Planejamento Abdutivo no Cálculo de Eventos

Silvio do Lago Pereira

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO GRAU DE MESTRE
EM
CIÊNCIA DA COMPUTAÇÃO

Área de concentração: Inteligência Artificial

Orientadora: Prof^{a}. Dr^{a}. Leliane Nunes de Barros

Durante a elaboração deste trabalho o autor recebeu apoio financeiro da CAPES.

— São Paulo, 3 de junho de 2002—

Planejamento Abdutivo no Cálculo de Eventos

Este exemplar corresponde à redação final
da dissertação, devidamente corrigida e
defendida por *Silvio do Lago Pereira*,
aprovada pela comissão julgadora.

São Paulo, 3 de junho de 2002.

Banca examinadora:

- Prof^{a}. Dr^{a}. Cristina Gomes Fernandes – IME-USP
- Prof^{a}. Dr^{a}. Leliane Nunes de Barros – IME-USP
- Prof. Dr. Márcio Rillo – EP-USP

À minha família.

Agradecimentos

À professora Leliane Nunes de Barros,
que me orientou e me incentivou nesse trabalho.
Aos professores do Departamento de Computação,
que contribuíram para minha formação no mestrado.
Aos colegas e amigos do IME, Ariane, Bianka, Cláudia, Daniel,
Emmanuel, Eudênia, Gordana, Leandro, Lorena, Luciano, Marcelo e
Mateus, que compartilharam comigo momentos de estudo e descontração.
Aos meus pais e ao meu grande amigo Itivaldo, pela força e apoio de sempre.
E por último, ao mais importante de todos: a Deus, que permitiu que
todas essas pessoas extraordinárias estivessem no meu caminho.

Resumo

Nesse trabalho, estabelecemos uma correspondência entre raciocínio abdutivo no cálculo de eventos e planejamento de ordem parcial. Para tanto, implementamos três sistemas de planejamento abdutivo baseado em cálculo de eventos (ABP, SABP e RABP) e três sistemas algorítmicos (de ordem parcial) correspondentes (POP, SNLP e TWEAK). Então, através de uma análise comparativa do comportamento desses sistemas, mostramos que planejadores (lógicos e algorítmicos) que implementam estratégias de planejamento correspondentes apresentam comportamentos idênticos (*i.e.* examinam o mesmo espaço de busca e apresentam praticamente a mesma eficiência). Também mostramos que, tanto para sistemas lógicos quanto algorítmicos, a eficiência de planejamento não depende apenas da política de proteção de submetas adotada em cada um deles, mas também das características específicas do domínio de planejamento considerado.

Abstract

In this work, we establish a correspondence between abductive reasoning in the event calculus and partial order planning. Aiming at this end, we implement three abductive planning systems based on event calculus (ABP, SABP and RABP) and three corresponding (partial order) algorithmic systems(POP, SNLP and TWEAK). Then, through a comparative analysis of the behavior of these systems, we show that (logical and algorithmic) planners that implement corresponding strategies of planning present identical behaviors (*i.e.* they examine the same search space and they present practically the same efficiency). Also we show that, in algorithmic and logical systems, the planning efficiency does not depend only on the subgoal protection politics adopted in each one of them, but also on the specific features of the planning domain considered.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O homem que não planeja seus passos
encontra problemas logo na sua porta.

Confucius

Pensador e Filósofo
(551 a.C. – 479 a.C.)

1.1 Motivação

Um dos principais objetivos da área de Inteligência Artificial (IA) é a criação de *agentes*, *i.e.* entidades capazes de demonstrar comportamento inteligente e efetivo na solução de problemas. Ao contrário de um agente puramente *reativo*, que simplesmente reage aos estímulos que recebe de seu ambiente, um agente *racional* deve ser capaz de prever situações futuras e de planejar suas ações, de acordo com os resultados que pretende atingir [56, 53]. De fato, *a habilidade de planejar é essencial ao comportamento inteligente* e sua implementação é extremamente importante em aplicações práticas como, por exemplo, robótica, manufatura, logística, planejamento de grades curriculares, planejamento de missões espaciais, planejamento de provas de teoremas, *etc.*

Nos últimos 30 anos, um grande número de algoritmos de planejamento foram propostos na área de IA. Dentre eles, aqueles provados corretos possuem grandes limitações, em particular, quanto à representação de ações e, conseqüentemente, não podem ser usados para resolver alguns tipos de problemas no mundo real. Por outro lado, os

chamados planejadores práticos [13], capazes de resolver problemas grandes, em geral, foram construídos de maneira *ad hoc*, sendo difícil explicar porquê eles funcionam ou mesmo porquê o seu comportamento pode ser considerado inteligente.

Segundo *Shanahan* [56],

> "A melhor maneira de se explicar um comportamento inteligente (...) é interpretá-lo como produto de um *raciocínio correto* sobre uma *representação correta*. (...) [e] A melhor [ferramenta], na verdade a única candidata real, que temos para explicar os conceitos de representação correta e raciocínio correto, é a lógica formal."

Green [23] foi o primeiro a implementar um sistema de planejamento dentro de uma abordagem lógica. Entretanto, embora seu sistema tenha sido muito admirado do ponto de vista teórico, na prática, ele se mostrou bastante ineficiente. Tal ineficiência é devida, sobretudo, à necessidade de se manter uma enorme quantidade de axiomas para estabelecer que propriedades persistem no mundo, após a execução de uma determinada ação (*problema da persistência*[1]). Conforme *McCarthy* e *Hayes* [43] observam, essa necessidade é inerente a qualquer formalismo para representação de ações e efeitos baseado em lógica monotônica. Em virtude disso, e devido ao fato de diversos algoritmos resolverem satisfatoriamente o problema da persistência temporal, criou-se uma falsa idéia de que a abordagem lógica não poderia ser usada na implementação de sistemas de planejamento realmente eficientes [53]. Este foi um dos motivos pelos quais, desde então, a abordagem lógica foi quase completamente abandonada, enquanto a abordagem algorítmica tornou-se predominante na área de planejamento em IA.

Recentemente, entretanto, *Shanahan* [57] publicou um artigo em que a abordagem lógica de planejamento é resgatada. Nesse artigo, ele mostra que usando *cálculo de eventos circunscritivo*, como formalismo para raciocinar sobre ações e efeitos, e *programação lógica abdutiva*, como técnica de prova automática de teoremas, é possível reproduzir a computação efetuada por um algoritmo clássico de planejamento. Assim, acredita-se que através do uso de lógica formal é possível construir planejadores corretos, fundamentados em princípios bem conhecidos, que possam ser facilmente validados, mantidos ou modificados. Resta investigar se é possível construir planejadores baseados em lógica que possam ser considerados tão eficientes quanto alguns planejadores conhecidos da literatura de planejamento em IA.

[1] Também conhecido como *problema do quadro* (*frame problem*).

1.2 Planejamento clássico

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* $S_G \subseteq S$;
- 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'.

O espaço de estados de um mundo pode ser modelado por um grafo orientado cujos nós representam os estados desse mundo e cujas arestas representam as ações que transformam um estado em outro [31]. Nesse caso, a *tarefa de planejamento* consiste na busca[2] de uma seqüência de ações aplicáveis $\langle a_1, \ldots, a_n \rangle$, *i.e.* um *plano*, que define nesse grafo um caminho que leva do estado inicial s_0 a um estado meta $s \in S_G$.

Essa caracterização de planejamento clássico [61] é baseada nas seguintes suposições:

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

1.3 Abordagem lógica

Na maioria dos sistemas de planejamento descritos na literatura de IA, se não em todos, propriedades do mundo são descritas através de fórmulas da lógica de primeira ordem. Nesse sentido, *latu sensu*, podemos dizer que todo sistema de planejamento é baseado em lógica. Nesse trabalho, porém, a expressão *baseado em lógica* será reservada para designar sistemas onde a lógica é empregada não apenas como uma linguagem para

[2] Tornar essa busca mais eficiente é a principal preocupação da área de planejamento em IA.

descrever ações mas, sobretudo, como um mecanismo de inferência através do qual seja possível "derivar" um plano como conseqüência lógica de um conjunto de axiomas que descrevem as ações do domínio, a situação inicial do mundo e a meta a ser atingida.

Conforme *Bacchus* [2] observa,

> "A Lógica e a abordagem lógica há muito tempo são criticadas como sendo computacionalmente ineficientes e, apesar disso, alguns dos planejadores de melhor desempenho na competição [de planejamento AIPS'2000] foram baseados em lógica. (...) [entretanto] Um exame mais profundo mostra que nesses sistemas o escopo do raciocínio lógico é severamente restrito (...). Em particular, o raciocínio efetuado nesses sistemas tem mais a ver com *verificação de modelos* (*i.e.* verificar se um *único* modelo satisfaz uma fórmula) do que com o que tradicionalmente é visto como sendo *raciocínio lógico*, *i.e.* a aplicação de teorias de prova dedutiva para produzir conclusões verdadeiras para todo um conjunto de modelos."

Assim, um sistema de planejamento desenvolvido segundo a abordagem lógica, *strictu sensu*, deve ser capaz de sintetizar planos através de um raciocínio lógico explícito, como efeito colateral da prova automática de teoremas.

1.4 Objetivos

A proposta desse trabalho consiste em:

- Apresentar, cronologicamente, as principais idéias da área de planejamento.
- Corroborar a conjectura de *Shanahan*, segundo a qual raciocínio abdutivo no cálculo de eventos e planejamento de ordem parcial são isomorfos.
- Mostrar que um planejador abdutivo baseado em cálculo de eventos é capaz de implementar métodos de planejamento sistemático e redundante.
- Estabelecer uma correspondência entre métodos de planejamento sistemático e redundante e planejamento abdutivo baseado em cálculo de eventos.
- Mostrar que, assim como nos sistemas algorítmicos, também nos sistemas lógicos a eficiência depende fortemente das características do domínio considerado.

Com isso, visamos principalmente mostrar que é possível desenvolver planejadores lógicos cuja eficiência seja equiparável àquela observada em alguns planejadores algorítmicos clássicos, selecionados da literatura de planejamento em IA.

1.5 Organização

Essa dissertação está organizada da seguinte maneira:

Capítulo 2. Introduzimos o Cálculo de Situações, um formalismo lógico especialmente criado para modelagem de mundos dinâmicos, e mostramos como podemos utilizá-lo para representar ações e raciocinar sobre seus efeitos. O objetivo desse capítulo é definir planejamento em termos de raciocínio dedutivo e mostrar que, usando prova automática de teoremas, a especificação lógica de um problema de planejamento corresponde, diretamente, à implementação de um planejador que resolve esse problema.

Capítulo 3. Introduzimos a representação Strips, cuja principal vantagem em relação ao cálculo de situações é eliminar a necessidade de axiomas de persistência temporal, permitindo assim um acréscimo de eficiência computacional. O objetivo desse capítulo é apresentar as principais idéias de planejamento algorítmico (sobretudo a idéia de ordem parcial), de modo que, mais tarde, possamos identificá-las e reproduzi-las nos sistemas de planejamento lógico que serão desenvolvidos.

Capítulo 4. Introduzimos o princípio de abdução e mostramos como utilizá-la para estender a capacidade dedutiva do Prolog. Em seguida, introduzimos o Cálculo de Eventos e mostramos que um meta-interpretador abdutivo especializado para esse formalismo implementa, de fato, um sistema de planejamento de ordem parcial. O objetivo desse capítulo é mostrar que, usando abdução e cálculo de eventos, podemos implementar um sistema de planejamento lógico que simula todos os passos de um sistema de planejamento algorítmico clássico.

Capítulo 5. Descrevemos a implementação de três sistemas de planejamento clássico e de suas respectivas versões abdutivas. Em seguida, relatamos dois experimentos que realizamos com esses sistemas: com o primeiro deles, confirmamos que *raciocínio abdutivo e planejamento de ordem parcial são isomorfos*; com o segundo, que *a eficiência de um planejador (lógico ou algorítmico) não depende apenas do método de planejamento que ele implementa mas, sobretudo, das características do domínio considerado.* O objetivo desse capítulo é mostrar que um sistema de planejamento lógico, baseado em abdução no cálculo de eventos, pode ser tão eficiente quanto um sistema de planejamento algorítmico, baseado em Strips.

Capítulo 6. Apresentamos a conclusão final desse trabalho, apontamos suas principais contribuições e indicamos como ele pode ser estendido de modo a tratar alguns aspectos importantes que não foram nele abordados.

Capítulo 2

Planejamento dedutivo

Falhar em planejar é
planejar para falhar.

Effie Jones

Socióloga

2.1 Cálculo de situações

O CÁLCULO DE SITUAÇÕES, introduzido originalmente por *McCarthy & Hayes* [40, 43], foi especialmente criado para descrever mudanças em mundos dinâmicos. Trata-se de formalismo para raciocínio sobre ações e efeitos cuja ontologia inclui *situações*, que são como instantâneos do mundo; *fluentes*, que denotam propriedades do mundo que podem mudar de uma situação para outra; e *ações*, que transformam uma situação em outra. Na linguagem do cálculo de situações[1], que é na verdade um dialeto da lógica de predicados, a constante s_0 denota a *situação inicial*, a função $do(\alpha, \sigma)$ denota a *situação resultante* da execução da ação α numa determinada situação σ, o predicado $poss(\alpha, \sigma)$ estabelece que é possível executar a ação α na situação σ e, finalmente, o predicado $holds(\phi, \sigma)$ estabelece que o fluente ϕ vale na situação σ.

Para exemplificar o uso do cálculo de situações num domínio de aplicação específico, vamos considerar o *mundo dos blocos* [45], ilustrado na figura 2.1. Nesse mundo, há

[1] Por convenção, símbolos com inicial maiúscula denotarão variáveis, enquanto aqueles com inicial minúscula denotarão constantes.

uma série de blocos dispostos sobre uma mesa, cuja superfície é suficientemente ampla para acomodar todos os blocos, e o agente é um robô equipado com um único braço. Os blocos podem ser empilhados uns sobre os outros mas, para que não caiam, somente um bloco pode ser posicionado diretamente em cima de outro. Ademais, como o braço não suporta mais de um bloco por vez, o robô só pode pegar um bloco se esse bloco estiver livre, ou seja, se não houver um outro bloco em cima dele. Nesse domínio, nosso interesse será raciocinar sobre os efeitos de se movimentar blocos de um local para outro. As tabelas 2.1 e 2.2 descrevem, respectivamente, as ações e os fluentes que usaremos na formalização desse domínio[2].

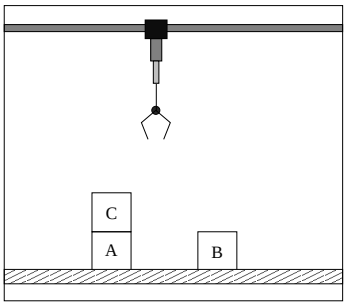


Figura 2.1: *Uma configuração inicial para o mundo dos blocos.*

ação	*descrição*
$stack(X, Y)$	*empilha o bloco X, que está sobre a mesa, em cima do bloco Y*
$unstack(X, Y)$	*desempilha o bloco X de cima de Y, movendo-o para a mesa*
$move(X, Y, Z)$	*move o bloco X, que está sobre o bloco Y, para cima do bloco Z*

Tabela 2.1: *Ações para o mundo dos blocos.*

fluente	*descrição*
$clear(X)$	*o bloco X está livre, i.e. não há outro bloco sobre ele*
$ontable(X)$	*o bloco X está posicionado diretamente sobre a mesa*
$on(X, Y)$	*o bloco X está posicionado diretamente sobre o bloco Y*

Tabela 2.2: *Fluentes para o mundo dos blocos.*

[2]Usaremos símbolos em inglês para manter a compatibilidade com a descrição que é normalmente usada nas competições de planejamento organizadas pelo AIPS - *Artificial Intelligence, Planning and Scheduling Conference.*

2.1.1 Descrição da situação inicial

Uma *situação* é como se fosse um instantâneo do mundo: numa determinada situação, o mundo se mantém estático; porém, em diferentes instantes do tempo, o mundo pode estar em diferentes situações. Essencialmente, o que diferencia uma situação de outra são as propriedades do mundo que podemos observar em cada uma delas e, sendo assim, a maneira mais simples de descrever uma situação é estabelecendo que fluentes valem nessa situação.

No cálculo de situações, a situação inicial de um mundo, designada pelo termo s_0, é descrita através de um conjunto de sentenças da forma $holds(\phi, s_0)$, denominadas *axiomas de observação*. Por exemplo, as sentenças [OB1]-[OB5] a seguir descrevem a situação inicial para o mundo dos blocos, conforme ilustrado na figura 2.1.

$holds(clear(b), s_0)$ [OB1]
$holds(clear(c), s_0)$ [OB2]
$holds(ontable(a), s_0)$ [OB3]
$holds(ontable(b), s_0)$ [OB4]
$holds(on(c, a), s_0)$ [OB5]

Como podemos notar, até mesmo em domínios muito simples, como é o caso do mundo dos blocos, a descrição completa de uma situação real é praticamente impossível. Por exemplo, na descrição acima, não há informação sobre a posição horizontal relativa dos blocos ou sobre a posição da garra do robô; nem sobre fatos negativos, tais como $\neg holds(clear(a), s_0)$ ou $\neg holds(ontable(c), s_0)$. A despeito desse fato, entretanto, dentro de um particular contexto de discurso, é importante que o conjunto de axiomas de observação descreva, completamente, todos os aspectos *relevantes* da situação inicial.

2.1.2 Descrição das ações do domínio

Enquanto a noção de situação é estática, a noção de ação é dinâmica. De fato, a situação de um mundo só se transforma como conseqüência da ocorrência de um evento, sendo a ação um tipo especial de evento [22]. Por exemplo, a queda de uma folha de uma árvore é um *evento*, enquanto a poda de uma árvore é uma *ação*. O que torna a ação

um evento especial é justamente o fato dela ser executada *intencionalmente* por um agente que, nesse caso, pode prever e controlar a sua ocorrência.

Considerando-se que no mundo haja um único agente, que as ações desse agente sejam determinísticas e que não ocorram eventos inesperados, uma ação α pode ser modelada como uma função $\sigma_f = \alpha(\sigma_i)$, onde σ_i denota a situação do mundo *antes* de α ser executada e σ_f denota a situação do mundo *após* a sua execução [61]. Sempre que uma ação é executada, algumas propriedades do mundo que eram falsas tornam-se verdadeiras, enquanto outras que eram verdadeiras tornam-se falsas. Desta forma, o mundo pode persistir numa determinada situação somente até que uma ação seja executada e altere uma de suas propriedades.

Leis causais

Uma componente essencial na modelagem de mundos dinâmicos é a descrição de *leis causais* do tipo *ação* $\Rightarrow$ *efeito*. No cálculo de situações, essas leis são representadas através de sentenças, denominadas *axiomas de efeito*, que descrevem como as ações de um agente afetam os valores dos fluentes no domínio. Um axioma de efeito tem a forma $holds(\phi, do(\alpha, \sigma))$ e estabelece que o fluente ϕ é um efeito da execução da ação α na situação σ. Por exemplo, o conjunto de axiomas[3] a seguir descreve os efeitos das ações do robô no mundo dos blocos.

$holds(on(X,Y), do(stack(X,Y),S))$ [BW1]
$holds(clear(Y), do(unstack(X,Y),S))$ [BW2]
$holds(ontable(X), do(unstack(X,Y),S))$ [BW3]
$holds(clear(Y), do(move(X,Y,Z),S))$ [BW4]
$holds(on(X,Z), do(move(X,Y,Z),S))$ [BW5]

Precondições

Tão importante quanto descrever os efeitos causados pela ocorrência de uma ação, é estabelecer em que circunstâncias ela pode ocorrer, ou seja, que precondições devem estar satisfeitas para que a ação possa ser executada numa determinada situação.

[3] Por convenção, toda variável sem quantificação explícita será considerada universal.

No cálculo de situações, as precondições de uma ação são estabelecidas através de sentenças da forma $poss(\alpha, \sigma) \leftarrow holds(\pi_1, \sigma) \wedge \cdots \wedge holds(\pi_n, \sigma)$, denominadas *axiomas de precondições*, que estabelecem que é *possível* executar a ação α na situação σ se suas precondições $\pi_1, \ldots, \pi_n$ estão satisfeitas nessa situação. Por exemplo, os axiomas a seguir descrevem as precondições para as ações *stack*, *unstack* e *move*, respectivamente.

$$poss(stack(X,Y),S) \leftarrow \quad \text{[BW6]}$$
$$holds(clear(X),S) \wedge holds(clear(Y),S) \wedge holds(ontable(X),S) \wedge X \neq Y$$
$$poss(unstack(X,Y),S) \leftarrow \quad \text{[BW7]}$$
$$holds(clear(X),S) \wedge holds(on(X,Y),S)$$
$$poss(move(X,Y,Z),S) \leftarrow \quad \text{[BW8]}$$
$$holds(clear(X),S) \wedge holds(clear(Z),S) \wedge holds(on(X,Y),S) \wedge X \neq Z$$

Em [BW6], estabelecemos que um bloco X pode ser empilhado sobre outro Y se ambos estão livres e o primeiro deles está sobre a mesa. Em [BW7], estabelecemos que um bloco X pode ser desempilhado de cima de outro bloco Y se X está livre e está posicionado em cima de Y. Finalmente, em [BW8], estabelecemos que um bloco X pode ser movido de cima de Y para cima de Z se tanto X quanto Z estão livres e X está posicionado em cima de Y. Note que as precondições $X \neq Y$ e $X \neq Z$, existentes nos axiomas [BW6] e [BW8], respectivamente, servem para impedir que um bloco seja posicionado em cima de si mesmo.

2.1.3 Persistência temporal

Para que a formalização de um mundo dinâmico seja útil, além de descrever o que muda, precisamos descrever também aquilo que permanece inalterado. Por exemplo, sejam Σ a conjunção dos axiomas de observação [OB1]-[OB5], Δ a conjunção dos axiomas de efeito [BW1]-[BW5] e $s_1 := do(move(c,a,b), s_0)$ a situação que resulta quando o bloco c, que estava sobre a, é movido para cima do bloco b, conforme ilustrado na figura 2.2. Claramente, temos $\Sigma \wedge \Delta \models holds(ontable(a), s_0)$ e $\Sigma \wedge \Delta \models holds(on(c,b), s_1)$. Entretanto, apesar do bloco a estar sobre a mesa na situação s_0, e da ação $move(c,a,b)$ não alterar esse fato na situação s_1, não é possível demonstrar que $\Sigma \wedge \Delta \models holds(ontable(a), s_1)$.

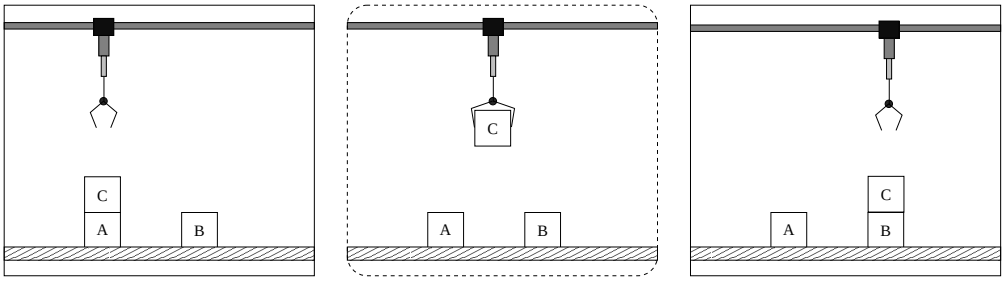


Figura 2.2: *A situação s_1 resulta da execução da ação $move(c, a, b)$ na situação s_0.*

Os axiomas de efeito conseguem descrever as mudanças que resultam da execução de uma ação, mas são incapazes de descrever aquilo que se mantém inalterado de uma situação para outra. Assim, para capturar a persistência dos fluentes que não são afetados por uma ação, precisamos ter *axiomas de persistência*[4]. Basicamente, um axioma de persistência estabelece que um fluente ϕ vale numa situação $do(\alpha, \sigma)$ se ele vale na situação σ e se a execução de α nessa situação é possível e não afeta o seu valor. Note que se a execução de α em σ não for possível, então não faz sentido falar sobre a validade de ϕ na situação resultante. Por exemplo, o axioma a seguir estabelece que, se é possível executar a ação *move* numa determinada situação, então, após a sua execução, todos os blocos que estavam sobre a mesa permanecerão sobre ela.

$$holds(ontable(V), do(move(X, Y, Z), S)) \leftarrow$$
$$\quad poss(move(X, Y, Z), S) \wedge holds(ontable(V), S)$$

Agora, tomando Δ' como Δ acrescido desse novo axioma, finalmente, podemos demonstrar que $\Sigma \wedge \Delta' \models holds(ontable(a), do(move(c, a, b), s_0))$, conforme era esperado.

Para garantir a persistência temporal de todos os fluentes no mundo dos blocos, precisamos dos seguintes axiomas de persistência temporal:

$$holds(on(V, W), do(stack(X, Y), S)) \leftarrow \qquad \text{[BW9]}$$
$$\quad poss(stack(X, Y), S) \wedge holds(on(V, W), S)$$

$$holds(clear(V), do(stack(X, Y), S)) \leftarrow \qquad \text{[BW10]}$$
$$\quad poss(stack(X, Y), S) \wedge holds(clear(V), S), V \neq Y$$

[4] Também denominados *axiomas de quadro* (*frame axioms*).

