de *fluentes funcionais*. As funções podem assumir valores discretos ou contínuos. Podemos extender o mundo dos blocos para ter um fluente funcional discreto *num_livres*(*X*), onde *X* indicaria o número de blocos livres.

2.2 - Situações e estados do mundo

Como dito anteriormente, uma situação representa um histórico de ações executadas desde a situação inicial. Também podemos entender cada situação como uma descrição do estado do mundo feita através de um conjunto de fluentes. Assim, cada situação está associada a um estado do mundo. Para realizar a associação entre um estado e uma situação *s*, temos que associar *s* a cada um dos fluentes que caracterizam o estado. Fazemos a associação por meio de um predicado *holds*(*f*, *s*) que significa que o fluente *f* é verdadeiro na situação *s*. Na tabela 3 temos *axiomas de observação* que descrevem o estado do mundo na situação inicial por meio do predicado *holds*.

holds(clear(b), s_0)
holds(clear(c), s_0)
holds(ontable(a), s_0)
holds(ontable(b), s_0)
holds(on(c,a, s_0), s_0)

Tabela 3. Axiomas de observação para um estado inicial no mundo dos blocos.

Veja que o predicado *holds* toma como um dos seus termos um fluente que por sua vez é um predicado. Tornar um predicado um termo de outro predicado é uma técnica conhecida como reificação. Com a reificação podemos obter sobre os fluentes um grau de flexibilidade maior do que se tivéssemos apenas os fluentes, constantes e valores devolvidos por funções. Com a reificação, os próprios predicados tornam-se elementos a serem manipulados.

Uma descrição de situação deve ser completa, deve descrever todos os fatos, tudo o que é verdadeiro e o que não é verdadedeiro na situação. Veja que na descrição da tabela 3 não citamos o fato de que o bloco *c* não está sobre a mesa na situação s_0, o que seria

expresso através do axioma $\neg ontable(c, s_0)$; também não representamos o fato de que o bloco *a* não está livre, axioma $\neg clear(a, s_0)$. Descrever todos os fatos do mundo dos blocos não chega a ser impossível, pelo menos nesse exemplo com apenas três blocos; afinal o númeto de objetos é finito (blocos *a*, *b* e *c*) e o número de fluentes é pequeno, só há três fluentes *clear*, *ontable* e *on*. Entretanto, se aumentássemos apenas um pouco o número de blocos seria extremamente trabalhoso descrever toda uma situação. Também poderíamos complicar a tarefa de descrição do mundo aumentando o número de fluentes. De fato, a descrição completa do mundo é quase impossível em termos práticos. Resolvemos esse problema adotando a *hipótese de mundo fechado* segundo a qual o que não estiver descrito deve ser considerado falso. Assim, se não dissermos $ontable(c, s_0)$ ou $clear(a, s_0)$ então é por que esses axiomas são falsos. A hipótese de mundo fechado é importante pois é adotada implicitamente em vários modelos lógicos e algoritmos de planejamento.

2.3 - Axiomatização de ações

A primeira observação que podemos fazer a respeito de ações é que elas têm pré-condições: requerimentos que precisam ser satisfeitos para que uma ação possa ser executada. Para expressar quando uma ação pode ser executada fazemos uso de um novo fluente *poss*, onde $poss(a, s)$ indica que a ação *a* pode ser executada na situação *s*. De posse desse novo fluente, podemos elaborar axiomas que expressem as pré-condições para execução de uma ação. No nosso exemplo do mundo dos blocos, cujo domínio contêm três ações temos os seguintes três *axiomas de pré-condições*:

$$poss(stack(X,Y), S) \leftarrow$$
$$holds(clear(X), S) \wedge holds(clear(Y), S) \wedge holds(ontable(X), S) \wedge X \neq Y \qquad \text{[PRE1]}$$

$$poss(unstack(X,Y), S) \leftarrow$$
$$holds(clear(X), S) \wedge holds(on(X, Y), S) \qquad \text{[PRE2]}$$

$$poss(move(X, Y, Z), S) \leftarrow$$
$$holds(clear(X), S) \wedge holds(clear(Z), S) \wedge holds(on(X, Y), S) \wedge X \neq Z \qquad \text{[PRE3]}$$

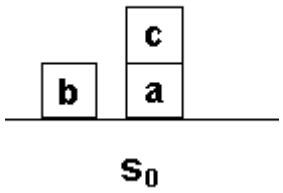
A interpretação desses axiomas é trivial. Por exemplo, em [PRE1] temos que a ação de empilhar um bloco *X* sobre um bloco *Y* só pode ser executada se o bloco *X* e o bloco *Y* estiverem livres e o bloco *X* estiver sobre a mesa.

Outro aspecto das ações a notar é que elas alteram o estado do mundo, ou seja o valor dos fluentes, produzem *efeitos*. Para cada fluente que uma ação afeta, é necessário um *axioma de efeito* que indique como a ação o afeta. Abaixo, temos os *axiomas de efeito* para nossas ações do mundo dos blocos.

$$holds(on(X,Y), do(stack(X,Y), S)) \qquad \text{[E1]}$$
$$holds(clear(Y), do(unstack(X,Y), S)) \qquad \text{[E2]}$$
$$holds(ontable(X), do(unstack(X,Y), S)) \qquad \text{[E3]}$$
$$holds(clear(Y), do(move(X,Y,Z), S)) \qquad \text{[E4]}$$
$$holds(on(X,Z), do(move(X,Y,Z), S)) \qquad \text{[E5]}$$

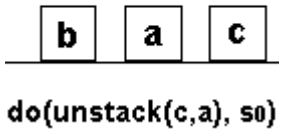
A interpretação desses axiomas também é trivial. Como exemplo, o axioma [E2], significa que ao desempilhar um bloco *X* de cima de um bloco *Y*, a partir de uma situação *S*, temos que a situação resultante terá o fluente *clear*(*Y*) verdadeiro.

Entretanto, representar como o mundo muda através de axiomas de efeito ainda não é o suficiente para representar um mundo dinâmico; ainda é necessário representar como o mundo **não** muda com a execução de ações. Tomemos o exemplo das figuras 1 e 2. A figura 1 mostra uma situação inicial `s0` onde um bloco `b` está livre e sobre a mesa; abaixo da figura temos os fluentes que são verdadeiros em `s0`. Na figura 2, temos a situação resultante após a execução da ação de desempilhar o bloco `c`, resultando em s_1 = *do(unstack(c,a), s0)*. Na figura 2, também temos os fluentes que forma deduzidos a partir de [E1]-[E5].



holds(clear(b), s0)
holds(clear(c), s0)
holds(ontable(a), s0)
holds(ontable(b), s0)
holds(on(c,a, s0), s0)

Figura 1 – Situação inicial



holds(*clear*(*a*), *do*(*unstack*(*c*,*a*), *s0*))
holds(*ontable*(*c*), *do*(*unstack*(*c*,*a*), *s0*))

Figura 2 – Situação após a ação `unstack(c,a)`

Observando a figura 2, temos que ainda é preciso descrever os fluentes que eram verdadeiros em s_0 e que continuam verdadeiros em s_1 para que tenhamos uma descrição comleta do estados s_1. Deveríamos ser capazes de deduzir que também valem os seguintes fatos.