$holds(ontable(V), do(stack(X,Y),S)) \leftarrow$ [BW11]
$\quad poss(stack(X,Y),S) \wedge holds(ontable(V),S) \wedge V \neq X$

$holds(on(V,W), do(unstack(X,Y),S)) \leftarrow$ [BW12]
$\quad poss(unstack(X,Y),S) \wedge holds(on(V,W),S) \wedge V \neq X$

$holds(clear(V), do(unstack(X,Y),S)) \leftarrow$ [BW13]
$\quad poss(unstack(X,Y),S) \wedge holds(clear(V),S)$

$holds(ontable(V), do(unstack(X,Y),S)) \leftarrow$ [BW14]
$\quad poss(unstack(X,Y),S) \wedge holds(ontable(V),S)$

$holds(on(V,W), do(move(X,Y,Z),S)) \leftarrow$ [BW15]
$\quad poss(move(X,Y,Z),S) \wedge holds(on(V,W),S) \wedge V \neq X$

$holds(clear(V), do(move(X,Y,Z),S)) \leftarrow$ [BW16]
$\quad poss(move(X,Y,Z),S) \wedge holds(clear(V),S) \wedge V \neq Z$

$holds(ontable(V), do(move(X,Y,Z),S)) \leftarrow$ [BW17]
$\quad poss(move(X,Y,Z),S) \wedge holds(ontable(V),S)$

Axiomas de persistência representam apenas o óbvio e, por esse motivo, podem parecer completamente dispensáveis. Entretanto, sem eles, o cálculo de situações não teria a menor utilidade. Por outro lado, como veremos na próxima seção, é exatamente o uso desses axiomas que torna os planejadores dedutivos tão ineficientes na prática.

2.2 Problemas na formalização de mundos dinâmicos

Três problemas que normalmente surgem quando tentamos formalizar ações num mundo dinâmico são: o *problema da qualificação*, o *problema da ramificação* e o *problema da persistência*[5]. Os dois primeiros estão relacionados à dificuldade de se estabelecer de maneira precisa, respectivamente, as precondições e os efeitos das ações num mundo complexo. O último deles, o problema da persistência, está relacionado à dificuldade de se estabelecer, parcimoniosamente, que propriedades persistem de uma situação para outra, quando uma ação é executada. Note que, apesar de discutirmos esses problemas dentro do contexto particular do cálculo de situações, a sua natureza universal transcende esse formalismo específico [56].

[5] Também conhecido como *problema do quadro* (*frame problem*).

2.2.1 O problema da qualificação

O *problema da qualificação* [41] surge porque é muito difícil, se não for praticamente impossível, definir precisamente em que circunstâncias uma determinada ação pode ser *garantidamente* executada. Por exemplo, o axioma a seguir estabelece que o robô pode pegar um bloco se sua garra está vazia e se esse bloco está livre.

$$poss(pickup(X), S) \leftarrow holds(handempty, S) \wedge holds(clear(X), S)$$

Sem dúvida, essas precondições são *necessárias* para que a ação $pickup(X)$ possa ser executada. Entretanto, no mundo real, não podemos garantir que elas também sejam *suficientes*. Ainda que a garra do robô esteja vazia e que o bloco X esteja livre, o robô não poderá pegá-lo se, por exemplo, a sua garra estiver quebrada ou se esse bloco estiver colado na mesa. Na prática, o número de precondições necessárias para a execução de uma ação pode ser extraordinariamente grande e mesmo que fosse possível enumerar explicitamente cada uma delas, computacionalmente, isso seria muito ineficiente. Então, a saída é selecionar apenas precondições realmente *relevantes* que, dentro do contexto de discurso, possam ser consideradas, de fato, necessárias e suficientes.

2.2.2 O problema da ramificação

O *problema da ramificação* [20] está relacionado à dificuldade que existe em se enumerar explicitamente *todos* os efeitos que a execução de uma determinada ação pode causar num mundo suficientemente complexo. Por exemplo, na nossa axiomatização para o mundo dos blocos, os axiomas [BW4] e [BW5] estabelecem $clear(Y)$ e $on(X, Z)$ como sendo os únicos efeitos da ação $move(X, Y, Z)$; entretanto, se o bloco X estiver empoeirado, como conseqüência da execução dessa ação, a poeira sobre o bloco também será movida. Nesse caso, porém, como esse efeito colateral é irrelevante no contexto considerado, podemos simplesmente ignorá-lo, sem nenhum prejuízo. O problema da ramificação surge, de fato, apenas quando os efeitos colaterais de uma ação são tão importantes quanto seus efeitos primários. Por exemplo, suponha que um bloco X, que estivesse apoiado diretamente sobre a mesa, pudesse ser *empurrado* de um local P para outro local Q (*i.e.* $push(X, P, Q)$). Então, conforme estabelecido pelo axioma a seguir, o único efeito primário dessa ação seria a mudança da posição horizontal do bloco X.

$holds(at(X,Q), do(push(X,P,Q),S))$

Entretanto, nem sempre esse seria o único efeito observado. Se houvesse outros blocos em cima do bloco empurrado, então, como conseqüência da execução dessa ação, estes outros blocos também seriam movidos. Nesse caso, porém, seria melhor listar apenas os efeitos primários da ação e deixar que seus efeitos colaterais pudessem ser inferidos como conseqüência lógica desses efeitos primários e do conhecimento que o agente tem sobre o seu mundo.

No cálculo de situações, efeitos colaterais são descritos através de sentenças denominadas *axiomas de restrição de estado.* Diferentemente dos axiomas de efeito e de persistência, que relacionam conhecimento em duas situações distintas (antes e depois da execução de uma ação), os axiomas de restrição de estado relacionam conhecimento dentro de uma mesma situação. Por exemplo, o axioma de restrição de estado a seguir estabelece que, numa determinada situação S, um bloco X está numa posição P, se ele está sobre um outro bloco Y que, por sua vez, está na posição P.

$holds(at(X,P),S) \leftarrow holds(on(X,Y),S) \wedge holds(at(Y,P),S)$

Assim, com esse axioma, quando um bloco for empurrado de um local para outro, podemos inferir que aqueles blocos que estiverem sobre ele também o serão. É justamente a proliferação descontrolada desse tipo de axioma, necessário para estabelecer as conseqüências implícitas de uma ação, que dá origem ao problema da ramificação.

2.2.3 O problema da persistência

A *lei da inércia* estabelece que, normalmente, as coisas tendem a permanecer no mesmo estado em que se encontram. De fato, quando uma ação é executada, o número de propriedades do mundo que se mantêm fixas é muito maior do que o número de propriedades que mudam. Conseqüentemente, em qualquer axiomatização descrevendo um mundo não-trivial, o número de axiomas de persistência, que descrevem apenas conhecimento de senso comum, tende a ser bem maior que o número de axiomas de efeito, que realmente descrevem as ações do domínio. Por exemplo, na nossa axiomatização para o mundo dos blocos, que é trivial, do total de 17 axiomas que temos, 9 são axiomas de persistência ([BW9]-[BW17]).

O *problema da persistência* [43] surge exatamente da necessidade de se manter uma enorme quantidade de axiomas para garantir que os fluentes que não são afetados pela ocorrência de uma ação possam persistir no tempo, de uma situação para outra.

Para termos uma idéia sobre como cresce o número de axiomas de persistência, vamos estender o domínio do mundo dos blocos com a adição de um novo fluente $colour(X, C)$, denotando que a cor de um bloco X é C, e de uma nova ação $paint(X, C)$, denotando o ato de pintar um bloco X com uma nova cor C. Então, supondo que inicialmente todos os blocos sejam azuis e que pintar um bloco seja sempre possível, para comportar essa extensão, precisamos de um novo axioma de observação, um novo axioma de efeito e um novo axioma de precondições:

$holds(colour(X, blue), s_0)$
$holds(colour(X, C), do(paint(X, C), S))$
$poss(paint(X, C), S)$

Evidentemente, a cor de um bloco não se altera quando o movemos de um lugar para outro; mas, sem axiomas de persistência estabelecendo esse fato explicitamente, não será possível, por exemplo, mostrar que o bloco c permanecerá azul depois de ter sido movido sobre o bloco b. Outro fato óbvio que deverá ser estabelecido por um axioma de persistência é que a cor de um bloco não muda quando um outro bloco é pintado. Mas isso ainda não é o suficiente. A adição da ação *paint* requer também a inclusão de três novos axiomas de persistência para os fluentes *clear*, *ontable* e *on*.

$holds(colour(V, W), do(stack(X, Y), S)) \leftarrow$
$\quad poss(stack(X, Y), S) \wedge holds(colour(V, W), S)$
$holds(colour(V, W), do(unstack(X, Y), S)) \leftarrow$
$\quad poss(unstack(X, Y), S) \wedge holds(colour(V, W), S)$
$holds(colour(V, W), do(stack(X, Y), S)) \leftarrow$
$\quad poss(move(X, Y, Z), S) \wedge holds(colour(V, W), S)$
$holds(colour(V, W), do(paint(X, C), S)) \leftarrow$
$\quad poss(paint(X, C), S) \wedge holds(colour(V, W), S) \wedge V \neq X$
$holds(clear(V), do(paint(X, C), S)) \leftarrow$
$\quad poss(paint(X, C), S) \wedge holds(clear(V), S)$
$holds(ontable(V), do(paint(X, C), S)) \leftarrow$

$$poss(paint(X,C),S) \wedge holds(ontable(V),S)$$
$$holds(on(V,W),do(paint(X,C),S)) \leftarrow$$
$$poss(paint(X,C),S) \wedge holds(on(V,W),S) \wedge W \neq X$$

Como podemos observar, dos 26 axiomas necessários na formalização dessa nova versão do mundo dos blocos[6], 16 são axiomas de persistência. De modo geral, como a maioria dos fluentes num domínio não são afetados pela execução de uma particular ação, para cada ação considerada, haverá necessidade de um axioma de persistência para cada um dos fluentes do domínio. Sendo assim, num domínio com m ações e n fluentes, haverá $O(m \times n)$ axiomas de persistência.

2.3 Uma solução para o problema da persistência

Como os axiomas de persistência representam apenas conhecimento de senso comum, intuitivamente, temos a impressão de que poderíamos dispensá-los. O ideal seria termos somente axiomas de efeito, supor que nenhuma mudança ocorre além daquelas resultantes desses axiomas e, então, empregar um formalismo que pudesse derivar corretamente as conseqüências esperadas. De fato, isso pode ser feito através de circunscrição.

2.3.1 Circunscrição

A idéia básica da *circunscrição* [41, 42, 38] é minimizar a extensão de um predicado, *i.e.* limitar o conjunto de objetos para os quais ele deve ser verdadeiro. Por exemplo, seja $\Sigma := clear(b) \wedge clear(c) \wedge on(c,a)$. Claramente, temos $\Sigma \models clear(b)$ e $\Sigma \models clear(c)$; mas, devido à neutralidade da lógica clássica, na ausência de informação explícita, não há como demonstrar $\Sigma \models clear(a)$, nem $\Sigma \models \neg clear(a)$. Seja $\textsc{Circ}[\Sigma; clear]$ a circunscrição de Σ minimizando a extensão do predicado $clear$, *i.e* tornando-o falso para todos os objetos do universo de discurso, exceto para aqueles que a sentença Σ o força a ser verdadeiro. Então, temos que $\textsc{Circ}[\Sigma; clear] \models \neg clear(a)$. De modo geral, para esse exemplo, temos que $\textsc{Circ}[\Sigma; clear] \models \forall X[clear(X) \leftrightarrow (X = b \vee X = c)]$.

[6] Excluindo-se os 5 axiomas de observação que descrevem a situação inicial.

Para uma definição mais formal de circunscrição, vamos usar a seguinte notação: sejam ρ_1, ρ_2 predicados n-ários e $\overline{x}$ uma tupla de n variáveis distintas, então:

- $\rho_1 = \rho_2$ abrevia $\forall \overline{x}(\rho_1(\overline{x}) \leftrightarrow \rho_2(\overline{x}))$;
- $\rho_1 \leq \rho_2$ abrevia $\forall \overline{x}(\rho_1(\overline{x}) \rightarrow \rho_2(\overline{x}))$;
- $\rho_1 < \rho_2$ abrevia $(\rho_1 \leq \rho_2) \wedge \neg(\rho_1 = \rho_2)$.

Definição 2.1 *A circunscrição de ϕ minimizando ρ, escrita como* CIRC$[\phi;\rho]$*, é a fórmula $\phi \wedge \neg\exists Q[\phi(Q) \wedge Q < \rho]$, onde $\phi(Q)$ é a fórmula obtida pela substituição de toda ocorrência de ρ em ϕ por Q.* □

De acordo com essa definição, a circunscrição da sentença ϕ minimizando o predicado ρ deve satisfazer ϕ, mais a exigência de que a extensão de ρ seja tão pequena quanto $\phi(Q)$ permite que ela seja. Note que sempre que uma nova fórmula é conectada a ϕ, uma fórmula correspondente também é conectada a $\phi(Q)$, que está dentro da parte quantificada existencialmente na circunscrição. Isso garante que todas as conseqüências lógicas da sentença original sejam mantidas após a circunscrição.

Por exemplo, considere novamente $\Sigma := clear(b) \wedge clear(c) \wedge on(c,a)$. Claramente, temos CIRC$[\Sigma; on] \models \neg on(b,d)$. Entretanto, uma vez disponível a informação de que o bloco b está realmente sobre o bloco d, a conclusão $\neg on(b,d)$ deverá ser descartada, ou seja, CIRC$[\Sigma \wedge on(b,d); on] \not\models \neg on(b,d)$. Sendo assim, podemos dizer que a circunscrição implica em um tipo de raciocínio não-monotônico que nos permite chegar a conclusões razoáveis, na ausência de informação em contrário, mas que não sejam estritamente garantidas pelos fatos estabelecidos. Uma futura adição de informação poderá descartar tais conclusões, sem no entanto causar inconsistência.

2.3.2 Cálculo de situações circunscritivo

Inspirado na lei da inércia, *McCarthy* propôs a conjectura de que "*um fluente não muda, a menos que ocorra um evento que o afete*" [41, 56]. Essa conjectura, juntamente com a circunscrição, é a base para a solução do problema da persistência encontrado na formalização de mundos dinâmicos.

Para resolver o problema da persistência no cálculo de situações, usando circunscrição, precisamos substituir todos os axiomas de persistência por um único axioma genérico, independente de domínio, denominado *axioma de persistência universal*. Esse axioma, que corresponde diretamente à conjectura de *McCarthy*, estabelece que os fluentes persistem, a menos que sejam afetados pela execução de uma ação.

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

Assim, uma vez especificado que ações afetam que fluentes do domínio[7] (*vide* tabela 2.4 - página 26), a circunscrição da axiomatização, minimizando o predicado $affects$, nos permitirá deduzir quais são os fluentes que não são afetados por uma determinada ação e que, portanto, devem persistir na situação resultante da execução dessa ação.

2.4 Planejamento dedutivo no cálculo de situações

Uma das principais características do cálculo de situações é que um termo designando uma situação carrega informações sobre o histórico do mundo. Por exemplo, o termo $do(stack(c, b), do(unstack(c, a), s_0))$ denota a situação que seria atingida se, a partir da situação inicial s_0, desempilhássemos o bloco c de cima de a e, em seguida, o empilhássemos sobre o bloco b. Reciprocamente, dada uma situação qualquer, podemos obter uma nova situação simplesmente executando uma seqüência de ações (plano) a partir dela. Esse isomorfismo tão natural entre situações e planos é que torna o cálculo de situações apropriado para resolver problemas de planejamento.

Usando o cálculo de situações, a tarefa de planejamento pode ser vista como sendo, essencialmente, um problema de dedução. De fato, essa abordagem lógica possibilita uma estreita correspondência entre *especificação* e *implementação*. Como veremos na subseção 2.4.3, os mesmos axiomas usados na especificação de um problema de planejamento servem de base para implementação de um planejador capaz de solucioná-lo.

[7] Na verdade essas informações poderiam ser compiladas, automaticamente, a partir dos axiomas de efeito que fazem parte da axiomatização do domínio.

2.4.1 O método de *Green*

Definição 2.2 *Um problema de planejamento num determinado domínio é definido por uma tupla da forma* $\langle \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$*, onde* $\mathcal{A}$ *é uma descrição das ações do agente nesse domínio,* $\mathcal{I}$ *é uma descrição da situação inicial do mundo desse agente e* $\mathcal{G}$ *é uma descrição da sua meta.* □

Conforme *Green* [23, 21] mostrou, dada a especificação lógica de um problema de planejamento, uma solução para ele pode ser obtida através da prova de um teorema. Por exemplo, suponha que a nossa meta seja atingir uma situação no mundo dos blocos em que o bloco c esteja sobre o bloco b. Então, esse problema pode ser especificado da seguinte maneira:

- $\mathcal{A}$: a conjunção dos axiomas [BW1]-[BW17], que descrevem as ações do domínio;
- $\mathcal{I}$: a conjunção dos axiomas [OB1]-[OB5], que descrevem a situação inicial;
- $\mathcal{G}$: a sentença $(\exists S) holds(on(c,b), S)$, que descreve a meta de planejamento.

Observe que a sentença $(\exists S) holds(on(c,b), S)$ é, na verdade, uma conjectura de que existe uma situação S em que a meta de planejamento $on(c,b)$ pode ser satisfeita. Então, se uma prova construtiva para essa conjectura instanciar a variável S a uma situação $do(\alpha_n, do(\ldots, do(\alpha_1, s_0)))$, a seqüência de ações $\langle \alpha_1, \ldots, \alpha_n \rangle$ correspondente, claramente, poderá ser uma solução para o nosso problema de planejamento.

Note porém que, sintaticamente, não há qualquer restrição quanto às ações que compõem uma situação e, sendo assim, é possível termos uma situação cuja seqüência de ações correspondente não seja executável no mundo real. Por exemplo, a partir dos axiomas de efeito [BW1] e [BW5] podemos derivar, respectivamente, as sentenças $holds(on(c,b), do(stack(c,b), s_0))$ e $holds(on(c,b), do(move(c,a,b), s_0))$. Então, $do(stack(c,b), s_0)$ e $do(move(c,a,b), s_0)$ são duas das possíveis instanciações que, através de uma prova construtiva, poderíamos obter para a variável S em $(\exists S) holds(on(c,b), S)$. Porém, de acordo com o axioma de precondições [BW6], não é possível executar a ação $stack(c,b)$ na situação s_0 e, sendo assim, apenas a situação $do(move(c,a,b), s_0)$ poderia ser tomada como uma *solução* para o nosso problema de planejamento.

De fato, uma condição necessária para que uma situação possa ser uma solução para um problema de planejamento é que ela seja *executável*. Por definição, uma situação

σ é executável se é s_0 ou, então, se é da forma $do(\alpha, \sigma')$, as precondições da ação α estão satisfeitas na situação σ' e essa situação é também executável. Os *axiomas de executabilidade* a seguir expressam essa definição:

$exec(s_0)$
$exec(do(A, S)) \leftarrow poss(A, S) \wedge exec(S)$

Agora, podemos então definir a tarefa de planejamento da seguinte forma:

Definição 2.3 *Seja $\langle \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ um problema de planejamento. Então, a tarefa de planejamento consiste em provar que $\mathcal{A} \wedge \mathcal{I} \models (\exists S).exec(S) \wedge \mathcal{G}(S)$. Se a prova for construtiva, a instanciação obtida para S será uma solução para o problema de planejamento.* □

Uma das vantagens desse método proposto por *Green* é que ele pode ser implementado diretamente como um programa lógico. Além disso, esse método é *correto*, *i.e.* produz apenas soluções corretas para os problemas de planejamento, e também *completo*, *i.e.* encontra uma solução sempre que há uma [1, 21].

2.4.2 Programação em lógica

Programação em lógica é uma teoria baseada em lógica de predicados de primeira ordem, com sintaxe e semântica definidas de modo usual. Para maior clareza, entretanto, empregaremos a terminologia e notação própria desse formalismo [1].

Uma *constante*, *função* ou *predicado* é uma seqüência alfanumérica iniciando com minúscula e uma *variável* é uma seqüência alfanumérica iniciando com maiúscula. Um *termo* é uma constante, uma variável ou uma função seguida de uma lista de termos. Um *átomo* é um predicado seguido de uma lista de termos. Um *literal* é um átomo (*literal positivo*) ou a negação de um átomo (*literal negativo*). Uma *cláusula* é uma sentença da forma $\phi_1 \vee \ldots \vee \phi_m$, onde cada ϕ_i é um literal. Uma *cláusula de Horn* é uma cláusula com no máximo um literal positivo. Uma *cláusula definida* é uma cláusula de Horn com exatamente um literal positivo. Cláusulas definidas são normalmente escritas como $\alpha \leftarrow \beta_1, \ldots, \beta_n$; nesse caso, dizemos que α é a *cabeça* e $\beta_1, \ldots, \beta_n$ formam o *corpo* da cláusula. Uma *cláusula unitária* ou *fato*, escrita como $\alpha \leftarrow$, é uma cláusula definida que não contém literais negativos. Uma *cláusula objetivo* ou *consulta* é uma cláusula sem

literais positivos, escrita como $\leftarrow \beta_1, \ldots, \beta_n$. E, finalmente, a cláusula vazia, escrita como $\leftarrow$, é a cláusula sem literais, interpretada como uma contradição.

Há dois significados para uma cláusula da forma $\alpha \leftarrow \beta_1, \ldots, \beta_n$:

- *declarativo:* α é verdade se $\beta_1, \ldots, \beta_n$ também são verdade.
- *procedimental:* para solucionar α, solucione $\beta_1, \ldots, \beta_n$.

Dentre eles, é o significado procedimental que permite que a lógica formal seja usada como uma linguagem de programação, distinguindo, assim, a lógica de predicados clássica da programação em lógica.

SLD-Resolução e SLDNF-Resolução

Um *programa lógico* é um conjunto finito e não vazio de cláusulas definidas, através do qual podemos realizar computações combinando dois mecanismos básicos: *unificação* e *resolução* [51]. Essa forma de computação é, na verdade, um método específico de prova de teoremas, considerado correto e completo [1, 10].

Formalmente, uma substituição é um mapeamento finito de variáveis em termos, denotado por $\theta := \{X_1/t_1, \ldots, X_n/t_n\}$, onde $X_1, \ldots, X_n$ são variáveis distintas e cada X_i é distinto de t_i, para $i = 1, \ldots, n$. Substituições são aplicáveis a expressões da linguagem, ou seja, termos, átomos e cláusulas. Se E é uma expressão, então $E\theta$ denota a expressão que se obtém substituindo-se, simultaneamente, cada ocorrência de X_i em E pelo termo t_i correspondente, de acordo com θ. Dizemos que a expressão $E\theta$ é uma *instância* de E. Uma cláusula C_1 é uma *variante* de uma cláusula C_2 se existem substituições θ_1 e θ_2 tais que $C_1\theta_1$ é idêntico a C_2 e $C_2\theta_2$ é idêntico a C_1. Essencialmente, uma variante de uma cláusula é essa mesma cláusula com algumas de suas variáveis renomeadas. Dois átomos A_1 e A_2 são *unificáveis* se existe uma substituição θ tal que $A_1\theta$ é idêntico a $A_2\theta$; nesse caso, dizemos que θ é um *unificador* para esses átomos. Um unificador θ_1 é um *unificador mais geral* (*u.m.g.*) para A_1 e A_2 se para todo unificador θ_2 de A_1 e A_2 existe uma substituição θ tal que $A_1\theta_1\theta$ é idêntico a $A_1\theta_2$.

Seja $\mathcal{P}$ um programa lógico. Sejam $G_0 := \leftarrow \alpha_1, \ldots, \alpha_m$ uma cláusula objetivo e $V_0 := \alpha \leftarrow \beta_1, \ldots, \beta_n$ uma variante de uma das cláusulas de $\mathcal{P}$, sem variáveis em comum com G_0. Suponha que, para algum i, os átomos α_i em G_0 e α em V_0 sejam

unificáveis com um *u.m.g.* θ. Então, podemos *resolver* essas duas cláusulas e obter a cláusula $G_1 := (\leftarrow \alpha_1, \ldots, \alpha_{i-1}, \beta_1, \ldots, \beta_n, \alpha_{i+1}, \ldots, \alpha_k)\theta$ como *resolvente*. Nesse caso, dizemos que α_i é o *literal selecionado* da cláusula objetivo. Iterando esse passo, ou seja, repetindo o processo com a resolvente G_1 e uma nova variante V_1 de uma das cláusulas do programa $\mathcal{P}$, obtemos uma seqüência de resolventes chamada *derivação*.

Uma SLD-*resolução*[8] é uma derivação maximal. Se uma tal derivação termina com a cláusula vazia, temos então uma SLD-*refutação*. Por exemplo, a tabela 2.3 mostra uma SLD-*refutação* para a cláusula objetivo $\leftarrow holds(ontable(c), S), exec(S)$, a partir do programa composto pelas cláusulas definidas $\mathcal{P}_1$,...,$\mathcal{P}_7$. Nessa refutação, uma anotação "$\mathcal{P}_j, \theta$" à direita de uma cláusula objetivo G_i indica que essa cláusula é uma resolvente entre a cláusula G_{i-1} e uma variante da cláusula $\mathcal{P}_j$, com *u.m.g.* θ (os literais selecionados para resolução estão em negrito). Observe que, compondo todos os *u.m.g.*'s usados nessa refutação, obtemos $S/do(unstack(c,a), s_0)$. Essa instanciação, sintetizada como efeito colateral da prova do teorema, é justamente um plano executável para atingir uma situação onde o bloco c está sobre a mesa.

$holds(clear(c), s_0) \leftarrow$	$\mathcal{P}_1$
$holds(ontable(a), s_0) \leftarrow$	$\mathcal{P}_2$
$holds(on(c,a), s_0) \leftarrow$	$\mathcal{P}_3$
$holds(ontable(X), do(unstack(X,Y), S)) \leftarrow$	$\mathcal{P}_4$
$poss(unstack(X,Y), S) \leftarrow holds(clear(X), S), holds(on(X,Y), S)$	$\mathcal{P}_5$
$exec(s_0) \leftarrow$	$\mathcal{P}_6$
$exec(do(A,S)) \leftarrow poss(A,S), exec(S)$	$\mathcal{P}_7$
$\leftarrow \mathbf{holds(ontable(c), S)}, exec(S)$	G_0
$\leftarrow \mathbf{exec(do(unstack(c, Y_1), S_1))}$	$\mathcal{P}_4, \{X_1/c, S/do(unstack(X_1, Y_1), S_1)\}$
$\leftarrow \mathbf{poss(unstack(c, Y_1), S_2)}, exec(S_2)$	$\mathcal{P}_7, \{A_2/unstack(c, Y_1), S_1/S_2\}$
$\leftarrow \mathbf{holds(clear(c), S_3)}, holds(on(c, Y_3), S_3), exec(S_3)$	$\mathcal{P}_5, \{X_3/c, Y_1/Y_3, S_2/S_3\}$
$\leftarrow \mathbf{holds(on(c, Y_3), s_0)}, exec(s_0)$	$\mathcal{P}_1, \{S_3/s_0\}$
$\leftarrow \mathbf{exec(s_0)}$	$\mathcal{P}_3, \{Y_3/a\}$
$\leftarrow$	$\mathcal{P}_6, \{\}$

Tabela 2.3: *Um exemplo de* SLD-*refutação.*

[8] *Linear resolution with Selection rule on Definite clauses.*

Se literais negativos são permitidos no corpo das cláusulas, então a derivação é denominada SLDNF-*resolução*[9]. Nesse tipo de derivação, quando o literal selecionado é positivo, para se obter a resolvente, procedemos da mesma forma que na SLD-resolução; quando é negativo, digamos $\neg\alpha$, usamos a seguinte regra para obter a resolvente:

- $\neg\alpha$ é bem sucedido se e somente se α falha finitamente,
- $\neg\alpha$ falha finitamente se e somente se α é bem sucedido.

Quando $\neg\alpha$ é bem sucedido, ele é removido da cláusula objetivo; quando falha, a derivação é interrompida. Essa regra, denominada *negação por falha*, é correta e completa com relação ao modelo de *completamento de Clark* de programas lógicos [1, 10].

2.4.3 Um planejador dedutivo em PROLOG

PROLOG (**Pro***gramming in* **Log***ic*) [11, 58] é um sistema de prova automática de teoremas, baseado em SLDNF-refutação, com as seguintes restrições:

- seleciona sempre o primeiro átomo à esquerda na cláusula objetivo para resolver com a cabeça de uma cláusula do programa;
- escolhe as cláusulas do programa na ordem em que elas são escritas, da primeira para a última, e emprega uma estratégia de busca em profundidade.

A estratégia de *busca em profundidade* é eficiente tanto em termos de tempo quanto de espaço; porém, não é completa. Conseqüentemente, o PROLOG não garante encontrar uma refutação para uma determinada cláusula objetivo, mesmo que uma tal refutação exista. Por outro lado, a estratégia de *busca em largura* é completa, mas é também muito ineficiente em termos de espaço. Assim, uma alternativa melhor, seria utilizar *busca em profundidade iterativa*, que é quase tão eficiente quanto busca em profundidade e completa como a busca em largura [53]. Especialmente em planejamento, a busca em profundidade iterativa tem ainda a vantagem de garantir que o primeiro plano encontrado seja *mínimo*, *i.e.* um plano com o menor número de passos possível.

[9] SLD *with Negation as Failure.*

A estratégia de busca em profundidade iterativa para o cálculo de situações pode ser facilmente implementada em PROLOG através das seguintes cláusulas:

```
(1) plan(s0).
(2) plan(do(A,S)) :- plan(S).
```

Usando essas cláusulas, e a capacidade de retrocesso automático do PROLOG, podemos forçar o sistema a considerar uma seqüência infinita de planos, de tamanhos crescentes, conforme ilustrado na figura 2.3.

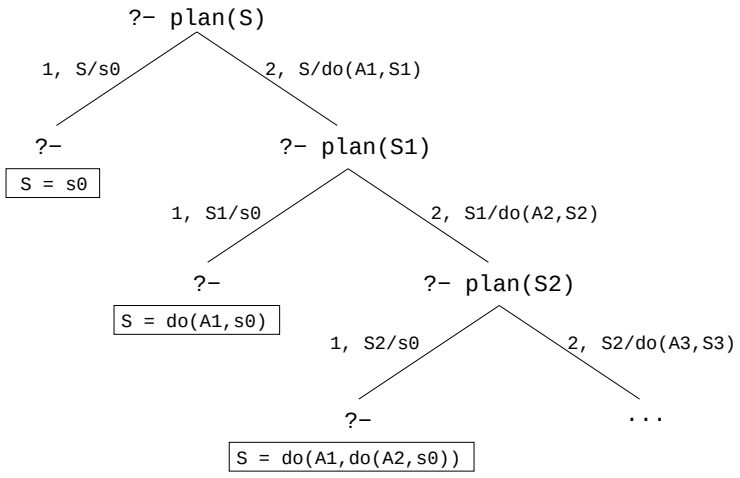


Figura 2.3: *Busca em profundidade iterativa.*

Outro aspecto interessante que deve ser ressaltado é que o mecanismo de negação por falha do PROLOG possibilita um tipo de raciocínio não-monotônico, através do qual é possível implementar um cálculo de situações circunscritivo. Assim, conforme vimos na subseção 2.3.2, em vez dos numerosos axiomas de persistência específicos do domínio, podemos usar um único axioma de persistência universal.

A tabela 2.4 mostra o código PROLOG que implementa um planejador para o mundo dos blocos, diretamente a partir de sua especificação formal. Nesse código, o predicado *plan* é o único tipo de controle que exercemos no planejador, sendo que as demais cláusulas correspondem exatamente à sua especificação lógica no cálculo de situações.

```
% situação inicial
  holds(clear(b),s0).
  holds(clear(c),s0).
  holds(ontable(a),s0).
  holds(ontable(b),s0).
  holds(on(c,a),s0).

% axiomas de efeito
  holds(on(X,Y),do(stack(X,Y),S)).
  holds(clear(Y),do(unstack(X,Y),S)).
  holds(ontable(X),do(unstack(X,Y),S)).
  holds(clear(Y),do(move(X,Y,Z),S)).
  holds(on(X,Z),do(move(X,Y,Z),S)).

% axiomas de precondições
  poss(stack(X,Y),S) :-
      holds(ontable(X),S), holds(clear(X),S), holds(clear(Y),S), X\=Y.
  poss(unstack(X,Y),S) :-
      holds(clear(X),S), holds(on(X,Y),S).
  poss(move(X,Y,Z),S) :-
      holds(clear(X),S), holds(clear(Z),S), holds(on(X,Y),S), X\=Z.

% axioma de persistência universal
  holds(F,do(A,S)) :-
      poss(A,S), holds(F,S), not affects(A,F).

  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)).

% implementa busca em profundidade iterativa
  plan(s0).
  plan(do(A,S)) :- plan(S).

% verifica executabilidade do plano
  exec(s0).
  exec(do(A,S)) :- poss(A,S), exec(S).
```

Tabela 2.4: *Um sistema de planejamento para o mundo dos blocos em* PROLOG.

Agora, usando os predicados *plan* e *exec*, podemos implementar um procedimento que enumera sistematicamente todos os planos executáveis no mundo dos blocos, conforme mostra a tabela 2.5.

```
1 ?- plan(S), exec(S).
S = s0 ;
S = do(stack(b,c),s0) ;
S = do(unstack(c,a),s0) ;
S = do(move(c,a,b),s0) ;
S = do(stack(c,a),do(unstack(c,a),s0)) ;
S = do(stack(c,b),do(unstack(c,a),s0)) ;
S = do(stack(a,b),do(unstack(c,a),s0)) ;
S = do(stack(a,c),do(unstack(c,a),s0)) ;
S = do(stack(b,a),do(unstack(c,a),s0)) ;
S = do(stack(b,c),do(unstack(c,a),s0)) ;
S = do(stack(a,c),do(move(c,a,b),s0)) ;
S = do(unstack(b,c),do(stack(b,c),s0)) ;
S = do(unstack(c,b),do(move(c,a,b),s0)) ;
S = do(move(c,b,a),do(move(c,a,b),s0)) ;
S = do(stack(b,c),do(unstack(b,c),do(stack(b,c),s0))) ;
S = do(stack(c,b),do(unstack(c,b),do(move(c,a,b),s0))) ;
...
```

Tabela 2.5: *Enumeração de planos executáveis.*

De fato, $\{S \mid plan(S) \wedge exec(S)\}$ é o conjunto de todas as situações que podem ser atingidas através de seqüências de ações executáveis, a partir da situação s_0. Ademais, se ϕ é a meta num determinado problema de planejamento, então $\{S \mid plan(S) \wedge exec(S) \wedge holds(\phi, S)\}$ é o conjunto de todas as soluções possíveis para esse problema. Assim, em termos procedimentais, é como se $\{S \mid plan(S) \wedge exec(S)\}$ fosse o conjunto de todos os nós da árvore de busca para o problema considerado e $holds(\phi, S)$ fosse a função capaz de identificar aqueles nós que representam situações onde a meta de planejamento é satisfeita.

A tabela 2.6 mostra alguns exemplos de consultas que podem ser feitas ao sistema de planejamento lógico obtido para o mundo dos blocos:

- Em (1), a meta é atingir uma situação em que o bloco a esteja sobre c.

- Em (2), a meta é atingir uma situação em que o bloco a esteja sobre b e esse, por sua vez, esteja sobre c. Esse problema, conhecido como *anomalia de Sussman*, é um exemplo de típico de *conflito de submetas*: se a é empilhado sobre b, então b não pode mais ser empilhado sobre c; por outro lado, se b é empilhado sobre c, que está sobre a, esse último não pode mais ser empilhado sobre b.

- Em (3), a meta é atingir uma situação em que haja uma pilha com (pelo menos) três blocos. O interessante é que essa meta não especifica que blocos.

- Finalmente, em (4), verificamos que fluentes valem na situação resultante da execução da seqüência de ações $\langle unstack(c,a), stack(b,c), stack(a,b)\rangle$. Esse tipo de problema é conhecido como *projeção temporal*.

```
1 ?- plan(S), exec(S), holds(on(a,c),S).
S = do(stack(a,c), do(unstack(c,a), s0))
Yes

2 ?- plan(S), exec(S), holds(on(a,b),S), holds(on(b,c),S).
S = do(stack(a,b), do(stack(b,c), do(unstack(c,a), s0)))
Yes

3 ?- plan(S), exec(S), holds(on(X,Y),S), holds(on(Y,Z),S).
S = do(stack(b,c),s0)))
X = b
Y = c
Z = a
Yes

4 ?- holds(F,do(stack(a,b),do(stack(b,c),do(unstack(c,a),s0)))).
F = on(a, b) ;
F = on(b, c) ;
F = clear(a) ;
F = ontable(c) ;
No
```

Tabela 2.6: *Exemplos de consultas ao sistema de planejamento em* PROLOG.

2.5 Considerações finais

Nesse capítulo, introduzimos um formalismo especialmente criado para modelagem de mundos dinâmicos – o *cálculo de situações* – e mostramos como podemos utilizá-lo para representar ações e raciocinar sobre seus efeitos. Em seguida, discutimos os problemas que surgem quando tentamos formalizar mundos dinâmicos, especialmente o problema da persistência, e as dificuldades que esses problemas impõem a um sistema de planejamento dedutivo. Finalmente, mostramos que, usando prova automática de teoremas, um sistema de planejamento pode ser implementado diretamente a partir de sua especificação lógica em cálculo de situações. Isso, sem dúvida, é uma grande vantagem na abordagem lógica de planejamento.

No próximo capítulo, como uma alternativa ao cálculo de situações, introduziremos o STRIPS, cuja principal vantagem é eliminar a necessidade de axiomas de persistência temporal e permitir, assim, um acréscimo de eficiência computacional. Então, com base nessa nova representação de ações, apresentaremos uma visão geral de planejamento clássico em Inteligência Artificial, segundo a abordagem algorítmica.

Capítulo 3

Planejamento algorítmico

Quem quiser alcançar um objetivo distante
tem que dar muitos passos curtos.

Helmut Schmidt

Economista e Político

3.1 A representação STRIPS

A representação STRIPS (**ST***anford* **R***esearch* **I***nstitute* **P***roblem* **S***olver*) foi proposta por *Fikes* e *Nilsson* [18], no início da década de 1970, como uma alternativa ao cálculo de situações. Desde então, essa representação tem sido amplamente utilizada na descrição de ações nos sistemas de planejamento desenvolvidos dentro da abordagem algorítmica.

Originalmente, a representação STRIPS permitia que estados e ações fossem descritos através de fórmulas arbitrárias da lógica de predicados de primeira ordem. Essa generalidade, entretanto, impossibilitou que uma semântica clara e precisa fosse estabelecida para essa representação. Em decorrência desse fato, várias restrições foram feitas quanto às fórmulas que poderiam ser usadas corretamente nessa representação [37]. Com essas restrições, a versão proposicional do STRIPS, na qual apenas fórmulas atômicas livres de variáveis são permitidas, passou a ser a representação de ações mais comumente empregada dentro da abordagem algorítmica de planejamento.

3.1.1 Estados e ações

Em STRIPS, um *estado* é representado por um conjunto de átomos que denotam as propriedades que valem no mundo e uma *ação* é representada por um operador que transforma um determinado estado em outro, através da adição ou remoção de átomos no conjunto que representa esse estado.

Um *operador* α é definido por um conjunto de precondições, $pre(\alpha)$, *i.e.* átomos denotando propriedades do mundo que devem valer para que a ação α possa ser executada, e por um conjunto de poscondições, $pos(\alpha)$, *i.e.* literais denotando propriedades do mundo que passam a valer, ou que deixam de valer, como conseqüência da execução da ação α. Por exemplo, considerando ainda o domínio do mundo dos blocos introduzido no capítulo anterior (página 7), o operador $move(c, a, b)$, definido a seguir, descreve a ação de mover o bloco c, de cima de a, para cima do bloco b.

$$
\begin{aligned}
&oper(act\ : move(c,a,b),\\
&\quad pre\ : \{clear(c), clear(b), on(c,a)\},\\
&\quad pos\ : \{clear(a), on(c,b), \neg clear(b), \neg on(c,a)\})
\end{aligned}
$$

Uma suposição implícita na representação STRIPS é que o conjunto $pos(\alpha)$ representa explicitamente *todos* os efeitos da ação α. Esse conjunto pode ser particionado em dois subconjuntos disjuntos, $add(\alpha)$ e $del(\alpha)$, contendo átomos que denotam, respectivamente, os efeitos positivos e negativos da ação α. Assim, uma forma alternativa bastante comum para o operador $move(c, a, b)$ seria a seguinte:

$$
\begin{aligned}
&oper(act\ : move(c,a,b),\\
&\quad pre\ : \{clear(c), clear(b), on(c,a)\},\\
&\quad add\ : \{clear(a), on(c,b)\},\\
&\quad del\ : \{clear(b), on(c,a)\})
\end{aligned}
$$

Dizemos que uma ação α é *aplicável* a um estado Σ se e somente se $pre(\alpha) \subseteq \Sigma$, ou seja, se suas precondições são satisfeitas nesse estado. Por exemplo, seja $\Sigma_{BW} := \{clear(b), clear(c), ontable(a), ontable(b), on(c,a)\}$ o conjunto que representa o estado inicial para o mundo dos blocos. Então, como $pre(move(c,a,b)) \subseteq \Sigma_{BW}$, segue que

$move(c, a, b)$ é aplicável ao estado Σ_{BW}. Se uma ação α é aplicável a um determinado estado Σ, então o estado *resultante* de sua aplicação a esse estado é representado pelo conjunto $\Sigma - del(\alpha) + add(\alpha)$. Por exemplo, o estado resultante da aplicação de $move(c, a, b)$ a Σ_{BW} é $\{clear(a), clear(c), ontable(a), ontable(b), on(c, b)\}$. Note que, como o conjunto $del(move(c, a, b))$ representa *todos* os efeitos negativos da ação $move(c, a, b)$, todas as propriedades que valem em Σ_{BW} e não são afetadas pela execução dessa ação (*e.g.* $clear(c)$, $ontable(a)$ e $ontable(b)$) continuam valendo no estado resultante. É por isso que em STRIPS não temos necessidade dos axiomas de persistência temporal.

Indutivamente, podemos definir o estado $res(\Sigma, \pi)$, resultante da aplicação de uma seqüência de ações $\pi := \langle \alpha_1, \ldots, \alpha_n \rangle$ a um estado Σ, da seguinte maneira:

(*a*) $res(\Sigma, \langle\rangle) \equiv \Sigma$,
(*b*) $res(\Sigma, \langle \alpha_1, \ldots, \alpha_n \rangle) \equiv res(\Sigma, \langle \alpha_1, \ldots, \alpha_{n-1} \rangle) - del(\alpha_n) + add(\alpha_n)$,

com a exigência de que cada ação α_i seja aplicável ao estado $res(\Sigma, \langle \alpha_1, \ldots, \alpha_{i-1} \rangle)$, para $i = 1, \ldots, n$. Caso contrário, o estado resultante é indefinido.

Outra suposição importante que fica implícita na representação STRIPS é que o estado inicial do mundo é sempre completamente definido e, de acordo com a *hipótese do mundo fechado* [50], a ausência de um átomo na descrição desse estado implica que, inicialmente, a sua negação é verdadeira. Essa suposição, juntamente com aquela de que toda mudança causada por uma ação é explicitamente descrita na sua representação, garante que um estado resultante seja também completamente definido.

3.1.2 Problemas e planos

Definição 3.1 *Um problema de planejamento em* STRIPS *é definido por uma tupla* $\langle \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, *onde* $\mathcal{A}$ *é um conjunto de operadores que descrevem as ações do domínio,* $\mathcal{I}$ *é um conjunto de átomos que descrevem o estado inicial e* $\mathcal{G}$ *é um conjunto de átomos que descrevem a meta de planejamento.* □

Diferentemente do estado inicial, que é completamente definido pelo conjunto $\mathcal{I}$, um *estado final* ou *estado meta* é apenas parcialmente definido pelo conjunto $\mathcal{G}$. Por exemplo, se temos um problema de planejamento onde $\mathcal{G} := \{ontable(a)\}$, então qualquer estado onde o bloco a esteja sobre a mesa é um estado meta para esse problema,

independentemente da configuração dos demais blocos existentes. De fato, enquanto o conjunto $\mathcal{I}$ especifica um único estado, o conjunto $\mathcal{G}$ representa um conjunto de estados.

Definição 3.2 *Sejam $\langle \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ um problema de planejamento em* STRIPS *e π uma seqüência de ações, onde cada ação é uma instância de um dos operadores especificados em $\mathcal{A}$. Dizemos que π é uma solução para o problema $\langle \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ se e só se $\mathcal{G} \subseteq res(\mathcal{I}, \pi)$, ou seja, se executando π a partir do estado inicial $\mathcal{I}$ atingimos um estado final, onde a conjunção dos átomos em $\mathcal{G}$ é satisfeita.* □

3.1.3 Complexidade dos problemas de planejamento

A dificuldade de um problema de planejamento depende, essencialmente, de como os operadores do domínio, necessários para atingir as diferentes submetas do problema, interagem entre si. De acordo com a taxonomia introduzida por *Korf* [31], um conjunto de submetas (ou problema de planejamento) pode ser classificado como:

- *independente*, se cada operador atinge uma única submeta e não remove precondições de nenhum outro operador do domínio. Uma propriedade importante de submetas independentes, que segue da definição, é que uma solução ótima global pode ser obtida pela simples concatenação de soluções ótimas para submetas individuais, em qualquer ordem. Solucionar uma única submeta independente pode não ser trivial, mas a complexidade de problemas com submetas independentes cresce apenas linearmente, em função do número de submetas no problema.
- *serializável*, se existe uma permutação das submetas na qual elas podem ser resolvidas, seqüencialmente, sem que nenhuma submeta já atingida nessa ordem seja violada. Se as submetas são resolvidas na ordem correta, a complexidade de problemas com submetas serializáveis é linear com relação ao número de submetas. Caso contrário, submetas serializáveis podem levar a uma complexidade exponencial, já que resolvê-las numa ordem errada pode requerer que uma mesma submeta seja estabelecida e violada um número exponencial de vezes.
- *não-serializável*, se submetas previamente estabelecidas devem ser necessariamente violadas a fim de que algum progresso seja feito em direção à meta principal, qualquer que seja a ordem considerada. Problemas com submetas não-serializáveis têm complexidade exponencial.

Percebendo que a classe de problemas serializáveis englobava uma variedade de problemas de complexidades muito diferentes, *Barret & Weld* [3] propuseram que ela fosse refinada em duas outras subclasses. Assim, segundo eles, um conjunto de submetas serializável pode ser classificado como:

- *trivialmente serializável*, se as submetas podem ser tratadas em qualquer ordem, sem que nunca uma submeta previamente estabelecida seja violada[1].
- *laboriosamente serializável*, se existe uma ordem serializável, mas pelo menos uma fração das permutações de submetas não podem ser resolvidas seqüencialmente, sem possivelmente violar uma submeta já estabelecida.

Se um problema é trivialmente serializável, a combinação das soluções para suas submetas não causa qualquer tipo de dificuldade adicional, que não seja encontrada apenas examinando-se cada submeta individualmente. Por outro lado, se um problema é laboriosamente serializável, combinar as soluções para as submetas individuais causa novas dificuldades, a despeito do fato dessas submetas serem ou não triviais.

Essa classificação de problemas será útil no capítulo 5 – *Resultados experimentais*, onde uma série de experimentos realizados com domínios de diferentes complexidades serão apresentados e analisados.

3.2 Planejamento como busca no espaço de estados

Como vimos na seção 1.2, o espaço de estados de um mundo pode ser modelado por um grafo cujos nós representam os possíveis estados desse mundo e cujas arestas representam as ações que transformam um estado em outro. Nesse caso, a tarefa de planejamento consiste na busca de uma seqüência de ações que definem, nesse grafo, um caminho que leva do estado inicial até um estado meta. Claramente, uma tal seqüência de ações será uma solução para o problema de planejamento considerado. Por exemplo, a figura 3.1 representa o espaço de estados para o nosso exemplo do mundo dos blocos. Observe que, nessa figura, o caminho destacado representa uma solução minimal para o problema conhecido como *Anomalia de Sussman* [45].

[1] Note que essa definição é menos restritiva que aquela para problemas independentes. Particularmente, em problemas trivialmente serializáveis, nem sempre a concatenação de soluções ótimas para submetas individuais será uma solução ótima global.

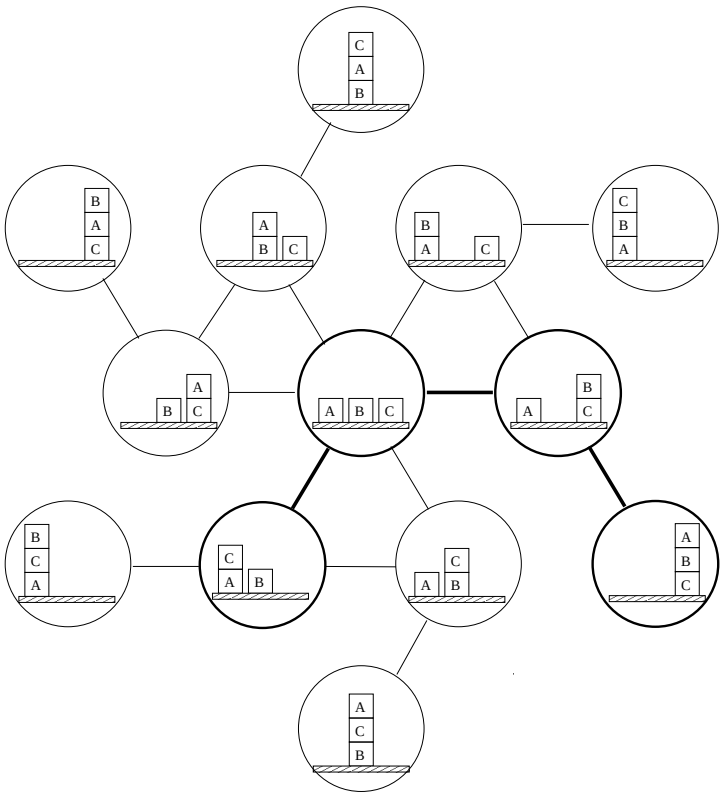


Figura 3.1: *Espaço de estados para o mundo dos blocos.*

Uma vantagem em modelar a tarefa de planejamento como busca é que isso nos possibilita aplicar diversos algoritmos de *força bruta* ou *heurísticos* bem conhecidos na literatura de Inteligência Artificial [32, 33, 52].

Basicamente, a busca num grafo representando um espaço de estados pode ser feita de duas maneiras: *progressiva*, *i.e.* a partir do nó que representa o estado inicial, tentamos encontrar um nó representando um estado meta; ou *regressiva*, *i.e.* a partir de um nó que representa um estado meta, tentamos encontrar o nó que representa o estado inicial.

3.2.1 Planejamento progressivo

O algoritmo não-determinístico PROG, apresentado na tabela 3.1, realiza uma busca progressiva num espaço de estados. Inicialmente, ele é chamado com o estado corrente

Σ igual ao estado inicial $\mathcal{I}$ do problema e com o plano parcial[2] π sendo uma seqüência de ações vazia. No início de cada iteração, esse algoritmo (1) verifica se a meta é satisfeita no estado corrente e, caso seja, devolve o plano π como solução do problema. Senão, (2) ele escolhe uma ação aplicável ao estado corrente, anexa-a[3] ao final do plano em construção e continua a busca no estado resultante de sua aplicação. Caso uma tal escolha não seja possível, (3) o algoritmo devolve FALHA.