holds(clear(b), do(*unstack*(*c*,*a*), *s0*))
holds(clear(c), do(*unstack*(*c*,*a*), *s0*))
holds(ontable(a), do(*unstack*(*c*,*a*), *s0*))
holds(ontable(b), do(*unstack*(*c*,*a*), *s0*))

Como conseqüência temos, por exemplo, que sem dois fatos, *holds(clear(b), do*(*unstack*(*c*,*a*), *s0*)) e *holds(ontable(b), do*(*unstack*(*c*,*a*), *s0*)), não poderemos deduzir que é possível empilhar o bloco `b` sobre o bloco `a`, pois esses dois fatos são pré-condições para execução dessa ação. Sendo assim, apenas com axiomas de observação do estado inicial, axiomas de efeito e axiomas de pré-condições não conseguimos deduzir todos os fatos e nem todas as ações que podem ser executadas nas situações seguintes à situação inicial. Portanto, precisamos de mais um tipo de axioma, axiomas que digam quais fluentes

permanecem verdadeiros após a execução de uma ação. Esses axiomas são chamados de axiomas de persistência, uma vez que eles resolvem o problema de descrever o que não muda após a execução de uma ação. Na próxima seção descrevemos os axiomas de persistência para o mundo dos blocos e um problema que surge na descrição de mundos dinâmicos.

2.4 - O problema da persistência

Como dito anteriormente, no cálculo de situações é necessário descrever o que não muda ao executarmos uma ação. Tal descrição pode ser feita através de *axiomas de persistência*. Um exemplo de axioma de persistência seria:

$$holds(ontable(V), do(move(X,Y,Z), S)) \leftarrow holds(ontable(V), S) \qquad [P1]$$

A interpretação desse axioma é trivial: se um bloco *V* está sobre a mesa em uma situação *S*, este bloco *V* continuará sobre a mesa em uma situação resultante da ação *move*(*X*,*Y*,*Z*), o que parece bem razoável pois a ação não interfere com o bloco *V*.

O *problema da persistência*, mais conhecido na literatura internacional como o problema do frame (*frame problem)* (McCarthy e Hayes, 1969), surge quando tentamos representar, em lógica formal, o que não muda no mundo ao executarmos uma ação.

Uma característica intuitiva do mundo real é, às vezes, chamada de "lei da inércia", a qual uma das suas variantes enunciamos abaixo.

"*Em geral, um fluente não é afetado por uma ação.*"

Uma das conseqüências dessa lei é que de todos os fluentes de um mundo representado no cálculo de situações, a maioria não é afetada após a execução de uma ação. Suponha um mundo no cálculo de situações que possua N fluentes e M ações. Suponha ainda que cada ação afeta apenas um fluente, o que é uma aproximação grosseira da realidade. Dessas duas suposições anteriores podemos concluir que cada ação **não afeta** N-1 fluentes. Portanto, para cada ação são necessários N-1 axiomas de persistência. Como temos M ações, o número de axiomas de persistência necessários é $O(NM)$. Disto concluímos que o número de axiomas de persistência é grande e pode ser em quantidade muito maior do que os axiomas de efeito e de pré-condições somados; e essa diferença aumenta com o número de fluentes e ações do mundo.

O problema da persistência trata-se de encontrar uma forma compacta de representar todos esses axiomas de persistência. De fato, parece desnecessário valer-se de tantos axiomas de persistência que no final das contas só dizem o óbvio. A lei da inércia responsável por esse problema é a própria solução. Uma forma de resolver o problema é embutir a lei da inércia como um axioma geral do cálculo de situações, um *axioma de persistência universal*.

A lei da inércia pode ser implementada através da *circunscrição* (McCarthy, 1977; McCarthy, 1986; Lifschitz, 1994). Adaptamos a circunscrição para o nosso cálculo de situações como uma regra em lógica de segunda ordem que expressa o seguinte:

> "S*e nada em nossos axiomas indicam que um fluente F seja afetado por uma ação, então esse fluente permanece inalterado após a execução dessa ação*."

Podemos indicar que um fluente é afetado por uma ação através de um predicado de segunda ordem *affects*; onde *affects*(*F*, *A*) indica que um fluente *F* é afetado pela ação *A*. Com axiomas do tipo *affects*(*F*, *A*) podemos dizer quais são todos os fluentes que são afetados por uma ação *A*. Se uma ação A é executada e não tivermos um axioma affects(g, A), então, pela circunscrição, o fluente g não mudou seu valor. A regra que implementa a circunscrição, conhecida como *axioma de persistência universal* é a seguinte:

$$holds(F, do(A, S)) \leftarrow poss(A, S) \wedge holds(F, S) \wedge \neg affects(A, F)$$

A circunscrição, expressa por essa regra é interpretada da seguinte forma: Se um fluente *F* não é afetado por *A* e *F* é verdadeiro na situação *S* e a ação *A* pode ser executada em *S* então o fluente será verdadeiro na situação resultante da execução da ação *A*.

Dada essa regra e o fluente de segunda ordem *affects*, basta enunciar como axiomas quais fluentes são afetados por quais ações; o que elimina a necessidade de enunciar quais fluentes não são afetados por uma ação. Abaixo temos tais axiomas, que chamaremos de *axiomas de afetamento*.

$$affects(stack(X,Y), clear(Y))$$
$$affects(stack(X,Y), ontable(X))$$
$$affects(unstack(X,Y), on(X,Y))$$
$$affects(move(X,Y,Z), on(X,Y))$$
$$affects(move(X,Y,Z), clear(Z))$$

Com o axioma de persistência universal aplicando a circunscrição conseguimos representar o que não muda no mundo com axiomas que estão na mesma ordem do número de ações, coseguindo uma forma compacta de representar a persistência de fluentes.

3. Algoritmos de Planejamento

Formalmente, um problema de planejamento clássico é caracterizado por:

- um espaço de estados S, finito e não-vazio;
- um estado inicial $s_0 \in S$;
- um conjunto de estados meta (ou objetivo) $S_G \subseteq S$;
- um conjunto finito de ações aplicáveis A(s), para cada estado $s \in S$;
- uma função de transição s'= f(a, s), para s, s' $\in$ S e a $\in$ A(s), que mapeia um estado s em outro estado distinto s'.

A tarefa de planejamento consiste em encontrar uma seqüência de ações aplicáveis $\langle a_1, ..., a_n \rangle$ que leve do estado inicial a um estado objetivo $s \in S_G$

Essa caracterização de planejamento clássico (Weld, 1994) é baseada nas seguintes suposições:

- *tempo atômico* – cada ação é executada instantaneamente, de forma ininterrupta, e não é possível a execução simultânea de duas ou mais ações;
- *determinismo* – o efeito de uma ação é uma função somente do estado corrente do mundo no momento em que ela é executada
- *onisciência* – o agente planejador tem conhecimento completo a respeito do estado inicial do mundo e dos efeitos de suas próprias ações;
- *causa de mudança única* – o mundo muda paenas quando o agente age e existe apenas um único agente no mundo.