Algoritmo PROG$(\mathcal{A}, \Sigma, \mathcal{G}, \pi)$

Entrada: *A descrição das ações* $\mathcal{A}$.
A descrição do estado corrente Σ.
A descrição da meta de planejamento $\mathcal{G}$.
Um plano parcialmente especificado π.

Saída: FALHA *ou um plano completo* π.

1. Se $\mathcal{G} \subseteq \Sigma$ então devolva π.
2. Escolha $\alpha_i \in \{\alpha \mid \alpha \in \mathcal{A} \wedge pre(\alpha) \subseteq \Sigma\}$.
2.1. Seja $\pi' :=$PROG$(\mathcal{A}, \Sigma - del(\alpha_i) + add(\alpha_i), \mathcal{G}, \pi \circ \alpha_i)$.
2.2. Se $\pi' \neq$ FALHA então devolva π'.
2.3. Retroceda.
3. Devolva FALHA.

Tabela 3.1: *Busca progressiva no espaço de estados.*

3.2.2 Planejamento regressivo

A idéia básica do planejamento regressivo consiste na *regressão* de um estado, através de uma ação. Dados um estado Σ e uma ação α, o estado *precedente* obtido pela regressão de Σ através de α é representado pelo conjunto $\Sigma + pre(\alpha) + del(\alpha) - add(\alpha)$. Para garantir a consistência da regressão, entretanto, apenas ações relevantes devem ser consideradas. Dizemos que uma ação α é *relevante* a um estado Σ se $add(\alpha) \cap \Sigma \neq \emptyset$ e $del(\alpha) \cap \Sigma = \emptyset$, ou seja, se seus efeitos positivos suportam pelo menos um átomo e seus efeitos negativos não ameaçam nenhum átomo desse estado [60, 44]. Por exemplo,

[2]Um plano parcial é uma seqüência de ações que leva do estado inicial a um estado intermediário (progressivo) ou, então, que leva de um estado intermediário a um estado meta (regressivo).

[3]O operador $\circ$ será empregado para denotar a operação de concatenação.

considere o estado $\Sigma_1 := \{clear(b), clear(c)\}$ e seja $move(a,b,c)$ a ação que move o bloco a, de cima de b, para cima de c. A condição $add(move(a,b,c)) \cap \Sigma_1 = \{clear(b)\} \neq \emptyset$ é satisfeita, mas $del(move(a,b,c)) \cap \Sigma_1 = \{clear(c)\} = \emptyset$, não. Portanto, a ação $move(a,b,c)$ não é relevante para o estado Σ_1 e, sendo assim, não deve ser usada para a sua regressão. Note que, qualquer que fosse o estado precedente Σ_0, obtido pela regressão de Σ_1 através da ação $move(a,b,c)$, o estado resultante da aplicação de $move(a,b,c)$ a Σ_0 teria que satisfazer Σ_1, no qual o bloco c está livre. Entretanto, isso não é possível, já que essa ação move o bloco a para cima do bloco c.

Algoritmo REGR$(\mathcal{A}, \mathcal{I}, \Sigma, \pi)$

Entrada: *A descrição das ações* $\mathcal{A}$.
A descrição do estado inicial $\mathcal{I}$.
A descrição do estado corrente Σ.
Um plano parcialmente especificado π.

Saída: FALHA *ou um plano completo* π.

1. Se $\mathcal{I} \subseteq \Sigma$ então devolva π.
2. Escolha $\alpha_i \in \{\alpha \mid \alpha \in \mathcal{A} \wedge add(\alpha) \cap \Sigma \neq \emptyset \wedge del(\alpha) \cap \Sigma = \emptyset\}$.
2.1. Seja $\pi' :=$REGR$(\mathcal{A}, \mathcal{I}, \Sigma + pre(\alpha_i) + del(\alpha_i) - add(\alpha_i), \alpha_i \circ \pi)$.
2.2. Se $\pi' \neq$ FALHA então devolva π'.
2.3. Retroceda.
3. Devolva FALHA.

Tabela 3.2: *Busca regressiva no espaço de estados.*

O algoritmo não-determinístico REGR, apresentado na tabela 3.2, realiza uma busca regressiva num espaço de estados. Inicialmente, esse algoritmo é chamado com o estado corrente Σ igual a um estado meta $\mathcal{G}$ e com π sendo uma seqüência de ações vazia. No início de cada iteração, o algoritmo (1) verifica se as propriedades do estado inicial são satisfeitas no estado corrente e, caso sejam, devolve o plano π como solução do problema de planejamento. Senão, (2) o algoritmo escolhe uma ação α_i que seja relevante ao estado corrente. Se tal escolha é possível, a ação escolhida é prefixada ao plano parcial e a busca continua no estado precedente, $\Sigma + pre(\alpha_i) + del(\alpha_i) - add(\alpha_i)$, obtido pela regressão de Σ através de α_i. Caso contrário, (3) o algoritmo devolve FALHA.

3.2.3 Comparação entre planejamento progressivo e regressivo

Na prática, o não-determinismo proporcionado pela primitiva *escolha*, usada nos algoritmos PROG e REGR, deve ser implementado através de busca e, desta forma, o fator de ramificação[4] da árvore gerada/explorada durante a busca tem um papel fundamental na eficiência desses algoritmos. De fato, se o fator de ramificação é r, a complexidade de PROG e REGR é $O(r^n)$, onde n é o número de ações existentes no plano [53]. Assim, para planos com o mesmo número de ações, será mais eficiente aquele planejador que explorar a árvore com o menor fator de ramificação. Se admitirmos a suposição de que apenas uma pequena parte dos átomos usados na descrição de um domínio de planejamento são empregados na descrição de uma particular meta de planejamento, então a árvore de busca regressiva deve apresentar um fator de ramificação bem inferior àquele da árvore de busca progressiva e, conseqüentemente, o planejamento regressivo deve ser muito mais eficiente que o progressivo.

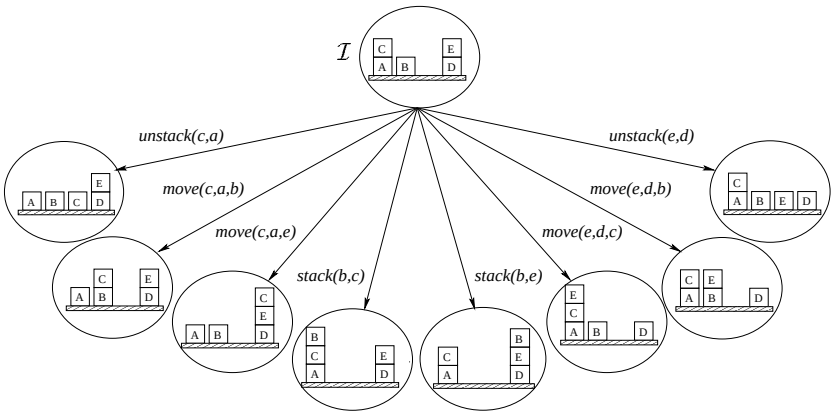


Figura 3.2: *Ramificação na árvore de busca progressiva.*

Como exemplo, considere um problema de planejamento cujo estado inicial é $\mathcal{I} := \{clear(b), clear(c), clear(e), ontable(a), ontable(b), ontable(d), on(c,a), on(e,d)\}$ e cuja meta é $\mathcal{G} := \{on(c,b)\}$. As figuras 3.2 e 3.3 ilustram, respectivamente, o primeiro nível das árvores de busca progressiva e regressiva para esse problema. Observe que, como $\mathcal{I}$ é uma descrição completa do estado inicial, a progressão a partir desse estado também gera estados completamente definidos e, portanto, cada nó na árvore de busca

[4] O número médio de filhos por nó.

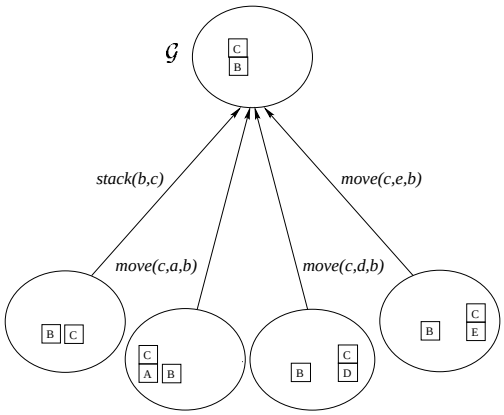


Figura 3.3: *Ramificação na árvore de busca regressiva.*

progressiva corresponde diretamente a um único estado do mundo. Além disso, ao expandir um nó, o planejamento progressivo considera todas as ações cujas precondições são satisfeitas no estado representado por esse nó. Então, quanto mais átomos tivermos na descrição de um estado, mais ações aplicáveis a ele teremos. Muitas ações serão aplicáveis, mas apenas algumas delas poderão contribuir para que a meta seja atingida. Por outro lado, como o conjunto $\mathcal{G}$ descreve um estado meta apenas parcialmente, a regressão a partir desse estado também deve gerar estados apenas parcialmente definidos e, portanto, cada nó na árvore de busca regressiva representa um conjunto de estados. Ao contrário do que ocorre no planejamento progressivo, no planejamento regressivo a busca é orientada pela descrição da meta de planejamento e, por esse motivo, pode ter um desempenho muito melhor.

3.3 Planejamento como busca no espaço de planos

Em vez de busca no espaço de estados, podemos realizar busca no espaço de planos [54]. O *espaço de planos* pode ser modelado por um grafo cujos nós representam planos parcialmente especificados e cujas arestas denotam operações de refinamento, tais como adição de novos passos ou adição de novas restrições de ordem temporal entre passos já existentes no plano [28]. Começamos com um plano simples e, então, consideramos meios de transformá-lo num plano completo que solucione o problema de planejamento. Nessa abordagem, a solução não é representada por um caminho no grafo, mas sim por

um nó que representa um tal plano completo. Essa idéia está ilustrada na figura 3.4, que mostra o refinamento de uma solução para o problema da *Anomalia de Sussman.*

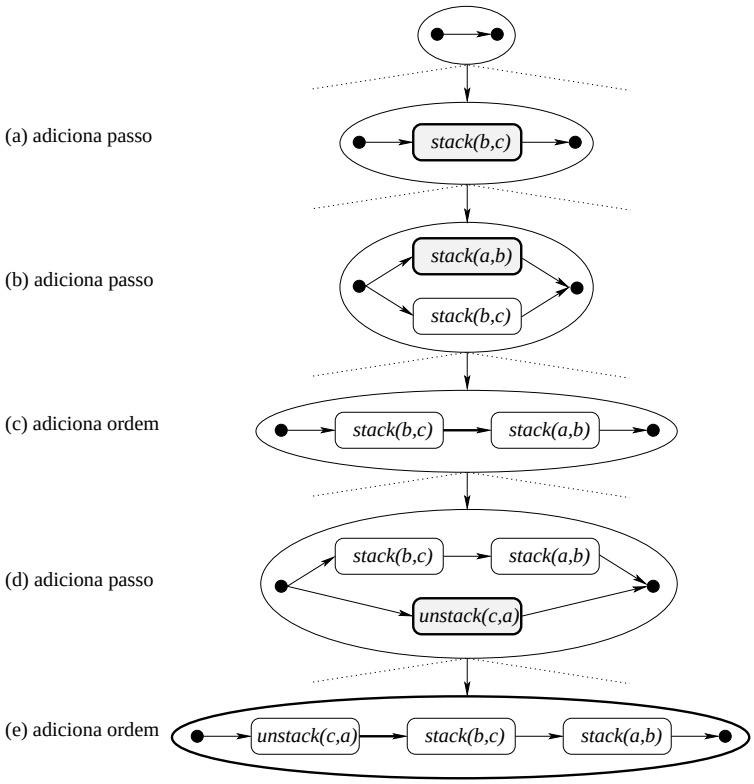


Figura 3.4: *Fragmento do espaço de planos para a Anomalia de Sussman.*

A maior motivação para essa mudança de paradigma é que, como a busca no espaço de planos não compromete a ordem de planejamento com a ordem de execução do plano construído, nesse novo paradigma, podemos evitar o retrocesso na escolha de submetas. Na busca no espaço de estados, esse retrocesso é necessário porque é possível que as submetas sejam tratadas numa ordem diferente daquela em que terão que ser atingidas durante a execução do plano obtido e, para garantir a completude da busca, é preciso que toda permutação possível das submetas seja considerada. Já no caso da busca no espaço de planos, como não há compromisso entre a ordem de planejamento e a ordem de execução, as submetas podem ser tratadas em qualquer ordem (observe, na figura 3.4, que a ordem em que as ações são incluídas no plano é diferente daquela em que

elas serão executadas). Assim, o planejamento como busca no espaço de planos exige que o agente planejador raciocine explicitamente sobre a construção de um plano, bem como sobre as possíveis interações que podem existir entre as ações incluídas nele.

3.3.1 Planejamento de ordem total

Quando um plano é definido como sendo uma seqüência totalmente ordenada de ações, e a inserção de uma nova ação é restrita ao início dessa seqüência, o planejamento de ordem total como busca no espaço de planos é isomorfo ao planejamento regressivo como busca no espaço de estados. Como a cada chamada recursiva o algoritmo REGR recebe como argumento uma seqüência de ações totalmente ordenada, é como se ele estivesse fazendo busca no espaço de planos e a operação de refinamento fosse simplesmente prefixar uma nova ação no início dessa seqüência. Assim, a grande vantagem em considerar busca no espaço de planos está na possibilidade de considerarmos outras operações de refinamento do plano (*e.g.* inserir ações em posições arbitrárias), que possam tornar o processo de planejamento mais eficiente.

3.3.2 Planejamento de ordem parcial

No planejamento de ordem parcial, planos são representados por triplas da forma $\langle \mathcal{S}, \mathcal{O}, \mathcal{L} \rangle$, onde $\mathcal{S}$ é um conjunto de *passos*, $\mathcal{O}$ é um conjunto de restrições de *ordem temporal* e $\mathcal{L}$ é um conjunto de *vínculos causais*. Cada passo do plano é uma instância distinta de um dos operadores do domínio, as restrições de ordem temporal impõem uma ordem parcial sobre esses passos[5] e os vínculos causais estabelecem o propósito de cada um deles no plano.

Um *vínculo causal* é uma estrutura da forma $\alpha_p \rightarrow \phi @ \alpha_c$, onde α_p é uma *ação produtora* cujo efeito ϕ é uma precondição para a *ação consumidora* α_c [59, 39]. Por definição, se $\alpha_p \rightarrow \phi @ \alpha_c \in \mathcal{L}$ então $\alpha_p \prec \alpha_c \in \mathcal{O}$. Uma estrutura da forma $\phi @ \alpha_c$ é denominada *pré-requisito* e, em $\alpha_p \rightarrow \phi @ \alpha_c$, dizemos que α_p *suporta* o pré-requisito $\phi @ \alpha_c$.

[5] Formalmente, um passo é uma ação rotulada e, assim, uma mesma ação pode ocorrer diversas vezes num mesmo plano. Por exemplo, na seqüência $\langle 3 : unstack(c, a), 1 : stack(c, a), 2 : unstack(c, a) \rangle$, o primeiro e o último passos são distintos, embora sejam ambos instâncias de um mesmo operador. De fato, as restrições de ordem devem ser definidas sobre os rótulos das ações e não sobre as ações propriamente. Porém, por uma questão de clareza, isso não será feito explicitamente.

Vínculos causais são usados para proteger submetas já satisfeitas, possibilitando que o planejador possa determinar quando a adição de um novo passo, ou a reutilização de um passo já existente no plano, interfere com (ameaça) decisões tomadas anteriormente.

Definição 3.3 *Sejam* $\langle \mathcal{S}, \mathcal{O}, \mathcal{L} \rangle$ *um plano parcialmente ordenado,* $\alpha_t \in \mathcal{S}$ *um passo e* $\lambda := \alpha_p \rightarrow \phi @ \alpha_c \in \mathcal{L}$ *um vínculo causal. Dizemos que o passo* α_t *ameaça o vínculo causal* λ *se e só se* $\mathcal{O} \cup \{\alpha_p \prec \alpha_t \prec \alpha_c\}$ *é consistente e* α_t *produz* $\neg\phi$ *como efeito.* □

Para evitar ameaças, o algoritmo de planejamento deve proteger os vínculos causais, adicionando novas restrições de ordem ao plano. Se um passo α_t ameaça um vínculo causal $\alpha_p \rightarrow \phi @ \alpha_c$, então esse passo deve ser *antecipado*, *i.e.* executado antes de α_p, ou então *postergado*, *i.e.* executado depois de α_c.

Definição 3.4 *Um plano parcialmente ordenado* $\langle \mathcal{S}, \mathcal{O}, \mathcal{L} \rangle$ *é completo se para todo passo* $\alpha_c \in \mathcal{S}$ *e toda precondição* $\phi \in pre(\alpha_c)$*, existe um vínculo causal* $\alpha_p \rightarrow \phi @ \alpha_c \in \mathcal{L}$ *tal que* $\alpha_p \in \mathcal{S}$*,* $\alpha_p \prec \alpha_c \in \mathcal{O}$ *e nenhum outro passo do plano ameaça esse vínculo.* □

Dado um plano completo $\langle \mathcal{S}, \mathcal{O}, \mathcal{L} \rangle$ para um determinado problema de planejamento, qualquer ordenação topológica dos passos em $\mathcal{S}$ que seja consistente com as restrições de ordem temporal em $\mathcal{O}$ é uma *solução* para esse problema.

Planos vazios

Para permitir que tanto planos parcialmente especificados quanto planos completos sejam representados da mesma maneira, introduzimos a idéia de *plano vazio*. O plano vazio tem dois passos virtuais, $\mathcal{S} = \{a_0, a_\infty\}$, uma restrição de ordem temporal, $\mathcal{O} = \{a_0 \prec a_\infty\}$ e nenhum vínculo causal, $\mathcal{L} = \{\}$. Para um determinado problema de planejamento $\langle \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, o passo a_0 não tem precondições e seus efeitos especificam que propriedades valem no estado inicial $\mathcal{I}$; por outro lado, o passo a_∞ não tem efeitos e suas precondições determinam que propriedades devem valer no estado meta $\mathcal{G}$. Por definição, o passo a_0 é o primeiro, a_∞ é o último, e todos os demais passos do plano devem ser precedidos por a_0 e seguidos por a_∞. Por exemplo, um diagrama[6] representando o plano vazio para a *Anomalia de Sussman* pode ser visto na figura 3.5.

[6] Nesse tipo de diagrama, que usaremos para representar planos parcialmente ordenados, adotaremos as seguintes convenções: (1) caixas representam passos do plano, (2) precondições e efeitos de um passo são posicionados, respectivamente, acima e abaixo da sua caixa correspondente, (3) linhas pontilhadas denotam restrições de ordem temporal, e (4) linhas sólidas denotam vínculos causais.

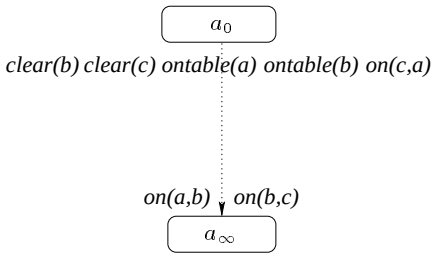


Figura 3.5: *O plano vazio para o problema da Anomalia de Sussman.*

O algoritmo POP

O POP (**P***artial* **O***rder* **P***lanner*) é um algoritmo de planejamento regressivo de ordem parcial que realiza busca no espaço de planos [8, 39, 61]. Nesse algoritmo, apresentado na tabela 3.3, a tarefa de planejamento consiste basicamente em escolher ações e adicioná-las ao plano, até que toda submeta seja atingida e toda ameaça, decorrente das interações entre essas ações, seja devidamente evitada.

Algoritmo POP$(\mathcal{A}, \Gamma, \langle \mathcal{S}, \mathcal{O}, \mathcal{L} \rangle)$

Entrada: *A descrição das ações* $\mathcal{A}$*;*
Um conjunto de pré-requisitos ainda não satisfeitos Γ*;*
Um plano de ordem parcial, parcialmente especificado, $\langle \mathcal{S}, \mathcal{O}, \mathcal{L} \rangle$.

Saída: FALHA *ou um plano de ordem parcial completo* $\langle \mathcal{S}, \mathcal{O}, \mathcal{L} \rangle$.

1. Se $\Gamma = \emptyset$, devolva $\langle \mathcal{S}, \mathcal{O}, \mathcal{L} \rangle$.
2. Remova um pré-requisito $\phi@\alpha_c \in \Gamma$.
3. Escolha um passo α_p (em $\mathcal{S}$ ou $\mathcal{A}$), possivelmente precedendo α_c, com efeito ϕ.
3.1. Adicione $a_0 \prec \alpha_p \prec \alpha_c$ a $\mathcal{O}$.
3.2. Se $\alpha_p \notin \mathcal{S}$, então adicione α_p a $\mathcal{S}$ e $\phi_i@\alpha_c$ a Γ, para cada $\phi_i \in pre(\alpha_p)$.
3.3. Adicione o vínculo causal $\alpha_p \rightarrow \phi@\alpha_c$ a $\mathcal{L}$.
4. Para cada $\alpha_t \in \mathcal{S}$ que ameaça um vínculo $\alpha_i \rightarrow \phi@\alpha_j \in \mathcal{L}$, escolha:
4.1. Adicione $\alpha_t \prec \alpha_i$ a $\mathcal{O}$ ou
4.2. Adicione $\alpha_j \prec \alpha_t$ a $\mathcal{O}$.
5. Sejam Γ', $\mathcal{S}'$, $\mathcal{O}'$ e $\mathcal{L}'$ os conjuntos de entrada devidamente alterados.
5.1. Se as alterações são consistentes, devolva POP$(\mathcal{A}, \Gamma', \langle \mathcal{S}', \mathcal{O}', \mathcal{L}' \rangle)$.
5.2. Senão, se não há como retroceder em alguma escolha feita, devolva FALHA.

Tabela 3.3: POP – *planejamento de ordem parcial.*

Inicialmente, POP recebe um conjunto Γ contendo todos, e somente, os pré-requisitos gerados a partir das precondições do passo a_∞, ou seja, $\Gamma = \{\phi@a_\infty \mid \phi \in pre(a_\infty)\}$. Em cada iteração, (1) o POP verifica se o conjunto Γ está vazio e, caso esteja, devolve o plano $\langle \mathcal{S}, \mathcal{O}, \mathcal{L} \rangle$ como solução. Caso contrário, (2) ele seleciona um dos pré-requisitos em Γ para satisfazer. Note que essa seleção não deve causar retrocesso, já que a ordem em que as submetas são tratadas pelo algoritmo é irrelevante para a ordem de execução: *se houver um plano que resolva o problema, esse plano será encontrado independentemente da ordem em que as submetas forem selecionadas para serem atingidas*. Uma vez selecionado um pré-requisito, (3) uma ação que o suporte é escolhida não-deterministicamente. Essa ação pode ser uma já existente no plano, *i.e.* um passo em $\mathcal{S}$, ou então uma nova instância de uma das ações descritas em $\mathcal{A}$. Então, (3.1, 3.2 e 3.3) o plano é alterado para comportar essa ação escolhida, o conjunto Γ é atualizado com os pré-requisitos correspondentes a esse novo passo, (4, 4.1 e 4.2) as possíveis ameaças resultantes dessa alteração são resolvidas, e (5.1) o processo continua normalmente com o plano alterado. Caso, em algum ponto do processo, nenhuma escolha apropriada possa ser feita, (5.2) o algoritmo devolve FALHA.

Um exemplo de planejamento de ordem parcial

Vamos ilustrar o funcionamento do algoritmo POP resolvendo o problema da *Anomalia de Sussman*. Inicialmente, o algoritmo é chamado com o plano vazio ilustrado na figura 3.5 e com $\Gamma = \{on(a,b)@a_\infty, on(b,c)@a_\infty\}$. Como o conjunto de pré-requisitos Γ não está vazio, o algoritmo precisa selecionar um de seus elementos para ser satisfeito. Suponha que o pré-requisito $on(a,b)@a_\infty$ seja selecionado e que a ação $stack(a,b)$ seja escolhida (não-deterministicamente) para suportá-lo. Então, o plano parcial é devidamente modificado para comportar essa ação, conforme ilustrado na figura 3.6, e passamos a ter

$\Gamma' = \{on(b,c)@a_\infty, clear(a)@stack(a,b), clear(b)@stack(a,b), ontable(a)@stack(a,b)\}$,
$\mathcal{S}' = \{a_0, stack(a,b), a_\infty\}$,
$\mathcal{O}' = \{a_0 \prec a_\infty, a_0 \prec stack(a,b), stack(a,b) \prec a_\infty\}$ e
$\mathcal{L}' = \{stack(a,b) \rightarrow on(a,b)@a_\infty\}$,

que serão fornecidos como entrada para a próxima iteração do algoritmo.

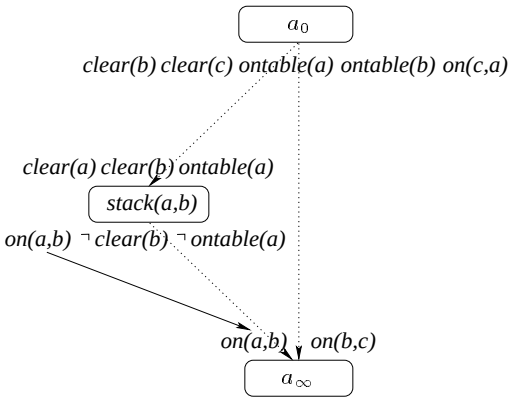


Figura 3.6: *Estabelecendo o vínculo causal* $stack(a,b) \rightarrow on(a,b)@a_\infty$.

Na próxima iteração, como Γ ainda não estará vazio, um outro pré-requisito deverá ser satisfeito. Como já dissemos, a ordem em que os pré-requisitos são selecionados para serem satisfeitos não afeta a completude do algoritmo e, portanto, podemos selecioná-los em qualquer ordem desejada. Então, suponha que $on(b,c)@a_\infty$ seja o segundo pré-requisito selecionado e que a ação $stack(b,c)$ seja escolhida para suportá-lo. Nesse caso, o plano será modificado conforme ilustrado na figura 3.7.

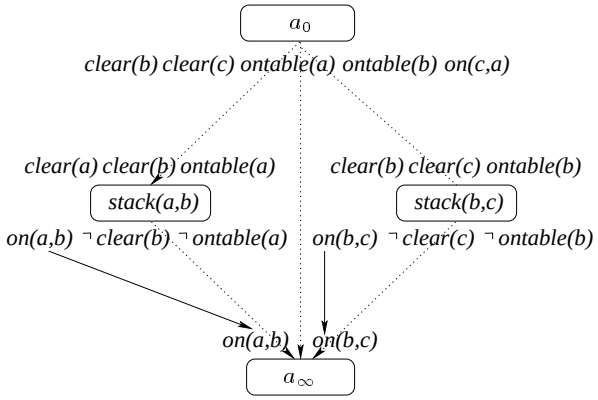


Figura 3.7: *Estabelecendo o vínculo causal* $stack(b,c) \rightarrow on(b,c)@a_\infty$.

Para ver como uma ameça pode ser evitada, suponha que na terceira iteração $clear(b)@stack(b,c)$ seja o pré-requisito selecionado e a_0 seja o passo escolhido para

suportá-lo. Estabelecemos esse fato adicionando o vínculo $a_0 \rightarrow clear(b)@stack(b,c)$ ao plano. Porém, como os passos $stack(a,b)$ e $stack(b,c)$ não estão ordenados entre si, nada impede que $stack(a,b)$ seja executado antes de $stack(b,c)$ e, desta forma, elimine a precondição $clear(b)$ necessária para esse último. De fato, o passo $stack(a,b)$ ameça o vínculo recém estabelecido e, para evitar essa ameça, devemos antecipá-lo ou postergá-lo. Como a restrição $stack(a,b) \prec a_0$ seria inconsistente com $\mathcal{O}$, a única possibilidade é postergar o passo $stack(a,b)$, fazendo com que ele seja executado depois do passo $stack(b,c)$. A figura 3.8 mostra o plano resultante dessas modificações. Observe que, nesse ponto, todos os pré-requisitos ainda não selecionados, exceto $clear(a)@stack(a,b)$, podem ser suportados diretamente pelo passo a_0.

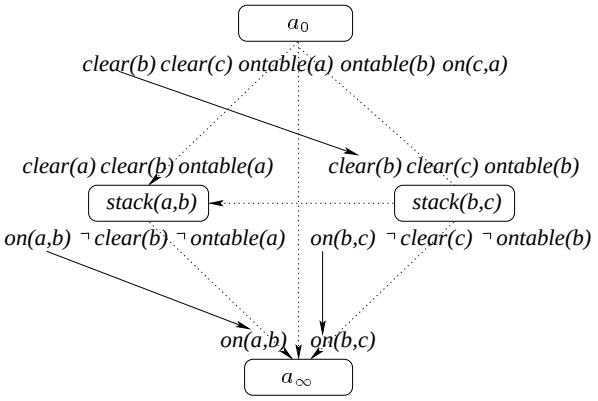


Figura 3.8: *Eliminando a ameaça* $stack(a,b) \times a_0 \rightarrow clear(b)@stack(b,c)$.

Finalmente, supondo que o pré-requisito $clear(a)@stack(a,b)$ seja o último selecionado e que a ação $unstack(c,a)$ seja escolhida para suportá-lo, obtemos o plano final ilustrado na figura 3.9.

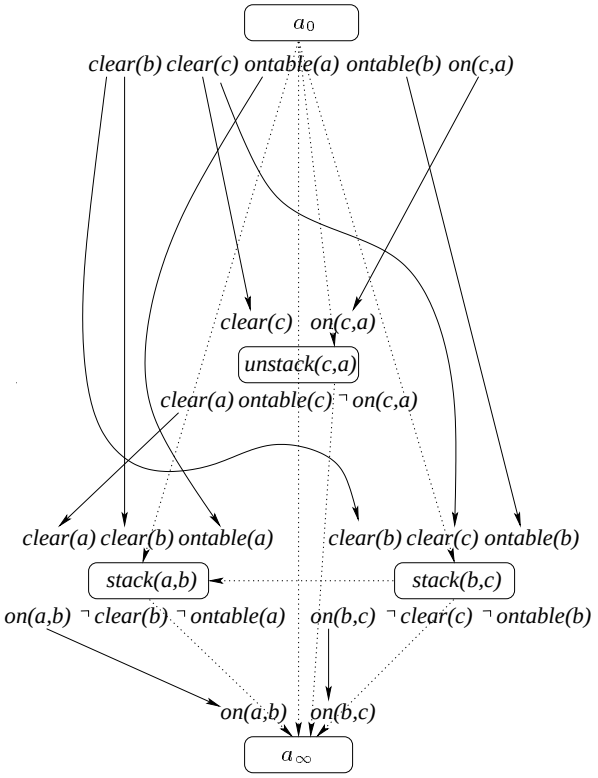


Figura 3.9: *Um plano parcialmente ordenado completo para a Anomalia de Sussman.*

3.3.3 Comparação entre planejamento de ordem total e parcial

O tempo gasto por um algoritmo de busca é $O(r^n)$, onde n representa o número de escolhas não-determinísticas que são feitas antes que uma solução seja encontrada e r representa o fator de ramificação, *i.e.* o número de alternativas existentes, em média, em cada ponto de escolha [53]. Desses parâmetros, o mais significativo é o fator de ramificação e, para POP, ele é bem menor que para REGR (que como vimos na subseção 3.3.1, realiza planejamento de ordem total como busca no espaço de planos). Isso acontece porque o algoritmo POP não retrocede na seleção de submetas (veja a linha 2 na tabela 3.3) e, ao expandir um nó, considera apenas as ações que sejam capazes de suportar a submeta escolhida (veja a linha 3 na tabela 3.3). Diferentemente, o algoritmo REGR, além de retroceder na escolha de submetas, considera todas as ações que satisfaçam qualquer uma das precondições de qualquer submeta ainda não atingida (veja a linha 2 na tabela 3.2). Sendo assim, o planejamento de ordem parcial é, geralmente, muito mais eficiente que o planejamento de ordem total.

3.4 Políticas de proteção

Além do POP, dois outros algoritmos de planejamento de ordem parcial bastante conhecidos são o SNLP [39] e o TWEAK[7] [8]. Esses algoritmos diferem entre si, essencialmente, com relação à *política de proteção* adotada em cada um deles: enquanto o POP protege as submetas apenas de ameaças negativas, o SNLP as protege de ameaças negativas e positivas e o TWEAK não as protege de nada. Como veremos, essas diferentes políticas de proteção de submetas implicam em diferentes comportamentos de busca (sistemático ou redundante) que, dependendo das características do domínio de aplicação considerado, podem melhorar ou piorar muito a eficiência do planejador.

3.4.1 Planejamento sistemático

Uma seqüência de ações $\langle \alpha_1, \ldots, \alpha_n \rangle$ é uma *solução potencial* com relação a um plano $\langle \mathcal{S}, \mathcal{O}, \mathcal{L} \rangle$, se ela é consistente com as restrições impostas nesse plano; ou seja, se ela

[7] Na verdade, tanto o POP quanto o SNLP são resultantes de modificações que foram feitas no algoritmo TWEAK, na tentativa melhorar a sua eficiência.

contém todos os passos existentes em $\mathcal{S}$ e satisfaz todas as restrições de ordem temporal existentes em $\mathcal{O}$ [25]. Então, como para um mesmo plano parcialmente ordenado podem existir diversas soluções potenciais distintas, cada nó no espaço de planos representa, de fato, um conjunto de soluções potenciais e a busca consiste na adição de restrições, e no conseqüente refinamento desses conjuntos [28], até que uma solução seja encontrada.

Como nenhuma restrição, de nenhum tipo, é removida durante o refinamento de um plano, segue que os nós sucessores obtidos contêm todas as restrições existentes no seu nó pai. Conseqüentemente, uma solução consistente com as restrições impostas num nó filho também é, necessariamente, consistente com as restrições impostas no seu nó pai. Sendo assim, o conjunto de soluções potenciais representado por qualquer nó na árvore de busca é um subconjunto daquele conjunto representado pelo seu nó pai.

Definição 3.5 *Um planejador é considerado sistemático se, a cada passo de refinamento realizado por ele, os conjuntos de soluções potenciais representados pelos nós sucessores obtidos são disjuntos.* □

Note que, para garantir a sistematicidade de um planejador, basta garantir que cada passo de refinamento realizado por ele seja sistemático [25]. Conforme *MacAllester & Rosemblitt* [39] perceberam, a simples idéia de *ameaça positiva* é suficiente para garantir a sistematicidade dos passos de refinamento no planejamento de ordem parcial.

Definição 3.6 *Sejam $\langle \mathcal{S}, \mathcal{O}, \mathcal{L} \rangle$ um plano, $\alpha_t \in \mathcal{S}$ um passo e $\lambda := \alpha_p \rightarrow \phi@\alpha_c \in \mathcal{L}$ um vínculo causal. O passo α_t ameaça o vínculo causal λ se e só se $\mathcal{O} \cup \{\alpha_p \prec \alpha_t \prec \alpha_c\}$ é consistente e α_t produz ϕ (ameaça positiva) ou $\neg\phi$ (ameaça negativa) como efeito.* □

Por exemplo, considere a árvore de busca representada na figura 3.10. Inicialmente, para suportar os pré-requisitos $g_1@a_\infty$, $g_2@a_\infty$ e $g_3@a_\infty$, foram adicionados, respectivamente, os passos a_1, a_2 e a_3. Com isso, o único pré-requisito que ainda resta a ser satisfeito é $p@a_3$. Então, supondo que ameaças positivas não sejam consideradas, dois refinamentos são possíveis: um que adiciona a restrição $a_1 \rightarrow p@a_3$ e outro que adiciona a restrição $a_2 \rightarrow p@a_3$. Claramente, esses refinamentos geram nós sucessores que compartilham algumas soluções potenciais iguais e, portanto, não são sistemáticos.

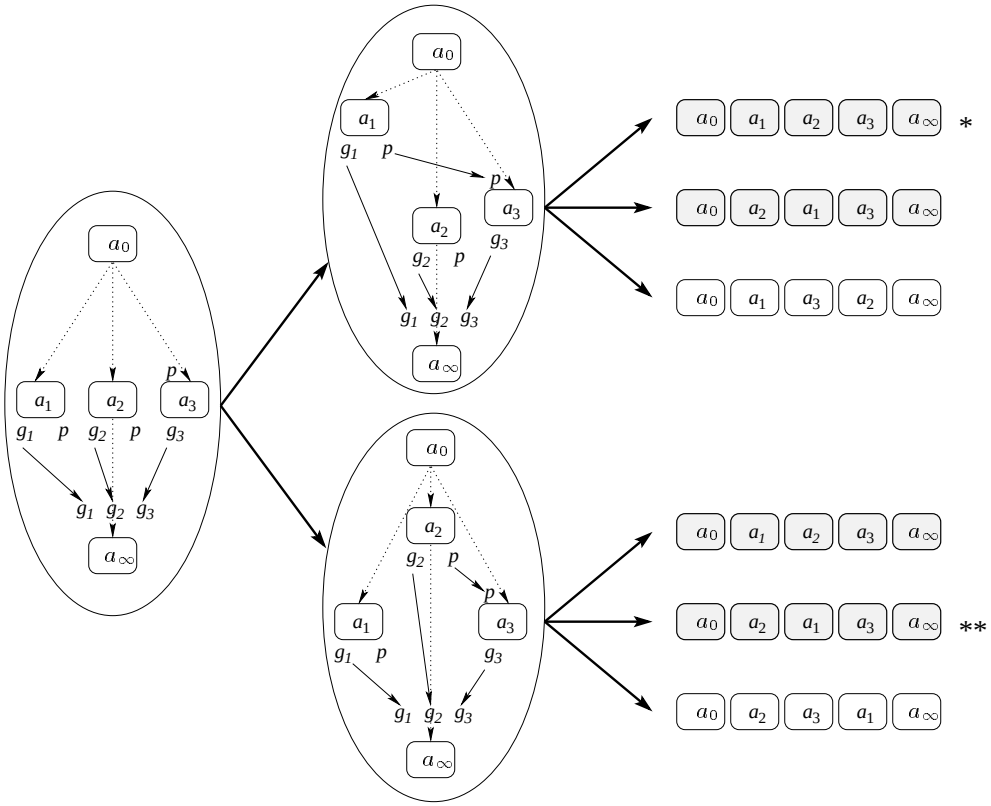


Figura 3.10: *Redundância no espaço de busca.*

Por outro lado, se ameaças positivas fossem consideradas, quando o vínculo causal $a_1 \rightarrow p@a_3$ fosse estabelecido, o passo a_2 se tornaria uma ameaça positiva para ele. Isso faria com que o planejador adicionasse uma restrição de ordem temporal extra, forçando o passo a_2 a ser executado antes de a_1 ou depois de a_3. Conseqüentemente, a seqüência de ações $\langle a_1, a_2, a_3 \rangle$ (marcada na figura 3.10 com um asterisco) não seria mais uma solução potencial para o nó obtido pelo primeiro refinamento. Analogamente, no segundo refinamento, o passo a_1 seria uma ameaça positiva para o vínculo $a_2 \rightarrow p@a_3$ e o planejador teria que antecipar a_1 com relação a a_2 ou, então, postergar a_1 com relação a a_3. Nesse caso, a seqüência de ações $\langle a_2, a_1, a_3 \rangle$ (marcada com dois asteriscos) não seria mais uma solução potencial para o nó obtido pelo segundo refinamento. Logo, esses refinamentos gerariam nós sucessores cujos conjuntos de soluções potenciais seriam disjuntos, garantindo assim a sistematicidade da busca. Uma prova formal desse resultado pode ser encontrada em [25].

O algoritmo SNLP

O SNLP (**S***ystematic* **N***on-***L***inear* **P***lanner*), desenvolvido por *MacAllester & Rosemblitt* [39], é essencialmente igual ao POP, sendo a única diferença o fato dele ser sistemático. Para garantir essa sistematicidade, toda vez que um vínculo $\alpha_p \rightarrow \phi@\alpha_c$ é estabelecido, o SNLP adiciona também o vínculo complementar $\alpha_p \rightarrow \neg\phi@\alpha_c$. Dessa forma, apenas um único suporte para cada pré-requisito do plano é permitido; o que garante que os conjuntos de soluções potenciais em cada ramo da árvore de busca sejam disjuntos e que, portanto, a busca seja sistemática.

Algoritmo SNLP$(\mathcal{A}, \Gamma, \langle \mathcal{S}, \mathcal{O}, \mathcal{L} \rangle)$

Entrada: *A descrição das ações* $\mathcal{A}$*;*
Um conjunto de pré-requisitos ainda não satisfeitos Γ*;*
Um plano parcialmente especificado e parcialmente ordenado $\langle \mathcal{S}, \mathcal{O}, \mathcal{L} \rangle$.

Saída: FALHA *ou um plano completo parcialmente ordenado* $\langle \mathcal{S}, \mathcal{O}, \mathcal{L} \rangle$.

1. Se $\Gamma = \emptyset$, devolva $\langle \mathcal{S}, \mathcal{O}, \mathcal{L} \rangle$.
2. Remova um pré-requisito $\phi@\alpha_c \in \Gamma$.
3. Escolha um passo α_p (em $\mathcal{A}$ ou $\mathcal{S}$), possivelmente precedendo α_c, com efeito ϕ.
3.1. Adicione $a_0 \prec \alpha_p \prec \alpha_c$ a $\mathcal{O}$.
3.2. Se $\alpha_p \notin \mathcal{S}$, então adicione α_p a $\mathcal{S}$ e $\phi_i@\alpha_c$ a Γ, para cada $\phi_i \in pre(\alpha_p)$.
3.3. Adicione os vínculos $\alpha_p \rightarrow \phi@\alpha_c$ e $\alpha_p \rightarrow \neg\phi@\alpha_c$ a $\mathcal{L}$.
4. Para cada $\alpha_t \in \mathcal{S}$ que ameaça um vínculo $\alpha_i \rightarrow \phi@\alpha_j \in \mathcal{L}$, escolha:
4.1. Adicione $\alpha_t \prec \alpha_i$ a $\mathcal{O}$ ou
4.2. Adicione $\alpha_j \prec \alpha_t$ a $\mathcal{O}$.
5. Sejam Γ', $\mathcal{S}'$, $\mathcal{O}'$ e $\mathcal{L}'$ os conjuntos de entrada devidamente alterados.
5.1. Se as alterações são consistentes, devolva SNLP$(\mathcal{A}, \Gamma', \langle \mathcal{S}', \mathcal{O}', \mathcal{L}' \rangle)$.
5.2. Senão, se não há como retroceder em alguma escolha feita, devolva FALHA.

Tabela 3.4: SNLP – *planejamento de ordem parcial sistemático.*

3.4.2 Planejamento redundante

A finalidade dos vínculos causais no planejamento de ordem parcial é garantir que submetas já satisfeitas permaneçam necessariamente satisfeitas, em todos os futuros refinamentos de um determinado plano parcial. Para tanto, o planejador deve proteger

esses vínculos causais, detectando e evitando as possíveis ameaças que surgem durante os refinamentos. No planejamento redundante, entretanto, não há a preocupação de evitar que uma submeta já satisfeita seja violada e, sendo assim, os vínculos causais não precisam mais ser mantidos como parte do plano parcial.

Sem os vínculos causais, o planejador não tem como saber quando um plano parcialmente ordenado está completo; a menos que ele verifique a validade de cada uma das precondições, de cada um dos passos do plano. Isso pode ser feito através do MTC (**M***odal* **T***ruth* **C***riterion*), conforme enunciado por *Chapman* [8]:

> CRITÉRIO DE VERDADE MODAL: Uma proposição p é necessariamente verdadeira numa situação s se e só se duas condições são satisfeitas: há uma situação t, igual ou necessariamente precedendo s, na qual p é necessariamente satisfeita; e para todo passo C, possivelmente antes de s, e toda proposição q, possivelmente codesignando com p, que C nega, há um passo W, necessariamente entre C e s, que adiciona uma proposição r, tal que r e p codesignam-se sempre que p e q codesignam-se. [*Chapman*'87, p. 340]

O MTC estabelece condições necessárias e suficientes para garantir que uma proposição seja verdadeira num dado instante do tempo, dado um plano parcialmente ordenado. De acordo com esse critério, uma submeta ϕ é necessariamente satisfeita no estado final atingido pela execução de um determinado plano parcial $\pi := \langle \mathcal{S}, \mathcal{O} \rangle$ se, para toda ordenação total π' de π, ϕ é satisfeita no estado final atingido pela execução de π'. Conforme *Chapman* provou, um plano parcialmente ordenado é completo e correto se e só se satisfaz o critério de verdade modal, esquematizado[8] na figura 3.11.

Definição 3.7 *Um plano parcialmente ordenado $\langle \mathcal{S}, \mathcal{O} \rangle$ é completo e correto se e só se, para todo passo $\alpha_c \in \mathcal{S}$ e para cada $\phi \in pre(\alpha_c)$, são satisfeitas duas condições:*

(i) existe um passo contribuidor $\alpha_p \in \mathcal{S}$, tal que $\alpha_p \prec \alpha_c \in \mathcal{O}$ e $\phi \in add(\alpha_p)$;
(ii) para cada passo $\alpha_t \in \mathcal{S}$, se $\phi \in del(\alpha_t)$, então $\alpha_t \prec \alpha_p \in \mathcal{O}$ ou $\alpha_c \prec \alpha_t \in \mathcal{O}$. □

[8] Na verdade, o critério originalmente estabelecido por *Chapman* é um pouco mais complexo, já que ele considerou a possibilidade de ações contendo variáveis e codesignação (unificação) como sendo uma das possíveis operações de refinamento do plano parcial. No nosso caso, porém, como estamos usando o STRIPS proposicional, esse critério pode ser simplificado [49].

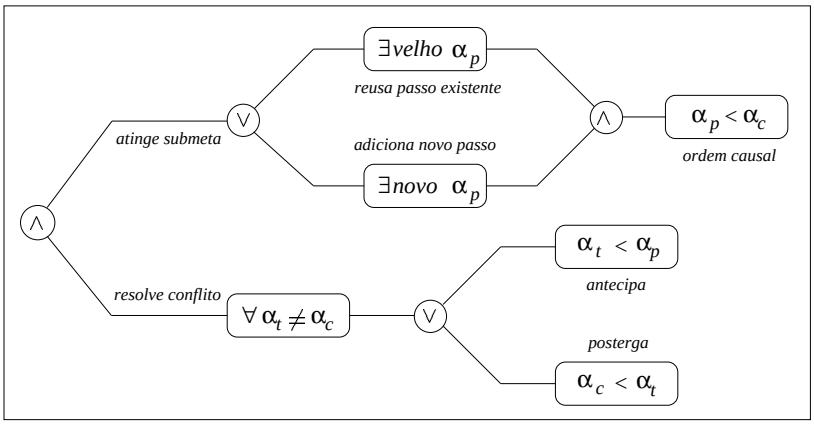


Figura 3.11: *Critério de verdade modal simplificado.*

Note que o MTC tem um papel duplo, uma vez que ele pode ser usado tanto para validação quanto para geração de planos [49]. Para validarmos um plano usando o MTC, basta verificarmos se ele é satisfeito para cada uma das precondições, de cada um dos passos existentes no plano. Se isso acontece, claramente, o plano está correto. Do contrário, se há alguma precondição ϕ, para algum passo α_c, para a qual o MTC não é satisfeito, dois casos podem ocorrer:

- Não existe um passo α_p tal que $\alpha_p \in \mathcal{S}$, $\phi \in add(\alpha_p)$ e $\alpha_p \prec \alpha_c \in \mathcal{O}$. Então, para corrigir o plano, há duas possibilidades: podemos reutilizar um passo α_p já existente no plano (se for consistente adicionar a restrição $\alpha_p \prec \alpha_c$ a $\mathcal{O}$) ou, então, podemos criar um novo passo α_p (adicionando α_p a $\mathcal{S}$ e $\alpha_p \prec \alpha_c$ a $\mathcal{O}$).

- Existe um passo α_p tal que $\alpha_p \in \mathcal{S}$, $\phi \in add(\alpha_p)$ e $\alpha_p \prec \alpha_c \in \mathcal{O}$; mas há também um passo $\alpha_t \in \mathcal{S}$ ameaçando o vínculo causal $\alpha_p \rightarrow \phi@\alpha_c$. Então, para corrigir o plano, podemos adicionar uma restrição de ordem temporal a $\mathcal{O}$ ($\alpha_t \prec \alpha_p$ ou $\alpha_c \prec \alpha_t$), de modo que $\mathcal{O}$ permaneça consistente e a ameaça seja evitada.

O algoritmo TWEAK

O TWEAK é um planejador de ordem parcial redundante, que foi desenvolvido por *Chapman* [8] a partir de uma interpretação procedimental do MTC. Ao contrário do POP e do SNLP, o TWEAK não guarda vínculos causais para garantir que submetas já estabelecidas não sejam violadas. Em decorrência disso, no TWEAK, tanto a verificação

do término do planejamento (linha 1 na tabela 3.5), quanto a seleção de submetas (linha 2 na tabela 3.5) devem ser feitas através do uso do MTC.

Algoritmo TWEAK$(\mathcal{A}, \Gamma, \langle \mathcal{S}, \mathcal{O} \rangle)$

Entrada: *A descrição das ações* $\mathcal{A}$.
Um conjunto de pré-requisitos ainda não satisfeitos Γ.
Um plano parcialmente ordenado, parcialmente especificado, $\langle \mathcal{S}, \mathcal{O} \rangle$.

Saída: FALHA *ou um plano parcialmente ordenado, completo,* $\langle \mathcal{S}, \mathcal{O} \rangle$.

1. Se todo pré-requisito em Γ está necessariamente satisfeito, devolva $\langle \mathcal{S}, \mathcal{O} \rangle$.
2. Selecione um pré-requisito $\phi@\alpha_c \in \Gamma$ não necessariamente satisfeito.
3. Escolha um passo α_p (em $\mathcal{A}$ ou $\mathcal{S}$), possivelmente precedendo α_c, com efeito ϕ.
3.1. Adicione $a_0 \prec \alpha_p \prec \alpha_c$ a $\mathcal{O}$.
3.2. Se $\alpha_p \notin \mathcal{S}$, então adicione α_p a $\mathcal{S}$ e $\phi_i@\alpha_c$ a Γ, para cada $\phi_i \in pre(\alpha_p)$.
4. Para cada $\alpha_t \in \mathcal{S}$ que ameaça o vínculo $\alpha_p \rightarrow \phi@\alpha_c$, escolha:
4.1. Adicione $\alpha_t \prec \alpha_p$ a $\mathcal{O}$ ou
4.2. Adicione $\alpha_c \prec \alpha_t$ a $\mathcal{O}$.
5. Sejam Γ', $\mathcal{S}'$ e $\mathcal{O}'$ os conjuntos de entrada devidamente alterados.
5.1. Se as alterações são consistentes, devolva TWEAK$(\mathcal{A}, \Gamma', \langle \mathcal{S}', \mathcal{O}' \rangle)$.
5.2. Senão, se não há como retroceder em alguma escolha feita, devolva FALHA.