Sob essa caracterização, um problema de planejamento pode ser representado no cálculo de situações como o problema de deduzir uma situação meta a partir dos axiomas de observação do estado inicial e os axiomas de efeito, pré-condição e persistência universal. A solução do problema estará representada pelo histórico que a situação meta carrega consigo.

Neste capítulo apresentamos alguns conceitos básicos de planejamento. Na parte 3.1 mostramos a representação STRIPS, muito utilizada por vários agentes planejadores. Na parte 3.2 apresentamos o conceito de plano totalmente ordenado e plano parcialmente ordenado. Na parte 3.3 veremos as diferenças entre planejamento como busca em um espaço de estados do mundo e planejamento como busca no espaço de planos. Por último, na parte 3.4, apresentamos dois algoritmos POCL e TOPI que apresentam os aspectos do planejamento, utilizando STRIPS e fazendo buscas pelo espaço de planos, pelo espaço de estados; gerando planos totalmente ordenados e parcialmente ordenados.

3.1 – A representação STRIPS

STRIPS (Stanford Research Institute Problem Solver) (Fikes e Nilsson, 1971) é uma representação alternativa ao cálculo de situações que têm sido usada amplamente em vários planejadores. Existem várias versões dessa linguagem. A que apresentamos é uma versão

proposicional onde não são permitidas variáveis. A razão de apresentarmos essa versão tem a ver com os algoritmos que apresentaremos na seção 3.4, pois todos eles utilizam-se dessa versão STRIPS. Entretanto, faremos algums observações sobre a representação STRIPS com variáveis. Uma outra restrição a STRIPS é que ela só aceita fórmulas atômicas (nada de coisas como `on(a,b) ∧ clear(c)`. Novamente, como no capítulo 1, usaremos o mundo dos blocos em muitos dos nossos exemplos; pois facilitará perceber as semelhanças e, principalmente, as diferenças com o cálculo de situações.

Em STRIPS representamos o estado do mundo por meio de um conjunto de átomos que denotam as propriedades do mundo que são verdadeiras. Essas propriedades são os fluentes que vimos no cálculo de situações. Um exemplo de estado seria Σ = {`clear(b), clear(c), ontable(a), ontable(b), on(c,a)`}. Note que este é o mesmo estado do mundo na situação inicial descrita em 1.2. A representação STRIPS pressupõe a *hipótese de mundo fechado*, isto é a propriedade que não estiver na lista de átomos do estado do mundo deve ser considerada falsa. Sendo assim, `clear(a)` e `ontable(c)` são falsos.

As ações são representadas por operadores que manipulam o conjunto de átomos que representa o estado do mundo. Tais operadores podem adicionar átomos ao estado do mundo ou remover átomos do estado do mundo. Abaixo, apresentamos um exemplo de operador.

```
Op( Ação : move(c, a, b),
    Pre : {clear(c), clear(b), on(c, a)},
    Add : {clear(a), on(b, c)},
    Del : {clear(b), on(c, a})
```

Esse operador representa a ação de mover o bloco `c` de cima do bloco `a` para cima do bloco `b`. `Ação` apenas descreve o operador, serve como um identificador do operador. A lista de átomos designada `Pre` representa todos os átomos que devem ser verdadeiros para que o operador seja aplicado ao estado do mundo. A lista `Add` contêm todos os átomos que devem ser adicionados ao estado do mundo. A lista `Del` contêm todos os átomos que devem ser retirados do estado do mundo.

No nosso estado do mundo Σ = {`clear(b), clear(c), ontable(a), ontable(b), on(c,a)`}, a aplicação do operador `move(c, a, b)` descrito, resultará em uma situação T = {`clear(a), on(b, c), clear(c), ontable(a), ontable(b)`}. Sempre que um operador α fôr aplicado a uma situação Σ, teremos uma situação resultante T = Σ + add(α) – del(α), onde add(α) é a lista `add` do operados α e del(α) é a lista `del` do operador α.

Como a versão STRIPS com que estamos trabalhando é a verão proposicional, temos que ter operadores para todas as ações que representem ações do tipo `move(X, Y, Z)`. Se quisermos operadores para todas as possibilidades de mover os blocos `a`, `b` e `c`, teríamos que descrever os operadores para `move(a, b, c)`, `move(a, c, b)`, `move(b, a, c)`, `move(b, c, a)`, `move(c, a, b)` e `move(c, b, a)`. As versões STRIPS mais complexas, que permitem o uso de variáveis na descrição dos seus operadores não precisam

de todos esses operadores `move`, basta apenas uma descrição de um operador geral de todas as ações `move`. Veja essa descrição abaixo.

```
Op( Ação : move(X, Y, Z),
    Pre : {clear(X), clear(Z), on(X, Y)},
    Add : {clear(Y), on(Z, X)},
    Del : {clear(Z), on(X, Y})
```

Nessa descrição, `X`, `Y` e `Z` são variáveis. Essa versão STRIPS torna a descrição de domínios mais fácil, mas adiciona um custo extra na aplicação de um operador pois as variáveis precisam ser instanciadas todas as vezes que aplicamos um operador.

Voltando à nossa versão proposicional de STRIPS e aos estados T e Σ, vemos que todos os átomos no estado Σ que não são afetados pela ação `move(c, a, b)` , mais precisamente {`clear(c), ontable(a), ontable(b)`} continuam verdadeiros no estado T, pois não são removidos. Essa característica de operadores STRIPS elimina a necessidade de dizer explcitamente quais propriedades permanecem inalteradas após a execução de uma ação, não existe o problema da persistência (frame problem). Esse é um dos motivos, não o único, da aceitação, por muitos planejadores, da representação STRIPS em detrimento do cálculo de situações.

Depois de mostrar como representamos estados do mundo e ações em STRIPS, definiremos um problema de planejamento e uma solução em STRIPS.

Definição. Um problema de planejamento em STRIPS é descrito pela tupla $\langle A, I, G\rangle$ onde I é um estado inicial, G um estado objetivo e A é um conjunto de operadores descrevendo as ações desse domínio.

Definição. A solução para um problema $\langle A, I, G\rangle$ é uma seqüência π de operadores de A tal que $G \subseteq \text{res}(I, \pi)$, ou seja, tal que a execução dos operadores em π, a partir do estado inicial leva a um estado que contenha os átomos de G.

É importante reparar que essa definição de plano implica que, enquanto I é a descrição completa do estado inicial, G é uma descrição parcial do estado objetivo. Muitos estados do mundo podem conter os átomos em G; portanto G não representa um único estado do mundo e sim um conjunto de estados que contêm em comum os átomos em G.

3.2 – Planos totalmente ordenados e planos parcialmente ordenados.

Até agora falamos de planos como uma seqüência de ações a serem executadas. No caso da representação STRIPS, um plano é uma seqüência de de operadores. Quando falamos em *seqüência* induzimos o leitor a pensar que em um plano todos os operadores estão ordenados de acordo com a ordem em que devem ser executados. Se temos um plano com as seqüência de ações S = $\{a_1, a_2, ..., a_n\}$, somos levados a crer que a ação a_1 deve ser executada primeiro, seguida da ação a_2, e assim por diante, até terminarmos com a execução da ação a_n. Um plano em que todos os passos estão ordenados é chamado de

plano totalmente ordenado. Quando deixarmos claro que um plano é totalmente ordenado, a ordem em que as ações aparecem em S, indicará a ordem em que as ações devem ser executadas.