Tabela 3.5: TWEAK – *planejamento de ordem parcial redundante.*

3.4.3 Refinamento dos parâmetros de complexidade

Na prática, algoritmos não-determinísticos devem ser implementados como busca, que pode ser feita de duas formas básicas: em *profundidade* ou em *largura*. Dessas duas, apenas a busca em largura é completa, *i.e.* garante que uma solução seja encontrada sempre que ela existir. Nesse tipo de busca, a árvore de busca é expandida, nível por nível, até que a solução seja encontrada e, portanto, sua complexidade é $O(r^p)$, sendo r o fator de *ramificação* médio da árvore e p a *profundidade* da solução.

Como os planejadores POP, SNLP e TWEAK são todos não-determinísticos, e desejamos que sejam completos, temos que implementá-los como busca em largura[9]. Nesse caso, todos eles terão complexidade $O(r^p)$ e, portanto, para uma comparação mais pre-

[9]Ou, equivalentemente, como busca em profundidade iterativa [53].

cisa do comportamento desses planejadores será necessária uma análise mais detalhada dos fatores que influenciam os parâmetros r e p em cada um deles.

Análise do fator de ramificação

Examinando os algoritmos apresentados nas tabelas 3.3, 3.4 e 3.5, podemos observar que, a cada iteração, eles realizam duas tarefas principais: *satisfazer um pré-requisito* e *resolver os conflitos resultantes*. Por exemplo, a figura 3.12 ilustra uma iteração[10] onde um nó representando um plano parcial é expandido em nós sucessores representando os possíveis refinamentos desse plano parcial.

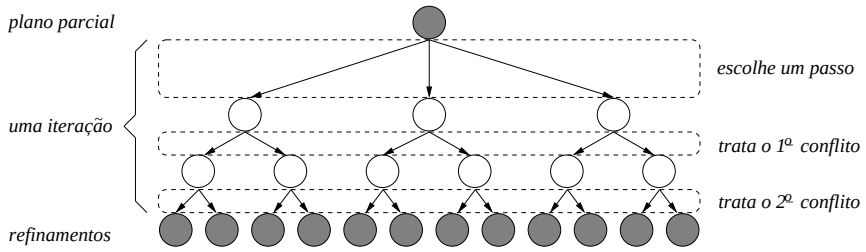


Figura 3.12: *Uma iteração num algoritmo de planejamento de ordem parcial.*

Para satisfazer um pré-requisito, um algoritmo pode reutilizar um passo já existente ou, então, inserir um novo passo no plano. Sejam r_A o número de passos existentes no plano que podem ser *reutilizados* por um algoritmo A e a_A o número de passos novos que podem ser *adicionados* ao plano, por esse mesmo algoritmo. Então, temos que cada pré-requisito pode ser satisfeito por $r_A + a_A$ passos distintos.

Para cada passo que é escolhido para satisfazer um determinado pré-requisito, um novo vínculo causal é estabelecido. Então, podem surgir dois tipos de conflitos: o novo vínculo causal é ameaçado por passos existentes no plano ou o novo passo ameaça vínculos causais já estabelecidos anteriormente. Seja c_A o número médio de conflitos resolvidos por vínculo, por um determinado algoritmo A. Como cada conflito pode ser resolvido por *antecipação* ou *postergação*, cada alternativa de escolha de passo para satisfazer um pré-requisito resulta, em média, em 2^{c_A} planos distintos (veja a figura

[10] Nessa iteração, estamos supondo que existem três passos (novos ou já existentes) para suportar o pré-requisito selecionado e que, para cada um deles, são tratados exatamente dois conflitos.

3.12). Assim, o número médio de planos gerados a partir do refinamento de um único plano, *i.e.* o fator de ramificação médio da árvore de busca, será $r = 2^{c_A} \times (r_A + a_A)$.

Note que a ramificação da árvore de busca cresce exponencialmente, em função do número médio de conflitos resolvidos. Então, quanto mais rígida for a estratégia de proteção adotada pelo planejador, maior será o crescimento do número de planos a serem examinados a cada nova iteração.

Análise da profundidade da solução

A cada iteração, um pré-requisito do plano é selecionado para ser satisfeito pelos algoritmos. Como a satisfação de um pré-requisito implica num refinamento do plano e, portanto, na geração de planos no próximo nível na árvore de busca, segue que a profundidade da solução corresponde diretamente ao número de pré-requisitos selecionados durante a busca.

Seja μ o número médio de precondições para os operadores de um domínio particular (incluindo o operador virtual a_∞ para o problema considerado). Então, nesse domínio, um plano com n passos terá cerca de $\mu \times n$ pré-requisitos. Seja p_A a porcentagem desses pré-requisitos que é selecionada por um determinado algoritmo A. Se esse algoritmo usa vínculos causais para estabelecer a validade dos pré-requisitos, necessariamente, ele terá que estabelecer um vínculo causal para cada um dos pré-requisitos do plano e, portanto, terá que selecionar 100% dos $\mu \times n$ pré-requisitos existentes. Por outro lado, se o MTC é utilizado, é possível que apenas uma parte dos pré-requisitos existentes no plano seja selecionada pelo algoritmo. Isso pode ocorrer porque, ao satisfazer um pré-requisito selecionado, o passo escolhido pelo algoritmo pode acabar satisfazendo também um outro pré-requisito do plano que ainda não foi selecionado. Então, como o MTC somente seleciona pré-requisitos que não estejam necessariamente satisfeitos, esse tal pré-requisito satisfeito "por acaso" não será mais selecionado durante a busca (exceto se for violado posteriormente).

Seja S o conjunto de pré-requisitos que são selecionados durante a busca ($|S| = p_A \times \mu \times n$). Se o método é redundante, *i.e.* se não há um estratégia de proteção de submetas, alguns dos pré-requisitos em S poderão ser selecionados mais de uma vez. Seja f_A a freqüência média em que cada pré-requisito em S é selecionado por um

determinado algoritmo A, até que a solução seja encontrada. Então, a profundidade da solução será $p = f_A \times p_A \times \mu \times n$.

Complexidade "refinada" do planejamento de ordem parcial

A partir das análises do fator de ramificação e da profundidade da solução na árvore de busca, chegamos à conclusão que a complexidade esperada dos planejadores[11] de ordem parcial apresentados é $O((2^{c_A} \times (r_A + a_A))^{f_A \times p_A \times \mu \times n})$. A tabela 3.6 apresenta um resumo descritivo dos fatores envolvidos nessa complexidade. Agora, com informações numa granularidade menor, podemos fazer uma análise comparativa mais precisa do comportamento de busca dos planejadores de ordem parcial.

fator	*descrição*
c_A	*média de conflitos, por pré-requisito selecionado pelo algoritmo A*
r_A	*passos reutilizados, por pré-requisito selecionado pelo algoritmo A*
a_A	*passos adicionados, por pré-requisito selecionado pelo algoritmo A*
f_A	*freqüência média em que os pré-requisitos são selecionados pelo algoritmo A*
p_A	*porcentagem dos pré-requisitos do plano selecionada pelo algoritmo A*
μ	*média de precondições por operador do domínio considerado*
n	*número de passos num plano-solução minimal*

Tabela 3.6: *Parâmetros de complexidade dos planejadores.*

3.4.4 Comparação entre planejamento sistemático e redundante

A análise comparativa que se segue pressupõe que todos os planejadores (POP, SNLP e TWEAK) estejam resolvendo o mesmo problema, dentro de um mesmo domínio.

Seleção de pré-requisitos

Como vimos, a altura da árvore de busca aumenta linearmente, em função do número de pré-requisitos selecionados para serem satisfeitos durante o planejamento. Ademais, o tempo gasto pelo planejador aumenta exponencialmente com relação à altura dessa

[11] Supondo que os algoritmos não-determinísticos sejam implementados como busca em largura.

árvore. Sendo assim, a maneira como os planejadores selecionam os pré-requisitos em cada iteração é, certamente, o fator que mais influencia na sua complexidade de tempo.

Para cada pré-requisito selecionado e satisfeito durante a busca, POP e SNLP guardam um vínculo causal que garante que esse pré-requisito permanecerá sempre *necessariamente satisfeito*. (Isso permite que os pré-requisitos selecionados por esses algoritmos sejam removidos do plano.) Ademais, nesses planejadores, um plano parcial é considerado completo somente quando todo pré-requisito nele existente tem um vínculo causal correspondente. Desta forma, POP e SNLP só terminam sua execução quando 100% dos pré-requisitos do plano foram selecionados e, portanto, devemos ter $p_{pop} = p_{snlp} = 1$. Além disso, como os pré-requisitos selecionados são removidos do plano, cada um deles só pode ser selecionado uma única vez e, portanto, devemos ter $f_{pop} = f_{snlp} = 1$. Então, como a profundidade da árvore de busca é dada por $p = f_A \times p_A \times \mu \times n$, segue que, para os planejadores POP e SNLP, ela será sempre $p = \mu \times n$, não importando as características específicas do domínio considerado.

Por outro lado, como o TWEAK não guarda vínculos causais, ele pode finalizar sua execução assim que um refinamento resultar num plano que satisfaça o MTC. Se, no domínio considerado, for possível que um mesmo passo suporte vários pré-requisitos de uma só vez, então pode acontecer de alguns pré-requisitos nunca serem selecionados durante a busca realizada pelo TWEAK. Daí segue que $p_{tweak} \leq 1$. Além disso, como o TWEAK não protege vínculos causais previamente estabelecidos (em iterações passadas), um pré-requisito já satisfeito pode ser violado e ter que ser selecionado novamente numa iteração futura. Nesse caso, então, teremos $f_{tweak} \geq 1$. Assim, no caso do TWEAK, a profundidade da árvore dependerá do domínio: em domínios com muita *redundância positiva*, *i.e.* com muitas ações adicionando uma mesma submeta, espera-se que a profundidade da solução seja menor para TWEAK do que para POP e SNLP; já em domínios com muita redundância negativa, *i.e.* com muitas ações removendo uma mesma submeta, espera-se que a profundidade da solução seja maior para TWEAK do que para os outros dois planejadores.

Estabelecimento de pré-requisitos

Como vimos, o fator de ramificação da árvore de busca aumenta linearmente com relação ao número de passos que podem ser usados para satisfazer um determinado pré-requisito selecionado. Então, quanto mais alternativas de suporte para um determinado pré-requisito forem consideradas por um planejador, mais tempo esse planejador levará para encontrar uma solução.

O número de passos novos a_A, que podem ser adicionados por um algoritmo A para satisfazer um pré-requisito selecionado, depende essencialmente do domínio. Quanto mais operadores satisfazendo um mesmo pré-requisito existirem no domínio, maior será o número de passos novos que poderão ser adicionados pelo algoritmo. Então, se estamos considerando o mesmo problema, no mesmo domínio, o parâmetro a_A deveria ser o mesmo para os três planejadores. Entretanto, se o planejador não resolve, na mesma iteração, todos os conflitos negativos resultantes da adição de um novo passo ao plano, como é o caso do TWEAK, então passos inconsistentes com o plano podem ser considerados e a inconsistência só será percebida mais tarde, em uma futura iteração. Mas, até que a inconsistência seja detectada, a árvore já se ramificou mais do que devia. Sendo assim, devemos ter $a_{snlp} = a_{pop} < a_{tweak}$.

Por outro lado, o número de passos do plano r_A, que podem ser reutilizados por um algoritmo A para satisfazer um pré-requisito selecionado, depende essencialmente do algoritmo A considerado. Em geral, quanto mais restrições de ordem um planejador inserir no plano, menor será a chance dele poder reutilizar um passo já existente. (Para entender o porquê, observe que se um plano possui n passos que contribuem com uma certa precondição ϕ, apenas aqueles passos que não estão ordenados após o passo α_j podem ser reutilizados para satisfazer o pré-requisito $\phi@\alpha_j$.) Restrições de ordem são inseridas num plano, basicamente, por dois motivos: primeiro, para garantir a consistência da relação $causa \rightarrow efeito$, quando um passo novo ou existente é usado para satisfazer um pré-requisito selecionado; segundo, para resolver conflitos de interação entre passos e manter o plano consistente. O número de restrições do primeiro tipo corresponde ao número de pré-requisitos distintos selecionados pelo algoritmo e, conforme já analisamos, esse número para o TWEAK nunca é maior do que aquele para os outros dois planejadores. As restrições de ordem do segundo tipo, entretanto,

dependem do número de conflitos tratados, a cada iteração, pelo planejador. Como TWEAK é o planejador que trata menos conflitos por iteração, enquanto o SNLP é o que trata mais, devemos ter $r_{snlp} < r_{pop} < r_{tweak}$.

Note, entretanto, que se os passos que podem ser reutilizados pelo TWEAK para satisfazer um pré-requisito $\phi@\alpha_j$ já estiverem ordenados com relação ao passo α_j, como a seleção é feita pelo MTC, o pré-requisito $\phi@\alpha_j$ será considerado necessariamente verdadeiro e jamais será selecionado. Nesse caso, então, o número de passos reutilizados pelo TWEAK pode ser muito menor do que aquele para os demais planejadores.

Proteção de pré-requisitos

A média de conflitos tratados, por pré-requisito selecionado em cada iteração, é também um fator importante a ser considerado na comparação dos planejadores:

- O SNLP trata tanto os conflitos entre o vínculo causal recém estabelecido e os passos já existentes no plano, quanto aqueles entre um novo passo adicionado e os vínculos estabelecidos anteriormente. Ademais, o SNLP considera não só os conflitos resultantes de ameaças negativas, mas também aqueles resultantes de ameaças positivas.

- O POP trata tanto os conflitos entre o vínculo causal recém estabelecido e os passos já existentes no plano, quanto aqueles entre o novo passo adicionado e os vínculos estabelecidos anteriormente. A diferença é que, ao contrário do SNLP, o POP considera apenas os conflitos resultantes de ameaças negativas.

- O TWEAK, ao contrário dos dois planejadores anteriores, trata apenas os conflitos ocasionados por ameaças negativas entre o vínculo causal recém estabelecido e os passos já existentes no plano. Se, por acaso, um novo passo adicionado ameaçar vínculos anteriormente estabelecidos, tais conflitos somente serão resolvidos nas próximas iterações do algoritmo.

Claramente, o TWEAK é o planejador que trata menos conflitos por pré-requisito selecionado, enquanto o SNLP é o que trata mais. Daí segue que $c_{tweak} < c_{pop} < c_{snlp}$.

Comportamento esperado para os planejadores

As considerações feitas nas subseções anteriores sugerem que o comportamento dos planejadores de ordem parcial depende, essencialmente, das características do domínio de planejamento considerado. Com base nelas, formulamos as duas hipóteses apresentadas a seguir, que serão comprovadas empiricamente através de um experimento que será descrito no capítulo 5 – *Resultados experimentais*.

Hipótese 1. *O domínio tem muito mais ameaças positivas que negativas.* Nesse caso, como o SNLP é o único que trata ameaças positivas e o TWEAK trata apenas parte das ameaças negativas, o número de conflitos tratados pelo SNLP será muito maior do que o número de conflitos tratados pelos outros dois planejadores ($c_{tweak} < c_{pop} \ll c_{snlp}$). Particularmente, se não houver ameaças negativas, apenas o SNLP tratará conflitos. Em compensação, como o SNLP é o planejador que adiciona mais restrições de ordem temporal, ele terá planos mais lineares e, conseqüentemente, o menor número de passos reutilizáveis ($r_{snlp} < r_{pop} < r_{tweak}$). Finalmente, como há muita redundância positiva no domínio, um único passo poderá contribuir com vários pré-requisitos ao mesmo tempo ($p_{tweak} \ll p_{pop} = p_{snlp}$) e, pelo mesmo motivo, o número de pré-requisitos já satisfeitos violados pelo TWEAK será muito baixo ($f_{tweak} \leq f_{pop} = f_{snlp}$). Como a complexidade desses planejadores é $O((2^{c_A} \times (r_A + r_n))^{f_A \times p_A \times \mu \times n})$ e os fatores dominantes c_A, f_A e p_A são mínimos para o TWEAK, segue que esse planejador terá o melhor desempenho nesse domínio. Por outro lado, como o expoente c_A é muito maior para o SNLP do que para o POP, segue que o SNLP terá o pior desempenho nesse domínio.

Hipótese 2. *O domínio tem muito mais ameaças negativas que positivas.* Diferentemente do caso anterior, como o número de ameaças positivas é baixo, o número de conflitos para SNLP não será tão maior que aquele para o POP ($c_{tweak} < c_{pop} < c_{snlp}$). Ademais, como há poucos operadores que contribuem com a mesma precondição, os três planejadores deverão ter aproximadamente o mesmo número de passos que poderão ser reutilizados ($r_{snlp} \leq r_{pop} \leq r_{tweak}$). Finalmente, como há muita redundância negativa no domínio, o TWEAK terá que selecionar quase todos os pré-requisitos ($p_{tweak} \leq p_{pop} = p_{snlp}$) e, provavelmente, satisfazer várias vezes um mesmo pré-requisito ($f_{pop} = f_{snlp} \ll f_{tweak}$). Agora, como os fatores c_A e r_A são muito

próximos para os três planejadores e o fator f_A é máximo para o TWEAK, segue que esse planejador deverá ter o pior desempenho nesse domínio. Entre POP e SNLP, temos que a profundidade da solução, o número de conflitos tratados e o número de passos reutilizáveis será aproximadamente o mesmo. Daí segue que POP e SNLP deverão ter aproximadamente o mesmo desempenho. Entretanto, com um plano mais linear (com mais restrições de ordem) as ameças podem ser verificadas mais rapidamente e, portanto, o desempenho do SNLP poderá ser um pouco melhor do que o do POP.

3.5 Considerações finais

Nesse capítulo, apresentamos os principais conceitos da área de planejamento clássico em Inteligência Artificial. Particularmente, vimos que o ganho de eficiência computacional nos planejadores algorítmicos, proporcionado pelo uso da representação STRIPS, é obtido ao custo de uma grande perda de expressividade na linguagem de representação de conhecimento (em comparação àquela usada no planejamento dedutivo). Além disso, vimos também que o planejamento regressivo no espaço de planos é, em geral, muito mais eficiente que planejamento progressivo no espaço de estados e que a política de proteção adotada pelo planejador (aliada às características do domínio de planejamento considerado) pode ter um grande impacto no seu comportamento de busca (sistemático ou redundante) e, conseqüentemente, na eficiência do seu desempenho.

Conforme vimos no capítulo anterior, no cálculo de situações, um plano $\langle \alpha_1, \ldots, \alpha_n \rangle$ é representado pelo termo $do(\alpha_n, do(\ldots, do(\alpha_1, s_0)))$, que denota a situação atingida pela execução desse plano, a partir da situação inicial s_0. A linearidade imposta pela função *do*, e o fato de que cada nova extensão do plano corresponde a um novo estado completamente definido pelas ações já consideradas, faz com que os planejadores que empregam o cálculo de situações como formalismo para representação de ações se restrinjam ao paradigma de planejamento de ordem total como busca no espaço de estados. Por outro lado, como veremos no próximo capítulo, usando cálculo de eventos abdutivo (em vez do cálculo de situações dedutivo), o planejamento de ordem parcial como busca no espaço de planos surge naturalmente. Desta forma, esperamos que com esse novo formalismo – *o cálculo de eventos* – possamos recuperar toda a expressividade da linguagem do cálculo de situações e, ao mesmo tempo, contar com a eficiência dos

planejadores algorítmicos discutidos nesse capítulo.

Capítulo 4

Planejamento abdutivo

A sabedoria consiste na
antecipação das conseqüências.

Norman Cousins

Escritor e Editor
(1915–1990)

4.1 Abdução

A ABDUÇÃO, originalmente introduzida por *Peirce* [47], é uma forma de raciocínio em que uma determinada hipótese é adotada como uma possível explicação para um fato observado, de acordo com leis conhecidas: *se observamos o fato* β *e sabemos que* $\alpha \rightarrow \beta$, *então* α *pode ser adotada como uma possível explicação para o fato* β. Evidentemente, a abdução é uma regra de inferência fraca, pois não pode garantir que uma explicação *abduzida* seja verdadeira, mas apenas que seja *plausível*.

Definição 4.1 *Sejam* Δ *um conjunto de sentenças descrevendo um domínio e* Γ *uma sentença descrevendo uma observação nesse domínio. A abdução consiste em encontrar um conjunto de sentenças* Π, *denominado explicação abdutiva de* Γ, *tal que*
(*a*) $\Delta \cup \Pi \models \Gamma$, *e*
(*b*) $\Delta \cup \Pi$ *é consistente.* □

Sejam $\Delta := \{verão \rightarrow chuva, umidade \wedge calor \rightarrow chuva, chuva \rightarrow enchente\}$ um domínio de interesse e $\Gamma := \{enchente\}$ uma observação nesse domínio. Então, por exemplo, entre as possíveis explicações abdutivas para Γ, temos as seguintes:

$\Pi_1 := \{chuva\}$,
$\Pi_2 := \{umidade, calor, verão\}$,
$\Pi_3 := \{umidade, calor\}$,
$\Pi_4 := \{verão\}$,
$\Pi_5 := \{umidade, calor, vento\}$, e
$\Pi_6 := \{enchente\}$.

Para evitar explicações arbitrárias (*e.g.* *vento* em Π_5) e triviais (*e.g.* Π_6), geralmente, os fatos que compõem uma explicação abdutiva são restringidos a pertencer a um conjunto básico de *causas primitivas* pré-estabelecidas, denominadas *abdutíveis*. Entretanto, mesmo com tal restrição, ainda é possível que para um mesmo fato observado tenhamos diversas explicações abdutivas distintas. De fato, a existência de múltiplas explicações plausíveis é uma característica básica do raciocínio abdutivo, sendo a seleção de uma explicação *preferencial* um importante problema a ser considerado [26].

Na prática, é muito difícil estabelecer critérios gerais, independentes do domínio de aplicação, através dos quais possamos identificar explicações preferenciais. Entretanto, no caso de planejamento, os critérios sugeridos por *Cox & Pietrzykowski* [12] são bastante apropriados. Segundo eles, uma explicação abdutiva Π, para uma observação Γ num determinado domínio Δ, deve ser:

- *básica*, *i.e.* não deve ser dada em termos de efeitos, mas sim de causas primitivas;
- *minimal*, *i.e.* não deve existir um conjunto $\Pi^* \subset \Pi$ tal que $\Delta \cup \Pi^* \models \Gamma$; e
- *compacta*, *i.e.* deve postular o menor número de causas possível.

Assim, retomando o último exemplo, $\Pi_1 := \{chuva\}$ não é uma explicação básica para $\Gamma := \{enchente\}$; $\Pi_2 := \{umidade, calor, verão\}$ é uma explicação básica mas não é minimal; $\Pi_3 := \{umidade, calor\}$ é básica e minimal, mas não é compacta; e, finalmente, $\Pi_4 := \{verão\}$ é básica, minimal e compacta.

Outro aspecto importante a ser ressaltado é que, por definição, a abdução é um tipo de *raciocínio não-monotônico* [26]. Assim, explicações consistentes com um determinado estado de conhecimento podem se tornar inconsistentes, quando novas in-

formações são consideradas. Por exemplo, se a fórmula $\neg verão$ for adicionada ao conjunto $\Delta := \{verão \rightarrow chuva, umidade \wedge calor \rightarrow chuva, chuva \rightarrow enchente\}$, então $\Pi_4 := \{verão\}$ deixará de ser uma explicação abdutiva plausível para $\Gamma := \{enchente\}$.

4.1.1 O mecanismo abdutivo em programação lógica

Programação em lógica pode ser facilmente estendida com abdução [19, 12, 55, 26, 14]. Conforme vimos na subseção 2.4.2, dados um programa lógico Δ e uma cláusula objetivo Γ_0, uma SLD-*refutação* para Γ_0 a partir de Δ é uma seqüência finita $\Gamma_0, \ldots, \Gamma_n$, onde Γ_n é a cláusula vazia e cada Γ_{i+1} é uma resolvente de Γ_i com uma das cláusulas em Δ. Então, se para algum Γ_i o literal selecionado λ não resolve com nenhuma das cláusulas em Δ, as seqüências iniciando com $\Gamma_0, \ldots, \Gamma_i$ não levarão à cláusula vazia e, portanto, não nos permitirão concluir a refutação. Entretanto, se o nosso interesse não é apenas refutar Γ_0, mas sim encontrar um conjunto de fatos Π tal que $\Delta \cup \Pi \models \Gamma_0$, então, incluindo em Π uma cláusula unitária que resolva com λ, podemos continuar a refutação com Γ_{i+1} igual a Γ_i, exceto pelo literal λ abduzido, que é removido da cláusula resultante Γ_{i+1}.

Por exemplo, seja $\Gamma_0 := \leftarrow enchente$ uma cláusula objetivo e $\Delta := \{enchente \leftarrow chuva, chuva \leftarrow verão, chuva \leftarrow umidade, verão\}$ um programa lógico. Como podemos observar na figura 4.1-a, resolvendo-se o literal *enchente* de Γ_0 com a cabeça da cláusula $enchente \leftarrow chuva$, obtemos $\Gamma_1 := \leftarrow chuva$; então, para resolver o literal *chuva* de Γ_1, temos duas possibilidades: usando a cláusula $chuva \leftarrow verão$, obtemos $\Gamma_2 := \leftarrow verão$ e, usando a cláusula $chuva \leftarrow umidade, calor$, obtemos $\Gamma'_2 := \leftarrow umidade, calor$. Em ambos os casos, entretanto, chegamos a um ponto onde nenhuma cláusula do programa pode ser usada para derivarmos uma nova cláusula objetivo ou a cláusula vazia. Logo, não há como refutar Γ_0. Por outro lado, como podemos observar na figura 4.1-b, abduzindo o literal *verão*, a cláusula Γ_2 nos permite derivar a cláusula vazia e, portanto, concluir a refutação. Analogamente, abduzindo o literal *umidade* em Γ'_2, obtemos a cláusula objetivo $\Gamma'_3 := \leftarrow calor$ e, finalmente, abduzindo o literal *calor* nessa última cláusula, encontramos uma outra refutação.