Muitas vezes, entretanto, a ordem em que executamos algumas ações não é relevante para se atingir um objetivo. Por exemplo, ao se fazer um compra no supermercado, não importa se, no carrinho de compras, colocamos as bolachas antes das maçãs ou depois da caixa de leite e dos cereais matinais; mas é importante colocar os sacos de cinco quilos de arroz antes das bolachas (a não ser que ninguém se importe em comer bolachas quebradas). Esse exemplo ilustra que existem algumas ações cuja ordem de execução é importante e outras cuja ordem de execução não é relevante. Podemos representar um plano apenas indicando as ordens relevantes de execução das ações. Chamamos de *plano parcialmente ordenado* àquele em que apenas algumas ações têm sua ordem de execução indicada. Representamos um plano parcialmente ordenado por meio de uma tupla $\langle S, O, L\rangle$ onde S é o conjunto de ações do plano, O é o conjunto de restrições de ordem das ações e L é um conjunto de links causais das ações. Um link causal é uma estrutura da forma $a_p \xrightarrow{c} a_c$, onde a_p é uma ação que satisfaz uma pré-condição c da ação a_c. Os links causais servem para indicar o motivo pelo qual uma ação é colocada antes de outra.

A existência de um link causal $a_p \xrightarrow{c} a_c$ implica necessariamente em uma restrição de ordem $a_p \prec a_c$, mas nem toda restrição de ordem implica em um link causal. Algumas restrições de ordem $a_p \prec a_c$ podem ser devidas ao fato de que se a ação a_c fôr executada antes da ação a_p algum pré-requisito de a_p será destruído por a_c ou algum efeito de de a_c será destruído por a_p. O tratamento e políticas para essas *ameaças* de destruição de efeitos e pré-requisitos será discutido na seção 2.5.

Seja $\langle S, O, L\rangle$ um plano parcialmente ordenado onde $S = \{a_1, a_2, a_3, a_4, a_5\}$, O = { $a_3 \prec a_2$, $a_1 \prec a_5$, $a_2 \prec a_4$, $a_5 \prec a_4$} e L = { }. A representação gráfica desse plano é a seguinte.

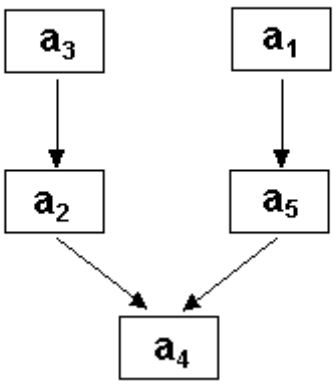


Figura 3 – Representação gráfica de um plano parcialmente ordenado.

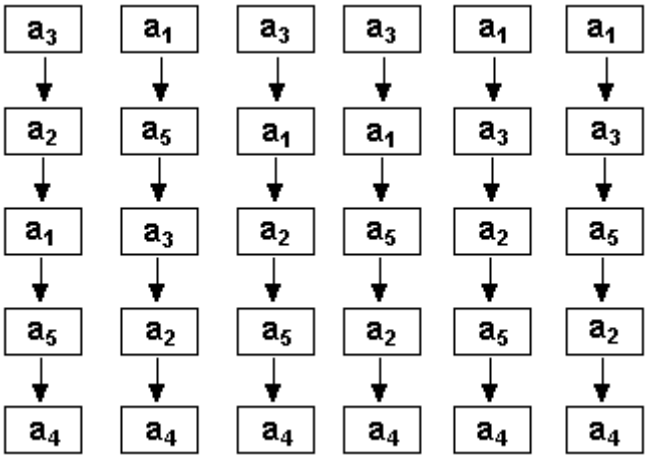


Figura 4 – conjunto de planos representado pelo plano parcial.

A representação de um plano parcialmente ordenado é bem mais complexa que a de um plano totalmente ordenado. Essa complexidade, entretanto, permite que um plano parcialmente ordenado represente todo um conjunto de planos. Um plano totalmente ordenado, por sua vez, pode representar apenas um plano. No plano da figura 3 temos representados 6 planos totalmente ordenados que apresentamos na figura 4. Essa capacidade de representar conjuntos de planos apresentará grande vantagem quando discutirmos o planejamento como busca em um espaço de planos, na próxima seção.

3.3 – Planejamento como busca em um espaço de estados e planejamento como busca em um espaço de planos.

Como vimos anteriormente na seção 1.2, o estado do mundo é determinado pelos valores assumidos pelos fluentes desse mundo. Cada combinação dos valores dos fluentes está associado a um estado do mundo, logo há diferentes estados do mundo. Ao conjunto de todos os estados do mundo possíveis damos o nome de *espaço de estados*.

Quando uma ação α modifica o mundo, ou seja, transforma um estado A em um estado B, podemos dizer que um estado A está associado a um estado B por meio dessa ação α. Assim, todos os estados do espaço de estados podem estar ligados a outros estados por meio de ações. O espaço de estados e as ações estão ligados de maneira a formar um um grafo direcionado onde os estados fazem o papel de nós e as ações fazem o papel de arcos ligando estados.

Seja dois estados *I* e *G* desse grafo e π um caminho no grafo que sai de *I* e chega a G. O caminho π é uma seqüência de arcos que por sua vez representa uma seqüência de ações, ou seja, π é um plano; um plano para, a partir do estado do mundo *I*, chegar ao estado do mundo *G*. Visto dessa forma, podemos planejar através da obtenção de caminhos nesse grafo. O planejamento feito dessa forma é ma verdade uma *busca pelo espaço de estados*.

De acordo com a definição de plano dada no final da seção 2.1, o estado objetivo *G* não precisa ser um estado, pode representar um conjunto de estados. Sendo assim, planejar no espaço de estados passa a ser a busca por um caminho no grafo que leve do estado *I* até algum dos estados pertencentes a *G*.

O caminho, o plano, que leva de um estado *I* a um estado *G* pode não existir pois o grafo pode ter componentes não conexos onde *I* está em um componente e *G* está em outro. Outra possibilidade é existir um caminho que leve do estado *G* até o estado *I* , mas não existir caminho de *I* para *G*; isso é possível em domínios com ações cujos efeitos não podem ser desfeitos por outras ações

Planejar como uma busca pelo espaço de estados não é a única abordagem possível, nem a mais eficiente em muitos casos. Uma alternativa é fazer *planejamento como busca pelo espaço de planos*. Tal espaço de planos consiste no conjunto de todos os planos parcialmente ordenados possíveis.

Para entender como é feita uma busca pelo espaço de planos, introduziremos o conceito de *plano vazio*. Um plano vazio é um plano parcialmente ordenado $\langle S, O, L \rangle$ onde $S = \{p_{inicial}, p_{final}\}$, $O = \{p_{inicial} \prec p_{final}\}$ e $L = \{\}$. $p_{inicial}$ e p_{final} são passos virtuais que não representam ações de fato. Suponha um problema de planejamento $\langle A, I, G \rangle$, um plano vazio para esse problema tem $p_{inicial}$ como uma ação sem pré-condições cujo efeito produz os fluentes do estado inicial *I* e p_{final} é como uma ação sem efeitos cujas pré-condições são os fluentes que caracterizam *G*.

O espaço de planos possui todos os planos possíveis, sendo que alguns deles são executáveis e outros não; alguns resolvem um problema de planejamento e outros planos resolvem outros problemas de planejamento. Um plano pode ser transformado em outro plano através de dois tipos de operadores: adição de uma nova ação e adição nova restrição de ordem. Não consideraremos como operador a adição de links causais, pois os links causais são apenas uma informação adicional para explicar a razão de uma ação ter sido adicionada a um plano. Não teremos operadores de remoção de ações e remoção de restrição de ordem, apesar de serem possíveis. Os planos do espaço de planos estão relacionados entre si por meio dos operadores. Essas relações também podem ser representadas por meio de um grafo orientado onde cada plano está representado por um nó e os nós estão ligados entre si por meio de arcos onde cada arco representa um operador.

Dois fatos sobre esse grafo são relevantes:

Fato 1. Como só temos operadores que adicionam ações e restrições de ordem aos planos, qualquer caminho do grafo termina em um plano mais complexo que o plano do começo do caminho.

Fato 2. Diferentemente da busca pelo espaço de estados em que os caminhos podiam começar por qualquer nó/estado, as buscas pelo espaço de planos sempre começam por um nó, o nó do plano vazio.

A partir desses dois fatos percebe-se que, na realidade nosso grafo é uma árvore com a raiz no plano vazio. Os operadores de adição de ações e adição de restrições de ordem agem como refinadores de planos. Ao percorrer essa árvore em direção às suas folhas temos planos cada vez maiores e com maior número de restrições de ordem. As folhas da árvore correspondem a planos totalmente ordenados.

No planejamento como busca pelo espaço de estados estávamos interessados em achar caminhos pelo grafo, pois o plano para resolver um problema de planejamento era um caminho. No planejamento como busca pelo espaço de planos os caminhos não são importantes, os nós é que são importantes pois neles se encontram os planos, e alguns desses nós podem ser planos com uma solução para um problema de planejamento. A busca pelo espaço de planos percorre a árvore de planos para achar o nó que resolve um problema.

A grande vantagem da busca pelo espaço de planos sobre a busca pelo espaço de estados advém dessa estrutura de árvore do espaço de planos. Podemos podar um ramo inteiro de uma árvore ao constatar que ele se origina de um nó cujo plano possui uma inconsistência devido a um conjunto de ações ou restrições de ordem conflitantes. Essas inconsistências podem ser independentes da ordem em que as ações são executadas. Tais inconsistências não podem ser detectadas durante uma busca pelo espaço de estados pois essa busca só enxerga uma ação de cada vez e assim perderá tempo verificando cada possível combinação de ordem entre ações conflitantes.

3.4 – Algoritmos TOPI e POCL

Existem vários algoritmos de planejamento que fazem a busca pelo espaço de planos e pelo espaço de estados. Apresentaremos dois deles: POCL e TOPI. Esses algoritmos exemplificam o planejamento através da busca de planos totalmente ordenados e parcialmente ordenados.

O algoritmo POCL (Partial Order Causal Link) é uma versão do algoritmo de McAllester; uma versão descendente do TWEAK (Chapman, 1987) e NONLIN (Tate, 1990), mas é conceitualmente mais simples. O algoritmo usa links causais para lembrar-se o propósito de introduzir um passo ao plano e para proteger esse propósito. A inovação chave de McAllester é uma inteligente e metódica técnica para criar e proteger links causais. Dizemos que um link causal $s_i \xrightarrow{p} s_j$ é ameaçado se algum passo s_k pode ser ordenado entre s_i e s_j,e s_k adiciona ou remove uma proposição que unifique com p.

Cada pré-condição p de um passo s_j do plano é uma pré-condição aberta se não têm um link causal $s_i \xrightarrow{p} s$ correspondente. O algoritmo procura por uma solução $\{S, O, B\}$ eliminando pré-condições abertas em G enquanto assegura os links causais em L. Na primeira chamada de POCL, $\{S, O, B\}$ é o plano vazio, G é o conjunto de pré-condições da ação p_{final} e L é um conjunto vazio.

Algoritmo: POCL($\langle S, O, B \rangle$, G, L)

1. Término: Se G está vazio, devolve o plano e pára.
2. Seleção de submeta: Seja c uma submeta em G, e seja s_{need} o passo para o qual c é pré-condição.
3. Seleção de operador: Seja S_{add} o passo que adiciona c. Se tal passo não existir então faz backtrack. $L' = L \cup \{s_{add} \xrightarrow{c} s_{need}\}$, $S' = S \cup \{ s_{add} \}$, $O' = O \cup \{ s_{add} \prec s_{need} \}$, e $B' = B \cup \{$ o conjunto de variáveis para fazer s_{add} adicionar c $\}$.
4. Atualização do conjunto meta: Seja $G' = (G - \{c\}) \cup \{$ pré-condições de S_{add}, se novas $\}$.
5. Proteção de links causais : Um passo s_k ameaça um link causal $s_i \xrightarrow{p} s_j$ quando ocorre entre s_i e s_j e adiciona ou remove p. Para cada passo s_k que ameaça um link causal $s_i \xrightarrow{p} s_j \in L'$:
Assegure que s_k não ameace adicionando uma restrição para O' e/ou B'.
6. Chamada recursiva: POCL($\langle S', O', B' \rangle$, G', L').

Figura 5 – Algoritmo POCL.

Os pontos de backtracking do POCL estão na escolha dos passos adicionados ao plano e nas restrições adicionadas a *O'* e *B'* na proteção de links causais.

O algoritmo TOPI (Total Ordem Prior Insertion) funciona definindo as condições meta como planejamento para submetas e construindo um plano regressivo. Ele considera todos os passos que podem possivelmente adicionar uma submeta sem remover outra submeta ainda não resolvida. Quando ele encontra tal passo, cria um novo estado de plano pela adição desse passo entre s_0 e todos os outros passos já existentes no plano, eliminando as submetas já resolvidas e adicionando as pré-condições do novo passo como novas submetas. O planejador termina quando todas as submetas abertas do plano G unificam com o estado inicial *I*. Veja o algoritmo:

Algoritmo TOPI($\langle S, O, G \rangle$, I, G)
1. **Término:** Se $G \subseteq I$, devolve o plano e pára.
2. Seleção de operador: Seja S_{add} um novo passo que adiciona um conjunto de pré-condições A tal que $A \cap G \neq \emptyset$, não remove $g \in G$, e têm um conjunto de pré-condições C. Seja $S' = S \cup \{ S_{add} \}$, $O' = O \cup \{$ restrições de ordem que fazem vir depois de s_0, mas antes de qualquer outro passo em S $\}$, e $B' = B \cup \{$ restrições que fazem adicionar A e não remover elementos de $G\}$.
3 Atualizar o conjunto meta: Seja G' o conjunto $(G - A) \cup C$.
4. Chamada recursiva: TOPI($\langle S', O', B' \rangle$, I, G')

Figura 6 – Algoritmo TOPI.