O procedimento que obtemos estendendo SLD-refutação com abdução, conforme descrevemos, é denominado SLDA-*refutação* [14] e o conjunto cumulativo de hipóteses

abdutivas Π, gerado por ele, é denominado *resíduo abdutivo*. Através desse procedimento, que foi provado correto e completo por *Denecker & De Schreye* [14], podemos estender a capacidade do PROLOG e permitir que, além de dedução, ele também seja capaz de realizar abdução.

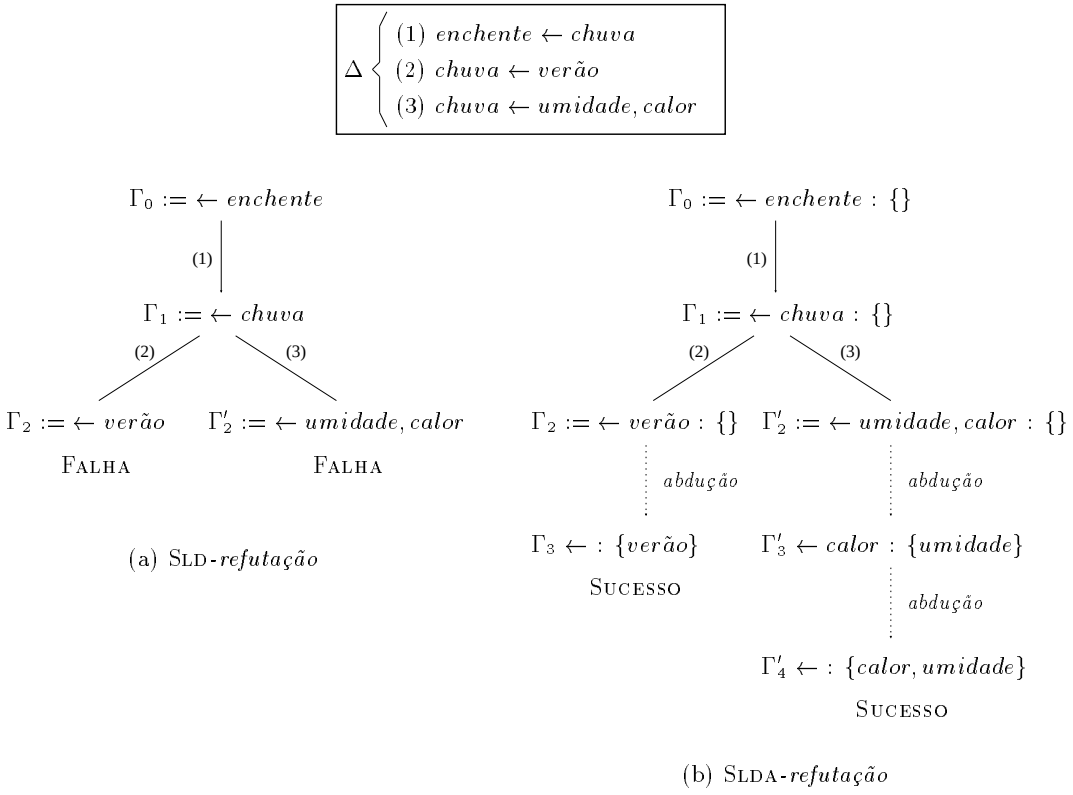


Figura 4.1: *Estendendo uma* SLD*-árvore com abdução.*

Um meta-interpretador que implementa o procedimento de SLDA-refutação pode ser visto na tabela 4.1. Nesse meta-interpretador, o metapredicado $axiom$ é usado para representar as cláusulas do programa-objeto no metanível. Assim, uma metacláusula da forma $axiom(\alpha, [\alpha_1, \ldots, \alpha_n])$ corresponde, de fato, a uma cláusula-objeto da forma $\alpha \leftarrow \alpha_1, \ldots, \alpha_n$. Isso permite que as cláusulas do meta-interpretador possam ser distinguidas daquelas que fazem parte do programa-objeto a ser manipulado, já que ambos os tipos de cláusulas são mantidos na mesma base de conhecimento (do PROLOG). Outro metapredicado usado é *abducible*; através dele podemos declarar o conjunto de fatos

abdutíveis no programa-objeto, garantindo assim que apenas explicações básicas sejam computadas [26]. Por exemplo, nas cláusulas [OBJ1]-[OBJ6] a seguir, esses metapredicados são utilizados para representar um programa-objeto no metanível.

```
axiom(chuva,[verão]).                                    [OBJ1]
axiom(chuva,[umidade,calor]).                            [OBJ2]
axiom(enchente,[chuva]).                                 [OBJ3]

abducible(verão).                                        [OBJ4]
abducible(umidade).                                      [OBJ5]
abducible(calor).                                        [OBJ6]
```

O meta-interpretador, propriamente dito, é implementado pelo predicado *slda*: em [SLDA1], a cláusula vazia é refutada trivialmente, sem que qualquer adição ao resíduo abdutivo seja necessária; em [SLDA2], o literal `G1`, selecionado da cláusula objetivo representada pela lista `[G1|Gs1]`, é resolvido com a cabeça de uma cláusula do programa-objeto, representada por `axiom(G1,Gs2)`, e o processo continua com a cláusula objeto `Gs3`, resultante da concatenação da sublista `Gs1` com a lista `Gs2`; finalmente, em [SLDA3], se o literal selecionado `G1` é abdutível, ele é adicionado ao resíduo abdutivo e o processo continua com a cláusula objetivo `Gs1`.

```
slda([],R,R).                                            [SLDA1]

slda([G1|Gs1],R1,R2) :- % passo de resolução             [SLDA2]
   axiom(G1,Gs2),
   append(Gs2,Gs1,Gs3),
   slda(Gs3,R1,R2).

slda([G1|Gs1],R1,R2) :- % passo de abdução               [SLDA3]
   abducible(G1),
   slda(Gs1,[G1|R1],R2).
```

Tabela 4.1: SLDA-*refutação em* PROLOG.

Então, dado um programa-objeto, uma metaconsulta `?- slda([`g_1`,...,`g_k`],[],R).` será bem sucedida se e somente se $g_1 \wedge \cdots \wedge g_k$ for uma conseqüência lógica da conjunção do resíduo abdutivo R com as cláusulas desse programa-objeto. Por exemplo, com

base no programa-objeto representado pelas cláusulas [OBJ1]-[OBJ6], a metaconsulta `?- slda([enchente],[],R).` fornecerá duas soluções: `R=[verão]` e `R=[calor,umidade]`.

Um cuidado especial deve ser tomado com relação às variáveis existentes num átomo a ser abduzido. Se tal átomo é tratado como uma cláusula unitária que pode ser adicionada diretamente ao programa, então suas variáveis que são existenciais passam a ser universais, o que é uma hipótese demasiadamente forte [15]. Assim, para garantir a corretude do meta-interpretador *slda*, as variáveis em átomos abduzidos deveriam ser *skolemizadas*, *i.e.* substituídas por constantes inéditas no contexto [12]. Ademais, como o PROLOG emprega busca em profundidade, esse meta-interpretador não é completo. Entretanto, utilizando busca em profundidade iterativa, limitada pelo número de fatos existentes no resíduo abdutivo, podemos torná-lo completo e ainda garantir que explicações minimais e compactas sejam sempre encontradas primeiro [55].

4.1.2 Estendendo abdução com negação por falha

Na programação em lógica, a negação é implementada através de um mecanismo, introduzido por *Clark* [10], conhecido como *negação por falha*. Esse mecanismo, que pressupõe a *hipótese do mundo fechado* [50], estabelece que a negação de um átomo ϕ pode ser inferida de um programa lógico Δ se não há uma SLD-refutação para a cláusula objetivo $\leftarrow \phi$ (*prova em modo negativo*), a partir de Δ. (Note que, assim como a abdução, a negação por falha é uma operação não-monotônica, já que a inclusão de novas cláusulas em Δ pode eliminar conclusões negativas anteriormente provadas.) Quando a negação por falha é embutida na SLD-refutação, o procedimento resultante, apresentado na tabela 4.2, é denominado SLDNF-*refutação* [1].

Nesse novo meta-interpretador, implementado pelo predicado *sldnf*, um literal `neg(G1)` é tomado como verdadeiro se não é possível provar que o literal complementar `G1` é uma conseqüência lógica do programa-objeto considerado (*vide* cláusula [SLDNF3]).

Em princípio, juntar abdução e negação por falha parece ser simples. Entretanto, por serem operações não-monotônicas, abdução e negação por falha interferem mutuamente entre si e, sendo assim, um cuidado especial deve ser tomado. Na negação por falha apenas as conseqüências lógicas do programa mais o resíduo abdutivo já computado devem ser consideradas e, portanto, a abdução deve ser desabilitada durante as

```
sldnf([]).                                                    [SLDNF1]

sldnf([G1|Gs1]) :- % passo de resolução                       [SLDNF2]
   axiom(G1,Gs2),
   append(Gs2,Gs1,Gs3),
   sldnf(Gs3).

sldnf([not(G1)|Gs]) :- % negação por falha                    [SLDNF3]
   not dem([G1]),
   sldnf(Gs).
```

Tabela 4.2: SLDNF-*refutação em* PROLOG.

provas em modo negativo. Por outro lado, quando novas hipóteses são adicionadas ao resíduo abdutivo, conclusões negativas previamente provadas podem se tornar falsas. Por exemplo, suponha que o literal selecionado numa consulta seja `not(G)`; então, o método usual de negação por falha é adotado e `not(G)` é assumido como verdadeiro se `G` não pode ser provado a partir do programa mais o resíduo corrente. Porém, mais tarde, novas adições ao resíduo abdutivo podem tornar `G` provável, o que deve ser evitado. Assim, para manter a corretude da negação por falha, somente hipóteses consistentes com as negações já provadas (*resíduo negativo*) podem ser adicionadas ao resíduo abdutivo.

Estendendo-se SLD-refutação com negação por falha e abdução, simultaneamente, obtemos um procedimento denominado SLDNFA-*refutação* [12, 55, 15, 14, 26, 57]. Esse procedimento é implementado pelo meta-interpretador apresentado na tabela 4.3.

Dados um programa Δ e uma consulta G, a metaconsulta `?- sldnfa(G,[],R,[],N)` produz um resíduo abdutivo R e um resíduo negativo N tais que $\Delta \cup R$ é consistente e $Comp(\Delta \cup R)$[1] $\models \{G\} \cup N$.

Resumidamente, conforme observamos na tabela 4.3, o meta-interpretador *sldnfa* faz o seguinte: em `[SLDNFA1]`, a cláusula vazia é refutada trivialmente; em `[SLDNFA2]`, temos um passo de resolução, onde o literal selecionado `G1` é resolvido com uma cláusula do programa-objeto; em `[SLDNFA3]`, se o literal selecionado `G1` é abdutível, e consistente com as negações já provadas e coletadas na lista `N1`, ele é adicionado ao resíduo abdutivo; finalmente, em `[SLDNFA4]`, se o literal é negativo e pode ser demonstrado por falha, a

[1] $Comp(\mathcal{P})$ denota o completamento de *Clark* para o programa lógico $\mathcal{P}$.

partir da união do programa-objeto com o resíduo abdutivo corrente `R1`, ele é adicionado ao resíduo negativo `N1`.

```
sldnfa([],R,R,N,N).                                                    [SLDNFA1]
sldnfa([G1|Gs1],R1,R2,N1,N2) :- % passo de resolução                   [SLDNFA2]
   axiom(G1,Gs2),
   append(Gs2,Gs1,Gs3),
   sldnfa(Gs3,R1,R2,N1,N2).
sldnfa([G1|Gs1],R1,R2,N1,N2) :- % passo de abdução                     [SLDNFA3]
   abducible(G1),
   check_naf(N1,[G1|R1]),
   sldnfa(Gs1,[G1|R1],R2,N1,N2).
sldnfa([not(G1)|Gs1],R1,R2,N1,N2) :- % negação por falha               [SLDNFA4]
   naf([G1],R1),
   sldnfa(Gs1,R1,R2,[[G1]|N1],N2).

check_naf([],R).                                                       [SLDNFA5]
check_naf([N|Ns],R) :-                                                 [SLDNFA6]
   naf(N,R),
   check_naf(Ns,R).

naf([G1|_],R) :-                                                       [SLDNFA7]
   not resolve(G1,R,_).
naf([G1|Gs1],R) :-                                                     [SLDNFA8]
   findall(Gs3,(resolve(G1,R,Gs2),append(Gs2,Gs1,Gs3)),Gss),
   check_naf(Gss,R).

resolve(G1,R,Gs) :-                                                    [SLDNFA9]
   axiom(G1,Gs).
resolve(G1,R,[]) :-                                                   [SLDNFA10]
   member(G1,R).
```

Tabela 4.3: SLDNFA-*refutação em* PROLOG.

Para evitar que conclusões negativas *provadas por falha* tornem-se falsas, o meta-interpretador $sldnfa$ as coleta numa lista de negações (*resíduo negativo*), cuja consistência é checada pelo metapredicado $check\_naf$ toda vez que o resíduo abdutivo é modificado. De fato, uma lista de negações N da forma $[[\gamma_1^1, \ldots, \gamma_{m_1}^1], \ldots, [\gamma_1^n, \ldots, \gamma_{m_n}^n]]$ representa uma conjunção $\neg(\gamma_1^1 \wedge \cdots \wedge \gamma_{m_1}^1) \wedge \cdots \wedge \neg(\gamma_1^n \wedge \cdots \wedge \gamma_{m_n}^n)$ e a fórmula $check\_naf(N, R)$ vale se essa conjunção é provável a partir do completamento da conjunção do resíduo abdutivo R com o programa-objeto. Então, para garantir a consistência de N em relação a R, em [SLDNFA5] e [SLDNFA6], o metapredicado $check\_naf$ refaz a prova por falha de cada um dos conjuntos de N. Para tanto, ele aplica a esses conjuntos o metapredicado auxiliar naf, que é definido em termos do predicado `findall`, nativo do SWI-PROLOG. A justificativa para as cláusulas [SLDNFA7] e [SLDNFA8], que implementam o metapredicado naf, é a seguinte: para mostrar que $\neg(\gamma_1 \wedge \cdots \wedge \gamma_n)$ vale, é preciso mostrar que, para toda cláusula-objeto λ :- $\lambda_1, \ldots, \lambda_n$ que resolve com γ_1, a fórmula $\neg(\lambda_1 \wedge \cdots \wedge \lambda_n \wedge \gamma_2 \wedge \cdots \wedge \gamma_n)$ também vale. Se nenhuma cláusula resolve com γ_1, então, sob a semântica do completamento, $\neg\gamma_1$ vale e, conseqüentemente, temos $\neg(\gamma_1 \wedge \cdots \wedge \gamma_n)$. Finalmente, as cláusulas [SLDNFA9] e [SLDNFA10] implementam a conjunção do programa-objeto com o resíduo abdutivo.

4.1.3 Planejamento como uma tarefa abdutiva

Eshghi [17] foi o primeiro a mostrar que planejamento pode ser visto como uma tarefa abdutiva. Basicamente, a idéia introduzida por ele foi a seguinte: sejam β uma meta de planejamento e $\alpha \rightarrow \beta$ uma lei causal, onde α é uma ação do domínio de planejamento e β é um de seus efeitos; então, supondo que a ocorrência dessa ação seja um fato abdutível, a explicação abdutiva $\{\alpha\}$ representa um plano para atingir β.

De fato, planos podem ser vistos como explicações de como um estado meta é atingido [17, 55, 57] e, portanto, a abdução é uma regra de inferência bastante adequada para resolver problemas de planejamento. Como veremos na subseção 4.3.1, dados um conjunto de fórmulas Δ, descrevendo as ações do domínio, um conjunto de fórmulas Σ, descrevendo um estado inicial, e um conjunto de fórmulas Γ, descrevendo um estado meta, o planejamento abdutivo consiste em encontrar um conjunto de fórmulas Π (resíduo abdutivo), descrevendo ações e restrições de ordem entre elas, tal que $\Delta \cup \Sigma \cup \Pi \models \Gamma$.

4.2 Cálculo de eventos

O CÁLCULO DE EVENTOS, introduzido originalmente por *Kowalski & Sergot* [34], é um formalismo lógico para raciocínio sobre ações e efeitos. Diferentemente do cálculo de situações, entretanto, o cálculo de eventos enfatiza os eventos que ocorrem no mundo e não as situações pelas quais o mundo passa. Nesse formalismo, as primitivas ontológicas consistem de *eventos*, *fluentes* e *instantes de tempo* [56]. Informalmente, a idéia básica do cálculo de eventos é estabelecer que um fluente *vale* num determinado instante do tempo se ele vale *inicialmente*, ou foi *iniciado* pela ocorrência de uma ação em algum instante anterior, e não foi *terminado* pela ocorrência de uma outra ação nesse meio tempo. A tabela 4.4 lista os predicados[2] usados nesse formalismo.

predicado	*significado*
$initiates(\alpha, \phi, \tau)$	*o fluente* ϕ *começa a valer após a ação* α*, no instante* τ
$terminates(\alpha, \phi, \tau)$	*o fluente* ϕ *deixa de valer após a ação* α*, no instante* τ
$releases(\alpha, \phi, \tau)$	*o fluente* ϕ *tem valor indefinido após a ação* α*, no instante* τ
$initially_p(\phi)$	*o fluente* ϕ *vale no instante inicial 0*
$initially_n(\phi)$	*o fluente* ϕ *não vale no instante inicial 0*
$happens(\alpha, \tau_1, \tau_2)$	*a ação* α *inicia-se no instante* τ_1 *e termina no instante* τ_2
$holdsAt(\phi, \tau)$	*o fluente* ϕ *vale no instante* τ
$clipped(\tau_1, \phi, \tau_2)$	*o fluente* ϕ *deixa de valer entre os instantes* τ_1 *e* τ_2
$declipped(\tau_1, \phi, \tau_2)$	*o fluente* ϕ *começa a valer entre os instantes* τ_1 *e* τ_2

Tabela 4.4: *Predicados do cálculo de eventos.*

4.2.1 Uma axiomatização para o cálculo de eventos

A tabela 4.5, a seguir, apresenta uma axiomatização para o cálculo de eventos, baseada em [57]. De acordo com o axioma [EC1], o predicado $holdsAt(F, T)$ é verdade se o fluente F vale no instante 0, representado por $initially_p(F)$, e não deixa de valer durante o intervalo de tempo $[0, T]$, representado por $\neg clipped(0, F, T)$. De acordo com o axioma [EC2], o predicado $holdsAt(F, T)$ é verdadeiro se o fluente F vale após a ocorrência de uma ação A, que o produz como efeito, num instante anterior a T, representado

[2] *Por uma questão de padronização, os símbolos no cálculo de eventos serão mantidos em inglês.*

por $happens(A, T_1, T_2) \wedge initiates(A, F, T_1) \wedge T_2 \prec T$, e não deixa de valer durante o intervalo de tempo $[T_1, T]$, representado por $\neg clipped(T_1, F, T)$. De acordo com [EC5], o predicado $clipped(T_1, F, T_2)$ é verdade se ocorre um evento no intervalo $[T_1, T_2]$, representado por $happens(A, T_3, T_4) \wedge T_1 \prec T_3 \wedge T_4 \prec T_2$, e esse evento torna o valor do fluente F falso ou indefinido, representado por $terminates(A, F, T_3) \vee releases(A, F, T_3)$. (Note que o predicado *releases* serve para estabelecer efeitos não-determinísticos de uma ação. Um exemplo clássico do uso desse predicado é o *problema da roleta russa* [24, 56], onde o fluente *bala_na_agulha* teria um valor indefinido após girarmos o tambor de um revólver carregado com uma única bala.) Analogamente a [EC1], [EC2] e [EC5], os axiomas [EC3], [EC4] e [EC6] estabelecem as condições para que a *negação* do predicado $holdsAt(F, T)$ seja verdadeira. E, finalmente, o axioma [EC7] estabelece que o instante do início da ocorrência de uma ação precede o instante do seu término.

$holdsAt(F, T) \leftarrow$ [EC1]
$\quad initially_p(F) \wedge \neg clipped(0, F, T)$

$holdsAt(F, T) \leftarrow$ [EC2]
$\quad happens(A, T_1, T_2) \wedge initiates(A, F, T_1) \wedge T_2 \prec T \wedge \neg clipped(T_1, F, T)$

$\neg holdsAt(F, T) \leftarrow$ [EC3]
$\quad initially_n(F) \wedge \neg declipped(0, F, T)$

$\neg holdsAt(F, T) \leftarrow$ [EC4]
$\quad happens(A, T_1, T_2) \wedge terminates(A, F, T_1) \wedge T_2 \prec T \wedge \neg declipped(T_1, F, T)$

$clipped(T_1, F, T_2) \leftrightarrow$ [EC5]
$\quad \exists A, T_3, T_4[happens(A, T_3, T_4) \wedge T_1 \prec T_3 \wedge T_4 \prec T_2 \wedge$
$\qquad (terminates(A, F, T_3) \vee releases(A, F, T_3))]$

$declipped(T_1, F, T_2) \leftrightarrow$ [EC6]
$\quad \exists A, T_3, T_4[happens(A, T_3, T_4) \wedge T_1 \prec T_3 \wedge T_4 \prec T_2 \wedge$
$\qquad (initiates(A, F, T_3) \vee releases(A, F, T_3))]$

$happens(A, T_1, T_2) \rightarrow T_1 \preceq T_2$ [EC7]

Tabela 4.5: *Axiomatização para o cálculo de eventos.*

4.2.2 Estado inicial, ações e planos no cálculo de eventos

Além dos axiomas [EC1]-[EC7], independentes de domínio, precisamos de axiomas que descrevam que fluentes valem inicialmente, bem como os efeitos das ações num domínio específico. No cálculo de eventos, o estado inicial do mundo é definido através dos predicados $initially_p$ e $initially_n$, enquanto os efeitos das ações são estabelecidos através dos predicados *initiates* e *terminates*. Por exemplo, a seguir, os axiomas [R1] e [R2] definem um estado inicial onde o nosso robô do mundo dos blocos encontra-se num local l_0, segurando um bloco b, enquanto os axiomas [R3] e [R4] estabelecem, respectivamente, os efeitos positivos e negativos da ação $walk(X, Y)$, que faz com que esse robô ande de um local X até outro local Y.

$$initially_p(at(l_0)) \quad [R1]$$

$$initially_p(holding(b)) \quad [R2]$$

$$initiates(walk(X,Y), at(Y), T) \leftarrow holdsAt(at(X), T) \ \wedge \ X \neq Y \quad [R3]$$

$$terminates(walk(X,Y), at(X), T) \leftarrow holdsAt(at(X), T) \ \wedge \ X \neq Y \quad [R4]$$

No cálculo de eventos, um plano é representado por um conjunto de fatos *happens*[3], estabelecendo a ocorrência de ações no tempo, e por um conjunto de restrições de ordem temporal $\prec$, estabelecendo uma ordem parcial sobre essas ações. Por exemplo, $\{happens(walk(l_0, l_1), t_1), happens(walk(l_1, l_2), t_2), t_0 \prec t_1, t_1 \prec t_2, t_2 \prec t_3\}$ é um plano.

4.2.3 Persistência temporal no cálculo de eventos

Dado um conjunto de fatos *happens* e $\prec$, representando um plano parcialmente ordenado, a partir dos axiomas [EC1]-[EC7] e [R1]-[R4], podemos determinar a validade dos fluentes do domínio em qualquer instante futuro do tempo, após a execução desse plano. Por exemplo, dado o plano $\{happens(walk(l_0, l_1), t_1), happens(walk(l_1, l_2), t_2), t_0 \prec t_1, t_1 \prec t_2, t_2 \prec t_3\}$, podemos concluir tanto $holdsAt(at(l_2), t_3)$, que é um efeito direto da ação $walk(l_1, l_2)$, quanto $holdsAt(holding(b), t_3)$ que é uma propriedade que persiste no tempo, desde o instante inicial 0. De fato, a axiomatização apresentada na tabela 4.5 captura, essencialmente, a *persistência temporal* dos fluentes e, portanto, ao contrário do cálculo de situações, o cálculo de eventos não requer o uso de axiomas de quadro.

[3] Uma versão binária desse predicado é definida como $happens(\alpha, \tau) \equiv_{def} happens(\alpha, \tau, \tau)$.

Conforme [55], a persistência embutida na axiomatização do cálculo de eventos é baseada em quatro pressupostos básicos:

- nenhum evento ocorre além daqueles que são conhecidos,
- nenhum evento afeta um dado fluente além daqueles que são conhecidos,
- os fluentes persistem até a ocorrência de algum evento que os afete, e
- todo fluente é efeito de algum evento conhecido.

4.3 Planejamento abdutivo no cálculo de eventos

Em vez de seguir a abordagem lógica sugerida por *Green* [23], onde a tarefa de planejamento é definida em termos de dedução no cálculo de situações, *Eshghi* [17] propôs um sistema de planejamento baseado em abdução no cálculo de eventos. Com esse novo paradigma de planejamento lógico, além de resolver o problema da persistência temporal, *Eshghi* mostrou que planos parcialmente ordenados também podem ser sintetizados como efeito colateral da prova de teoremas.