Os pontos de backtracking são todos os passos possivelmente adicionados e restrições de variáveis. Note como, quando da escolha de um operador, as novas restrições adicionadas em *O'* obrigam o plano gerado a ser totalmente ordenado.

Os algoritmos TOPI e POCL apresentam todos os aspectos do planejamento que definimos em 3.2 e 3.3. Enquanto POCL trabalha fazendo uma busca pelo espaço de planos e criando um plano parcialmente ordenado, TOPI funciona fazendo uma busca pelo espaço de estados e gerando um plano totalmente ordenado.

4. GOLOG : Ações complexas no cálculo de situações

No capítulo 2 mostamos que o cálculo de situações consegue representar e raciocinar sobre ações simples. O mesmo cálculo de situações pode representar ações complexas e procedimentos como por exemplo:

- **If** *CarInDriveway* **then** *drive* **else** *walk* **endIf**
- **While** *(∃ block) ontable(block)* **do** *removeAblock* **endWhile**
- **Proc** *removeAblock (π x) [pickup(x); putaway(x)]* **endProc**

Aqui, introduzimos a declaração de um procedimento (removeAblock) e também um operador não-determinístico π ; *(π x)[δ(x)]* significa pegar, não deterministicamente, um *x* individual e, para esse *x*, realizar *δ(x)*. Esse tipo de não-determinismo é útil para aplicações robóticas e similares.

O objetivo deste capítulo é proporcionar uma visão de ações complexas e procedimentos baseados no cálculo de situações. Isso resultará em uma linguagem de programação lógica para modelar comportamentos complexos em um mundo dinâmico, GOLOG (alGOl. in LOGic).

4.1 – Ações complexas no cálculo de situações

Definiremos expressões de ações complexas utilizando alguns símbolos extralógicos adicionais (e.g., **while**, **if**, etc.) que agem como "abreviações"para expressões lógicas na linguagem do cálculo de situações. Devemos pensar nesses símbolos extralógicos como macros que expandem em genuínas fórmulas do cálculo de situações.

Abaixo definimos a abreviação *Do(δ, s, s')*, onde δ é uma expressão de ação complexa. Intuitivamente, *Do(δ, s, s')* expandirá em uma fórmula do cálculo de situações que diz que é possível atingir a situação s' a partir da situação s através da execução da seqüência de ações especificadas por δ. Note que as ações complexas podem ser não-determinísticas, isto é, pode ter diferentes execuções terminando em diferentes situações.

Do é definido indutivamente na estrutura do seu primeiro argumento:

1. Ações primitivas.

$$Do(a, s, s') \stackrel{\text{def}}{=} Poss(a[s], s) \wedge s' = do(a[s], s)$$

A notação *a[s]* significa o resultado de restaurar o argumento situação *s* para todos os fluentes funcionais mencionados pela ação *a*. Por exemplo, se *a* é *read(favoriteBook(John))*, e *favoriteBook* é um fluente funcional (o que significa que seu valor é dependente da situação) então *a[s]* é lido como *read(favoriteBook(John, s)*.

2. Ações de teste

$$Do(\phi?, s, s') \stackrel{\text{def}}{=} \phi[s] \wedge s = s'.$$

Exemplos:

se ϕ é $(\forall x)$. ontable(x) $\wedge \neg$on(x, A),
Então $\phi[s]$ equivale a
$(\forall x)$. ontable(x, s) $\wedge \neg$on(x, A, s).

se ϕ é $(\exists x)$ on(x, favoriteBlock(Mary)),
então $\phi[s]$ equivale a
$(\exists x)$ on(x, favoriteBlock(Mary, s), s).

3. Seqüência

$$Do([\delta_1,\delta_2], s, s') \stackrel{\text{def}}{=} (\exists s''). Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s').$$

4. Escolha não-determinística de duas ações

$$Do([\delta_1 | \delta_2], s, s') \stackrel{\text{def}}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s').$$

5. escolha não-determinística de argumentos de ações

$$Do((\pi x)\, \delta(x), s, s') \stackrel{\text{def}}{=} (\exists x)\, Do(\delta(x), s, s').$$

6. Interação não-determinística: Executa zero ou mais vezes.

$$Do(\delta^*, s, s') \stackrel{\text{def}}{=} (\forall P).\{(\forall\, s_1)\, P(s_1,s_1) \wedge (\forall s_1, s_2, s_3)[P(s_1, s_2]) \wedge Do(\delta, s_2, s_3) \leftarrow P(s_1, s_3)\} \leftarrow P(s, s').$$

Em outras palavras, fazer a ação δ zero ou mais vezes leva você de s para s'se, e somente se, (s, s') está em todo conjunto tal que:

(a) (s_1, s_1) está no conjunto para todas as situação
(b) Sempre que (s_1, s_2) está no conjunto, e fazer δ na situação s_2 leva para a situação s_3, então (s_1, s_3) está no conjunto.

4.1.1 - Procedimentos

Podemos extender as definições de ações complexas para incluir procedimentos. A dificuldade de dar uma semântica para chamadas de procedimentos recursivos usando expansão de macros é não existir forma direta para uma macro expadir o corpo de um procedimento quando este inclui uma chamada recursiva para si mesmo.

1. Começamos com uma definição de macro auxiliar: Para qualquer varia'vel P de aridade n + 2, tomando um par de argumentos situação:

$$Do(P(t_1, ..., t_n), s, s') \stackrel{\text{def}}{=} P(t_1[s], ..., t_n[s], s, s').$$

Segue que expressões da forma $P(t_1, ..., t_n)$ ocorrendo em programas servirão como chamadas de procedimento, e $Do(P(t_1, ..., t_n), s, s')$ significará que executar o procedimento P nos parâmetros existentes $t_1, ..., t_n$ causa uma transição da situação s para s'. Note que na expansão de macro, os parâmetros reais (t_i) são primeiro avaliados com respeito a situação corrente s ($t_i[s]$) antes de passar para o procedimento P; assim o mecanismo que estamos definindo é uma *chamada por valor*.

2. Agora, definimos a semântica do cálculo de situações para programas envolvendo procedimentos recursivos. Supomos, no estilo de programação estruturada em blocos padrão, que um programa consiste de uma seqüência de declarações de procedimentos $P_1, ..., P_n$, com parâmteros formais $v_1, ..., v_n$, onde v_i é um vetor de parâmetros,e corpos de procedimentos $\delta_1, ..., \delta_n$ respectivamente, seguido pelo corpo de um programa principal δ_0. $\delta_1, ..., \delta_n, \delta_0$ são ações complexas como descritos em 1 acima. Então o programa vai ter a forma:

$$proc\ P_1(v_1)\ \ \delta_1\ endProc;\ \ proc\ P_n(v_n)\ \delta_n\ endProc;\ \delta_0$$

Definimos o resultado de avaliar um programa desta forma como segue:

$$Do(\{proc\ P_1(v_1)\ \ \delta_1\ endProc;\ \ proc\ P_n(v_n)\ \delta_n\ endProc;\ \delta_0\}, s, s') \stackrel{\text{def}}{=}$$
$$(\forall P_1, ..., P_n).[\Lambda_{i=1,n}(\forall s_1, s_2, v_i).\ Do(\delta_i, s_1, s_2) \leftarrow P_i(v_i, s_1, s_2)] \leftarrow Do(\delta_0, s, s').$$

Mostramos, assim, como ações complexas podem ser expressas no cálculo de situações.

4.2 – Um interpretador GOLOG em Prolog

As definições de ações complexas acima podem ser utilizadas para definir um meta-interpretador GOLOG em Prolog. A seguir, temos uma implementação do interpretador

```
% Diretivas de compilação.
% Garanta que este arquivo seja lido primeiro.
:- dynamic(proc/2).
:- set_flag(all_dynamic, on).

:- op(800, xfy, [&]).   /* Conjunção */
:- op(850, xfy, [v]).   /* Disjunção */
:- op(870, xfy, [=>]).  /* Implicação */
:- op(880, xfy, [<=>]). /* Equivalência */
:- op(950, xfy, [:]).   /* Seqüência de ação */
:- op(960, xfy, [#]).   /* Escolha não-determinística de ação */

do(E1 : E2, S, S1) :- do(E1, S, S2), do(E2, S2, S1).
do(?(P), S, S) :- holds(P, S).
do(E1 # E2, S, S1) :- do(E1, S, S1); do(E2, S, S1).
do(if(P, E1, E2), S, S1) :- do(?(P) : E1 # ?(-P) : E2, S, S1).
do(star(E), S, S1) :- S1 = S; do(E : star(E), S, S1).
do(while(P, E), S, S1) :- do(star(?(P) : E) : ?(-P), S, S1).
do(pi(V, E), S, S1) :- sub(V, _, E, E1), do(E1, S, S1).
do(E, S, S1) :- proc(E, E1), do(E1, S, S1).
do(E, S, do(E, S)) :- primitive_action(E), poss(E, S).

% sub(Name, New, Term1, Term2) :
% Term2 é Term1 com Name substituído por New.
sub(X1, X2, T1, T2) :- var(T1), T2 = T1.
sub(X1, X2, T1, T2) :- not var(T1), T1 = X1, T2 = X2.
sub(X1, X2, T1, T2) :- not T1 = X1, T1 = ..[F|L1], sub_list(X1, X2, L1,
L2), T2 = .. [F|L2].
sub_list(X1, X2, [], []).
sub_list(X1, X2, [T1|L1], [T2|L2]) :- sub(X1, X2, T1, T2), sub_list(X1,
X2, L1, L2).

% O predicado holds implementa transformações no teste de condições.
holds(P & Q, S) :- holds(P, S), holds(Q, S).
holds(P v Q, S) :- holds(P, S); holds(Q, S).
holds(P => Q, S) :- holds(-P V Q, S).
holds(P <=> Q, S) :- holds((P => Q) & (Q => P), S).
holds(-(-P), S) :- holds(P, S).
holds(-(P & Q), S) :- holds(-P V -Q, S).
holds(-(P v Q), S) :- holds(-P & -Q, S).
holds(-(P => Q), S) :- holds(-(-P v Q), S).
holds(-(P <=> Q), S) :- holds(-((P => Q) & (Q => P)), S).
holds(-all(V, P), S) :- holds(some(V, -P), S).
holds(-some(V, P), S) :- not holds(some(V, P), S).
holds(-P, S) :- isAtom(P), not holds(P, S).
holds(all(V, P), S) :- holds(-some(V, -P), S).
holds(some(V, P), S) :- sub(V, _, P, P1), holds(P1, S).
holds(A, S) :- restoreSitArg(A, S, F), F; not retoreSitArg(A, S, F),
isAtom(A), A.

isAtom(A) :- not (A = -W; A = (W1 & W2); A = (W1 => W2); A = (W1 <=> W2);
A = (W1 v W2); A = some(X, W); A = all(X, W).

RestoreSitArg(poss(A), S, poss(A, S)).
```

Figura 6 – Um interpretador GOLOG em Prolog.

5. Planejamento Hierárquico.

Uma das motivações para o planejamento hierárquico (HTN planning – Hierarchical Task Network planning) era fechar a brecha entre o as técnicas de planejamento em IA como o planejamento baseado em STRIPS, e técnicas de pesquisa operacional para gerenciamento de projetos e escalonamento (Tate, 1990). Assim, existem algumas similaridades entre o planejamento hierárquico e o planejamento tipo STRIPS, mas também algumas diferenças significativas.

5.1 – Tarefas primitivas, rede de tarefas e tarefas não-primitivas.

O planejamento hierárquico usa ações e estados do mundo que são similares àqueles usados no planejamento STRIPS. Cada estado do mundo é representado por um conjunto de átomos verdadeiros naquele estado. Ações, que no planejamento hierárquico são usualmente chamados de *tarefas primitivas*, correspondem a transições de estado; i. e., cada ação é um mapeamento parcial do um conjunto de estados para outro conjunto de estados.

A diferença primária entre planejadores hierárquicos e planejadores STRIPS está sobre o que eles planejam e como eles planejam. No planejamento STRIPS, o objetivo é encontrar uma seqüência de ações que leve o mundo para um estado que satisfaça certas condições ou estado objetivo . O planejamento continua procurando operadores que tenham os efeitos desejados e tornando as pré-condições desses operadores em novas sub-metas. Em contraste, os planejadores hieráquicos procuram por planos que realizem *redes de tarefas* (task networks), que podem incluir coisas além de estados objetivos; e eles planejam por decomposição de tarefas e resolução de conflitos.

Uma rede de tarefas é uma coleção de tarefas que precisam ser cumpridas, junto com restrições de ordem em que as tarefas devem ser executadas, a forma como as variáveis são instanciadas, e quais literais devem ser verdadeiros antes ou depois de cada tarefa. Deferente do planejamento STRIPS, as restrições podem ou não conter condições que devem ser verdadeiras no estado final.

Formalmente, uma rede de tarefas é uma construção sitática da forma $[(n_1 : \alpha_1) \ldots (n_m : \alpha_m), \phi]$, onde

- Cada α_i é uma tarefa
- n_i, é um rótulo para α_i (para distingui-lo de qualquer outra ocorrência de α_i na rede)
- ϕ é uma fórmula booleana construída de restrições de variáveis de ligação como ($v = v'$) e ($v = c$), restrições de ordenação como ($n \prec n'$), e restrições de estado como (*initially l*), (n, l), (l, n), (n, l, n'), onde v e v' são variáveis, l é um literal, c é uma constante, e n e n' são rótulos de nó. Intuitivamente, ($n \prec n'$) significa que a tarefa rotulada com n deve preceder a tarefa rotulada com n'; (n, l), (l, n) e (n, l, n') significam, respectivamente, que l deve ser verdadeiro no estado imediatamente após n, imediatamente antes de n, e em todos os estados entre n e n'. (*initially l*) significa que l deve ser verdadeiro no estado inicial. Negação e disjunção são permitidas na fórmula de restrição.

Como exemplo, a Figura 1 mostra uma rede de tarefas do mundo dos blocos e sua representação gráfica. Nesta rede de tarefas há três tarefas, liberar v_1, liberar v_2 e mover v_1 para v_2. A rede de tarefas também inclui restrições para que mover v_1 deva ser feito por último, v_1 e v_2 devem permanecer livres até movermos v_1, e que a variável v_3 é limitada pela localização de v_1 antes de v_1 ser movido.

$$[(n_1 : achieve[clear(v_1)]\,)\,(n_1 : achieve[clear(v_1)]\,)$$
$$(n_3 : do[move(v_1, v_2, v_3)]\,)$$
$$(n_1 \prec n_3) \wedge (n_2 \prec n_3) \wedge (n_1, clear(v_1), n_3)$$
$$\wedge\, (n_2, clear(v_2), n_3) \wedge (on(v_1, v_3), n_3) \wedge \neg(v_1 = v_2)$$
$$\wedge\, \neg(v_1 = v_3) \wedge \neg(v_2 = v_3)]$$

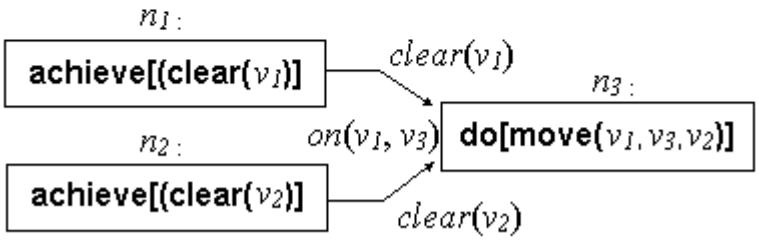


Figura 1 : Uma rede de tarefas e sua representação gráfica.

Uma rede de tarefas que contenha apenas tarefas primitivas é chamada rede de tarefas primitivas. Tal rede pode ocorrer, por exemplo, em problemas de escalonamento. Neste caso, um planejador hierárquico tentaria encontrar um escalonamento que satisfaça todas as restrições.

Em um caso mais geral, uma rede de tarefas pode conter *tarefas não-primitivas*, as quais o planejador precisa compreender para realizá-las. Tarefas não-primitivas não podem ser executadas diretamente porque elas represntam atividades que podem envolver a realização de muitas outras tarefas. Por exemplo, a tarefa de viajar para Nova York pode ser realizada de várias maneiras: indo de avião, dirigindo um carro ou tomando um navio. Ir de avião envolve tarefas como fazer reservas, ir ao aeroporto, comprar uma passagem e embarcar no avião; e voar só funcionará se certas condições forem satisfeitas: disponibilidade de passagens, estar no aeroporto na hora certa, ter dinheiro suficiente para a passagem, etc.

As formas de realizar uma tarefa não-primitiva são representadas usando-se construções chamadas *métodos*. Um método é uma construção sintática da forma (α, d) onde α é uma tarefa não-primitiva, e d é uma rede de tarefas. Ele estabelece que uma forma de realizar a tarefa α é realizando todas as tarefas na rede de tarefas d sem violar as restrições de d.

5.2 – Algoritmo de planejamento hierárquico

Diferentes sistemas que usam que usam algoritmos heurísticos têm sido criados para o planejamento hierárquico (Tate, 1990; Vere, 1983; Wilkins, 1988), e muitos artigos recentes têm tentado proporcionar descrições formais desses algoritmos (Yang, 1990; Kambhampati, *et al*., 1992). A próxima figura apresenta a essência desses algoritmos. Como mostra essa figura, o planejamento hierárquico trabalha expandindo tarefas e resolvendo conflitos interativamente, até um plano livre de conflitos que seja composto apenas de tarefas primitivas ser encontrado.

```
1. Entrar com o problema de planejamento P.
2. Se P contêm apenas tarefas primitivas então
      resolver os conflitos em P e devolver o resultado
      Se os conflitos não puderem ser resolvidos, devolver falha.
3. Escolher uma tarefa não-primitiva t em P.
4. Escolher uma expansão para t.
5. Substituir t pela expansão.
6. Encontrar interações entre as tarefas em P e sugerir formas de lidar
com elas.
7. Aplicar uma das formas sugeridas no passo 6.
8. Ir para o passo 2.
```

Os conceitos de tarefas primitivas e tarefas não-primitivas do planejamento hierárquico têm similaridades com as ações simples e ações complexas do GOLOG. Assim, GOLOG é uma linguagem adequada para trabalhar com o planejamento hierárquico. A vantagem de se utilizar o GOLOG está no fato de que seus mecanismos de controle e procedimentos são decompostos em axiomas do cálculo de situações, e isso permite que se raciocine logicamente sobre o planejamento hierárquico.

6. Conclusão

O planejamento hierárquico é uma técnica de planejamento muito utilizada em sistemas práticos de Inteligência Artificial. Mostramos que o raciocínio lógico sobre o planejamento hierárquico pode ser obtido utilizando-se a linguagem GOLOG para implementar o planejamento hierárquico. Dessa forma, temos o algoritmo de planejamento hierárquico expresso em termos de sentenças do cálculo de situações, que nos fornece as ferramentas para raciocinar logicamente.

7. Referências

CHAPMAN, D. 1987. Planning for conjuntive goals, Artificial Intelligence. 32(3) pages 333-377.

FIKES, R. E., AND NILSSON, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving, Artificial Intelligence, vol. 2 (3/4), pages 189-208.

KAMBHAMPATI, S. AND HENDLER, J. 1992. A Validation Structure Based Theory of Plan Modification and Reuse. Atificial Intelligence, May, 1992.

LIFSCHITZ, V. 1994. Circumscription, In D. M. Gabbay, C. J. Hogger, and J. A Robinson, editors, The Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 3, pages 297-352.

MCCARTHY, J. 1963. Situations, Actions and causal laws, Memo 2, Stanford University Artificial Intelligence Project, Stanford, California.

MCCARTHY, J., AND HAYES, P. J. 1969. Some Philosophical Problems from Standpoint of Artificial Intelligence, In B. Meltzer, D. Michie and M. Swann, ed., Machine Inteligence 4, pages 463-502.

MCCARTHY, J. 1977. epistemological Problems of Artificial Intelligence, In Proceedings of IJCAI-77, pages 1038-1044.

MCCARTHY, J. 1986 Applications of circumscription to Formalizing Common Sense Knowledge, Artificial Intelligence, 26(3):89-116.

Reiter, Raimond. 2000. Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems. 3.3.

SACERDOTI, E. 1974. Planning is a hierarchy of abstraction spaces. Artificial Inteligence 5:115-135.

TATE, A. 1990. Generating Project Networks In Proc. IJCAI-77, pp. 888-893.

WILKINS, D. 1988. Domain-independent Planning: Representation and Plan Generation. In Allen, james; Hendler, James; ant Tate, Austin, editors 1990, Readings in planning. Morgam Kaufmann, CA.

YANG, Q. 1990. Formalizing planning knowledge for hierarchical planning Computational Intelligence Vol.6., 12-24.

WELD, D. S. 1994. An Introduction to Least Commitement Planning, AI Magazine, 15(4), pages 27-61.