4.3.1 Especificação lógica de planejamento abdutivo

Definição 4.2 *Um domínio de planejamento é uma conjunção finita de fórmulas da forma*

$initiates(\alpha, \phi, \tau) \leftarrow \beta,$
$terminates(\alpha, \phi, \tau) \leftarrow \beta,$ *ou*
$releases(\alpha, \phi, \tau) \leftarrow \beta,$

onde β *é da forma* $(\neg)holdsAt(\phi_1, \tau) \wedge \cdots \wedge (\neg)holdsAt(\phi_n, \tau)$, α *é um termo não-variável denotando uma ação,* ϕ, $\phi_1, \ldots, \phi_n$ *são termos não-variáveis denotando fluentes e* τ *é um termo denotando um instante de tempo.* □

Definição 4.3 *Uma situação inicial é uma conjunção finita de fórmulas da forma*

$initially_n(\phi)$, *ou*
$initially_p(\phi)$,

onde ϕ *é um termo livre de variáveis denotando um fluente, na qual cada fluente ocorre no máximo uma vez.* □

Definição 4.4 *Uma meta é uma conjunção finita de fórmulas da forma*

$(\neg) holdsAt(\phi, \tau)$,

onde ϕ é um termo livre de variáveis denotando um fluente e τ é um termo constante denotando um instante de tempo. □

Definição 4.5 *Uma narrativa é uma conjunção finita de fórmulas da forma*

$happens(\alpha, \tau)$, *ou*
$\tau_1 \prec \tau_2$,

onde α é um termo livre de variáveis denotando uma ação e τ, τ_1, τ_2 são termos constantes denotando instantes de tempo. □

Definição 4.6 *Sejam Δ um domínio, Σ uma situação inicial, EC a conjunção dos axiomas do cálculo de eventos e Γ uma meta de planejamento. Um plano para atingir Γ é uma narrativa Π tal que*

$\textsc{Circ}[\Delta; initiates, terminates, releases] \wedge \textsc{Circ}[\Sigma \wedge \Pi; happens] \wedge EC \models \Gamma$ *e*
$\textsc{Circ}[\Delta; initiates, terminates, releases] \wedge \textsc{Circ}[\Sigma \wedge \Pi; happens] \wedge EC$ *é consistente.* □

Circunscrevendo[4] os predicados *initiates*, *terminates* e *releases* assumimos que as ações não têm efeitos inesperados e circunscrevendo o predicado *happens*, que não há ocorrências de eventos inesperados. Conforme *Shanahan* [57], para garantir a condição de consistência na definição acima, basta garantir que o domínio Δ seja *livre de conflitos.*

Definição 4.7 *Um domínio é livre de conflitos se, para todo par de fórmulas em Δ da forma $initiates(\alpha, \phi, \tau) \leftarrow \beta_1$ e $terminates(\alpha, \phi, \tau) \leftarrow \beta_2$, temos $\models \neg(\beta_1 \wedge \beta_2)$* □

4.4 Implementando um meta-interpretador abdutivo para cálculo de eventos

Para transformar a especificação lógica de planejamento abdutivo numa implementação prática, *Shanahan* [57] propôs o uso de um meta-interpretador abdutivo especializado

[4] A definição da operação de circunscrição, CIRC, foi apresentada na subseção 2.3.1.

para o cálculo de eventos. Então, dados os axiomas descrevendo o problema de planejamento, o meta-interpretador produziria um resíduo representando um plano que seria uma solução para o referido problema.

4.4.1 Compilando cláusulas-objetos em metacláusulas

Para especializar um meta-interpretador, digamos *dem*, com relação a uma cláusula-objeto $\alpha_0 \leftarrow \alpha_1, \ldots, \alpha_n$, basta embutir o conhecimento expresso por essa cláusula, diretamente, na definição do metapredicado *dem*. Isso pode ser feito, por exemplo, substituindo-se a metacláusula `axiom(`α_0`,[`α_1`,...,`α_n`])`, que representa a referida cláusula-objeto, por um nova metacláusula `dem([`α_0`|Gs1]) :- dem([`α_1`,...,`α_n`|Gs1])`, que é adicionada à definição do meta-interpretador.

Segundo *Levi & Ramundo* [36], o resultado obtido por um meta-interpretador especializado com a compilação de uma cláusula objeto é equivalente àquele obtido pelo meta-interpretador básico correspondente, na presença da cláusula-objeto que foi compilada. Entretanto, devido ao grau de controle extra disponível no metanível, uma série de manobras podem ser realizadas durante a compilação de uma cláusula-objeto. Através dessas manobras podemos, por exemplo, evitar laços infinitos, alterando a ordem em que os literais no corpo da cláusula são resolvidos (*vide* subseção 4.4.2), e considerar conhecimento incompleto, dando um tratamento especial a predidados para os quais não temos uma definição completa (*vide* subseção 4.4.3).

4.4.2 Compilação de axiomas do cálculo de eventos

Considere um meta-interpretador básico *dem*, que simula exatamente o funcionamento do PROLOG. Suponha, por exemplo, que desejamos especializá-lo com relação à cláusula-objeto a seguir, correspondente ao axioma [EC2] do cálculo de eventos, onde o predicado *before* é empregado para representar uma restrição de ordem temporal.

```
holdsAt(F,T) :-                                              [EC2]
   happens(A,T1,T2),
   initiates(A,F,T1),
   before(T2,T),
   not clipped(T1,F,T).
```

Então, podemos adicionar a seguinte metacláusula à definição do meta-interpretador:

```
dem([holdsAt(F,T)|Gs1]) :-                                    [EC2C]
   axiom(initiates(A,F,T1),Gs2),
   axiom(happens(A,T1,T2),[]),
   axiom(before(T2,T),[]),
   not dem([clipped(T1,F,T)]),
   append(Gs2,Gs1,Gs3),
   dem(Gs3).
```

Embora os resultados obtidos com essa versão especializada do meta-interpretador sejam os mesmos obtidos através de sua versão básica na presença de [EC2] (ou melhor, na presença da cláusula de *axiom* que descreve [EC2]), para uma mesma consulta com o predicado *holdsAt*, a computação realizada pela versão especializada não corresponde exatamente àquela realizada pela versão básica. Isso acontece devido a algumas manobras que foram realizadas na forma compilada de `[EC2]`, que denominamos `[EC2C]`:

- *A ordem dos literais* `happens(A,T1,T2)` *e* `initiates(A,F,T1)` *foi invertida.* Com isso, podemos obter um espaço de busca bem menor que aquele que teríamos se esses literais fossem tratados na ordem oposta [9]. Em geral, num mundo dinâmico há inúmeros eventos que podem ocorrer (*happens*), embora apenas alguns deles sejam capazes de contribuir (*initiates*) com um determinado efeito desejado; então, resolvendo-se `initiates(A,F,T1)` antes de `happens(A,T1,T2)`, garantimos que eventos irrelevantes para o fluente `F` não sejam considerados. Em decorrência disso, podemos ter uma ramificação bem menor na árvore de refutação e, conseqüentemente, a computação realizada poderá ser bem mais eficiente.

- *O tratamento das precondições de* `initiates(A,F,T1)` *foi adiado.* Embora o literal `initiates(A,F,T1)` seja resolvido imediatamente no início da metacláusula, o tratamento de suas precondições é adiado até que os literais `happens(A,T1,T2)` e `before(T2,T)` também tenham sido resolvidos. Com isso, evitamos a possibilidade de ocorrência de laços infinitos; já que *holdsAt* é definido em termos de *initiates* e *vice-versa*.

- *A prova do literal* `not clipped(T1,F,T)` *foi antecipada.* Embutindo a negação por falha do literal `clipped(T1,F,T)` no próprio corpo da metacláusula `[EC2C]`,

forçamos que suas precondições sejam tratadas antes das precondições do literal `initiates(A,F,T1)`. Com isso, garantimos que o processo continuará apenas quando a ação `A`, considerada na resolução de `initiates(A,F,T1)`, for realmente efetiva, ou seja, quando o efeito `F` produzido por ela não for necessariamente anulado pela ocorrência de uma outra ação, já postulada no programa-objeto.

A compilação do axioma [EC4] pode ser feita de maneira análoga àquela apresentada para o axioma [EC2]. A novidade, porém, é que como [EC4] não é uma cláusula definida, precisamos introduzir a função `neg` para representar esse axioma. Com essa função, fórmulas do cálculo de eventos da forma $\neg holdsAt(F,T)$ podem ser escritas em PROLOG como `holdsAt(neg(F),T)`. Assim, a cláusula-objeto correspondente ao axioma [EC4] será

```
holdsAt(neg(F),T) :-
   happens(A,T1,T2),
   terminates(A,F,T1),
   before(T2,T),
   not declipped(T1,F,T).
```

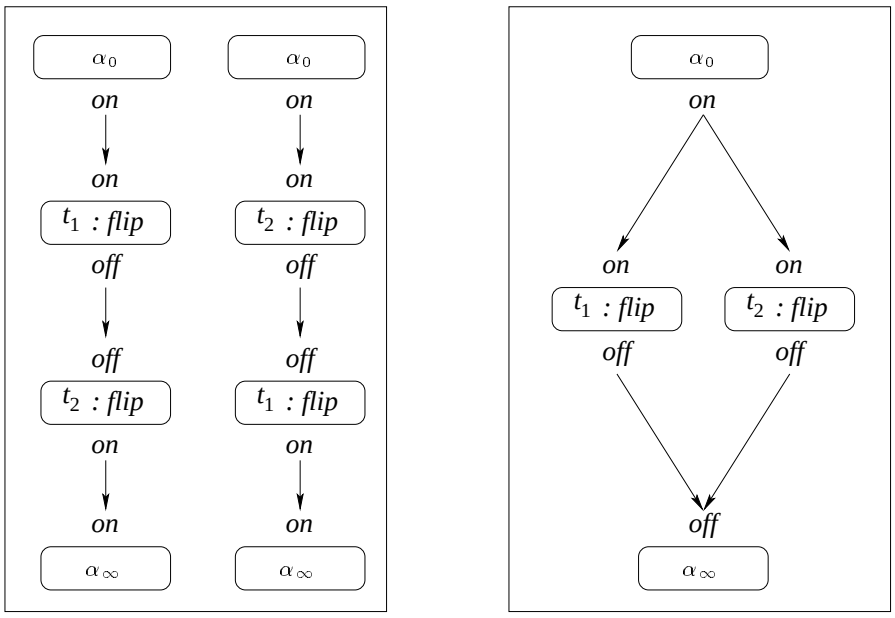
cuja compilação resultará na seguinte metacláusula:

```
dem([holdsAt(neg(F),T)|Gs1]) :-
   axiom(terminates(A,F,T1),Gs2),
   axiom(happens(A,T1,T2),[]),
   axiom(before(T2,T),[]),
   not dem([declipped(T1,F,T)]),
   append(Gs2,Gs1,Gs3),
   dem(Gs3).
```

4.4.3 Tratamento de conhecimento incompleto

Um dos problemas da negação por falha é a sua incapacidade de tratar conhecimento incompleto. Através dela, todo fato que não é explicitamente estabelecido como verdade é automaticamente assumido como sendo falso. Entretanto, no raciocínio temporal com cálculo de eventos, temos informação incompleta sobre $before$ e, portanto, usar negação por falha com esse predicado pode resultar em conclusões incorretas.

Por exemplo, suponha que duas ações que comutam ($flip$) o interruptor de uma determinada lâmpada ocorram numa ordem desconhecida, ou seja, temos $happens(flip, t_1)$



(a) suposição do cálculo de eventos (b) suposição da negação por falha

Figura 4.2: *Conhecimento incompleto e negação por falha.*

e $happens(flip, t_2)$, mas não temos $before(t_1, t_2)$, nem $before(t_2, t_1)$. Então, caso a lâmpada esteja acesa inicialmente, concluímos que ela permanecerá acesa no final. Esse raciocínio parte do princípio de que o tempo é linear (*i.e.* necessariamente temos $t_1 \prec t_2$ ou $t_2 \prec t_1$) e, sendo assim, qualquer que seja a ordem das ocorrências, a última delas anulará o efeito da primeira (figura 4.2-a). Entretanto, se não estabelecemos explicitamente que t_1 precede t_2 ou *vice-versa*, a negação por falha assume que nem uma coisa nem outra vale; levando-nos a concluir, incorretamente, que a lâmpada ficará apagada no final (figura 4.2-b).

Assim, para garantir a corretude do raciocínio temporal no cálculo de eventos, temos que especializar o meta-interpretador abdutivo de tal forma que a linearidade da relação de ordem temporal ($t_1 \not\prec t_2 \rightarrow t_2 \prec t_1$) seja embutida no metanível [15, 57].

Ao abduzir um fato $before(t_1, t_2)$, o meta-interpretador deve garantir que t_1 e t_2 sejam distintos e que o fato $before(t_2, t_1)$ ainda não tenha sido abduzido. Se uma dessas duas condições não é satisfeita, o passo de abdução falha; senão, $before(t_1, t_2)$ é adicionado ao resíduo abdutivo e o fecho transitivo da relação representada pelo predicado $before$ é computado. Ademais, para provar a negação de um átomo $before(t_1, t_2)$, o

meta-interpretador tem que abduzir o átomo simétrico $before(t_2, t_1)$. Assim, se mais tarde a negação do literal $before(t_2, t_1)$ também tiver que ser provada, teremos uma falha. Como resultado dessas manobras, uma conclusão nunca poderá ser baseada na ausência simultânea de dois fatos *before* simétricos.

Note que *before* não é o único predicado sobre o qual temos conhecimento incompleto. Na verdade, considerações similares devem ser feitas quanto ao tratamento do predicado *holdsAt*: quando o meta-interpretador encontra um literal `not holdsAt(F,T)`, ele tenta provar `holdsAt(neg(F),T)` e, inversamente, quando encontra um literal `not holdsAt(neg(F),T))`, ele tenta provar `holdsAt(F,T)`. Como a validade de *holdsAt* depende da ordem de ocorrência das ações abduzidas, em ambos os casos, podemos ter novas adições de restrições de ordem temporal ao resíduo. Outro predicado sobre o qual também temos conhecimento incompleto é *happens*; nesse caso, porém, a negação por falha desse predicado não causará problemas.

4.4.4 O sistema de planejamento AECP

O AECP (**A***bductive* **E***vent* **C***alculus* **P***lanner*), proposto por *Shanahan* [57], nada mais é que o resultado da compilação de todos os axiomas do cálculo de eventos, diretamente, em cláusulas de um meta-interpretador abdutivo, estendido com negação por falha e devidamente modificado para tratar conhecimento incompleto a respeito de restrições de ordem temporal. Dado um programa-objeto descrevendo um domínio de planejamento (cláusulas `initiates`, `terminates` e `releases`) e a situação inicial do mundo (cláusulas `initially`), fornecemos ao AECP uma lista de literais `holdsAt` representando a meta de planejamento e, como resposta, ele devolve um resíduo abdutivo, contendo literais `happens` e `before`, que representa o plano desejado.

Com base no AECP, implementamos três sistemas de planejamento abdutivo baseado no cálculo de eventos (APB, SAPB e RAPB). Esses sistemas, que diferem do AECP em vários aspectos importantes, serão descritos na seção 5.1.

Correspondência entre o AECP e o POP

Em [57], *Shanahan* estabelece uma correspondência entre o AECP e um algoritmo de planejamento de ordem parcial (POP) através de uma simples inspeção de código. Para isso, ele toma como exemplo a seguinte metacláusula:

```
aecp([holdsAt(F,T)|Gs1],R1,R4) :-                                    [EC2-1]
   axiom(initiates(A,F,T1),Gs2),                                     [EC2-2]
   check_naf(N,[happens(A,T1,T2),before(T2,T)|R1],R2),               [EC2-3]
   check_naf([clipped(T1,F,T)],R2,R3),                               [EC2-4]
   append(Gs2,Gs1,Gs3),                                              [EC2-5]
   aecp(Gs3,R3,R4,[clipped(T1,F,T)|N]).                              [EC2-6]
```

Em `[EC2-1]`, um pré-requisito `holdsAt(F,T)` é selecionado e, em `[EC2-2]`, uma ação é escolhida para suportá-lo. Então, em `[EC2-3]` e `[EC2-4]`, o plano é modificado com uma nova ação e uma nova restrição de ordem, necessárias para satisfazer o pré-requisito selecionado, e as possíveis ameaças ao vínculo causal recém-estabelecido são resolvidas. Em seguida, em `[EC2-5]`, o conjunto de pré-requisitos a serem satisfeitos é atualizado com as precondições da ação adicionada ao plano e, finalmente, em `[EC2-6]`, o planejador é chamado recursivamente com o plano modificado. Observe que esses são exatamente os passos executados pelo algoritmo POP, apresentado na tabela 3.3.

4.5 Considerações finais

Nesse capítulo introduzimos abdução como uma regra de inferência que possibilita o raciocínio hipotético. Então, através do uso de um meta-interpretador, mostramos que a capacidade dedutiva do PROLOG pode ser estendida com abdução. Também introduzimos o cálculo de eventos como um formalismo apropriado para resolver problemas de raciocínio temporal onde temos conhecimento incompleto sobre a ordem dos eventos e mostramos que esse formalismo, juntamente com abdução, nos permite reproduzir, naturalmente, o procedimento de um algoritmo de planejamento de ordem parcial.

No próximo capítulo, vamos estabecer uma correspondência mais precisa entre planejamento de ordem parcial e planejamento abdutivo no cálculo de eventos. Diferentemente da análise comparativa feita por *Shanahan* [57], baseada em simples inspeção de código, propomos uma análise comparativa em termos das estruturas de dados manipuladas, do espaço de busca explorado e da eficiência obtida.

Capítulo 5

Resultados experimentais

Nossa sabedoria vem de nossa experiência;
e nossa experiência, de nossos erros.

Sacha Guitry

Teatrólogo e Cineasta
(1885–1957)

5.1 Implementação dos sistemas comparados

Os resultados apresentados nesse capítulo foram obtidos comparando-se o desempenho de três sistemas de planejamento algorítmico, *i.e.* baseados na representação STRIPS, com relação àquele observado em três sistemas de planejamento lógico, *i.e.* baseados em cálculo de eventos abdutivo, correspondentes.

5.1.1 Planejadores baseados em STRIPS

A partir dos algoritmos de planejamento de ordem parcial, apresentados no capítulo 3, implementamos três sistemas de planejamento baseados em STRIPS proposicional: um planejador de ordem parcial (POP), sua versão sistemática (SNLP) e sua versão redundante (TWEAK). As implementações foram feitas em PROLOG, de tal forma que estruturas de dados (*e.g.* passos, restrições de ordem e vínculos causais) e procedimentos (*e.g.* estabelecimento de submetas, fecho transitivo da relação de ordem temporal e tratamento de conflitos) comuns fossem compartilhados por todos os sistemas. Em conseqüência disso, podemos garantir que as implementações diferem tão somente quanto à política de proteção (*vide* seção 3.4) e à estratégia de seleção de submetas adotadas em cada um dos planejadores.

5.1.2 Planejadores abdutivos baseados em cálculo de eventos

Uma característica interessante do AECP, proposto por *Shanahan* [57], é que ele permite uma representação mais expressiva do mundo. Por exemplo, usando o predicado $happens(\alpha, \tau_i, \tau_f)$, podemos representar ações com tempo de duração e, usando o predicado $releases(\alpha, \phi, \tau)$, podemos representar ações com efeitos não-determinísticos. Entretanto, para que pudéssemos realizar uma comparação realmente justa com os planejadores baseados em STRIPS, algumas modificações foram feitas no AECP. Essas modificações ([M1] a [M6]), descritas a seguir, foram inspiradas em idéias provenientes de diversos trabalhos de análise conceitual da tarefa de planejamento [4, 5, 6].

M1: Um cálculo de eventos simplificado

O planejamento clássico é baseado em três hipóteses fundamentais: (*i*) *tempo atômico*, (*ii*) *efeitos determinísticos* e (*iii*) *onisciência* [61]. Da hipótese (*i*), segue que precisamos mudar o predicado $happens(\alpha, \tau_i, \tau_f)$ para a versão binária $happens(\alpha, \tau)$, de modo que apenas ações instantâneas sejam consideradas. Em conseqüência dessa mudança, o axioma [EC7] não será mais necessário. Da hipótese (*ii*), segue que o predicado *releases* não pode ser usado, já que não são admitidas ações com efeitos indeterminados. Isso implica em uma modificação no axioma [EC5], que define o predicado *clipped*. Finalmente, da hipótese (*iii*), mais o fato de que a representação STRIPS não admite precondições negativas, segue que não há mais necessidade de termos o predicado $initially_n$, nem os axiomas [EC3], [EC4] e [EC6]. Com essas mudanças, podemos especificar uma axiomatização simplificada para o cálculo de eventos (*vide* tabela 5.1), incluindo apenas aspectos relevantes ao planejamento clássico.

$holdsAt(F,T) \leftarrow$ $\quad initially_p(F) \wedge \neg clipped(0,F,T)$	[SEC1]
$holdsAt(F,T) \leftarrow$ $\quad happens(A,T_1) \wedge initiates(A,F,T_1) \wedge (T_1 \prec T) \wedge \neg clipped(T_1,F,T)$	[SEC2]
$clipped(T_1,F,T_2) \leftrightarrow$ $\quad \exists A,T[happens(A,T) \wedge (T_1 \prec T) \wedge (T \prec T_2) \wedge terminates(A,F,T)]$	[SEC3]

Tabela 5.1: *Axiomatização simplificada para o cálculo de eventos.*

A partir dessa axiomatização simplificada, e das idéias provenientes do AECP, implementamos um sistema de planejamento que denominamos ABP (**Ab***ducive* **P***lanner*). Como os axiomas [SEC1]-[SEC3] equivalem ao critério de verdade modal de *Chapman* [8], que é correto, segue que o sistema que implementamos é também correto. Ademais, como utilizamos busca em profundidade iterativa [53] na implementação desse sistema, segue que ele é completo.

M2: Representação de ações

Considere, por exemplo, a ação de andar de um local X até outro local Y, designada pelo termo $walk(X,Y)$. Em STRIPS, essa ação poderia ser representada pelo operador

$$oper(act : walk(X,Y),\ pre : \{at(X), X \neq Y\},\ add : \{at(Y)\},\ del : \{at(X)\}),$$

cuja representação correspondente no cálculo eventos seria a seguinte:

$$initiates(walk(X,Y), at(Y), T) \leftarrow holdsAt(at(X), T) \wedge X \neq Y$$
$$terminates(walk(X,Y), at(X), T) \leftarrow holdsAt(at(X), T) \wedge X \neq Y$$

Na nossa implementação dos planejadores baseados em STRIPS, operadores são representados através do predicado $oper(I,P,A,D)$, onde I é o identificador da ação, P é o seu conjunto de precondições, A é o seu conjunto de efeitos positivos e D é o seu conjunto de efeitos negativos. Assim, o operador $walk(X,Y)$ seria codificado como:

```
oper(walk(X,Y),[holdsAt(at(X),T), diff(X,Y)],[at(Y)],[at(X)]).
```

Por outro lado, na implementação do AECP, as cláusulas para os predicados *initiates* e *terminates* são representadas no metanível através do predicado $axiom(H,B)$, onde H é a cabeça da cláusula e B é o seu corpo. Assim, por exemplo, as cláusulas do cálculo de eventos que descrevem a ação $walk(X,Y)$ seriam codificadas da seguinte forma:

```
axiom(initiates(walk(X,Y),at(Y),T),[holdsAt(at(X),T), diff(X,Y)]).
axiom(terminates(walk(X,Y),at(X),T),[holdsAt(at(X),T), diff(X,Y)]).
```

Note que, na codificação da representação STRIPS, o primeiro parâmetro do predicado *oper* é o nome da ação, enquanto na codificação da representação do cálculo de eventos, o primeiro parâmetro do predicado *axiom* é o efeito da ação. Como o

PROLOG utiliza o primeiro parâmetro do predicado como chave de indexação[1], a busca da descrição de uma ação através do predicado *oper* deve tomar tempo aproximadamente constante, enquanto a mesma busca através do predicado $axiom$ deve tomar tempo proporcional ao número de cláusulas para o predicado $axiom$ existentes na base de conhecimento. Assim, a fim de estabelecer uma melhor correspondência entre planejadores lógicos e algorítmicos, na implementação do ABP, as cláusulas da forma $axiom(initiates(\alpha, \phi, \tau), B)$ são mantidas no metanível como $initiates(\alpha, \phi, \tau, B)$. Analogamente, cláusulas da forma $axiom(terminates(\alpha, \phi, \tau), B)$ são mantidas como $initiates(\alpha, \phi, \tau, B)$. Com essa modificação, esperamos que o acesso à descrição de uma ação nos planejadores lógicos seja tão eficiente quanto nos planejadores algorítmicos.

M3: Predicados abdutíveis e executáveis

No AECP, os metapredicados *abducible* e $executable$ são usados para declarar predicados abdutíveis e ações executáveis, respectivamente. A declaração de predicados abdutíveis é importante para que o planejador saiba que predicados podem ser adicionados como hipóteses ao resíduo abdutivo. Já a declaração de ações executáveis permite que o planejador seja capaz de distinguir uma ação primitiva de uma ação composta, ou seja, uma ação que é definida em termos de outras ações mais simples. Como o uso de ações compostas somente faz sentido em planejadores hierárquicos (*vide* subseção 6.2.1), e em nossos experimentos não estamos considerando nenhum planejador desse tipo, vamos assumir que todas as ações descritas na base de conhecimento são executáveis. Dessa forma, não teremos mais necessidade do metapredicado $executable$. Ademais, como no raciocínio abdutivo para planejamento de ordem parcial apenas os predicados *happens* e $before$ podem ser abduzidos, também não teremos mais necessidade do metapredicado *abducible*. Assim, diferentemente do AECP, o ABP não utilizará esses metapredicados.

M4: Restrições de codesignação

Na implementação do AECP, o procedimento de unificação "nativo" do PROLOG é utilizado como um método de adição de restrições de codesignação (*i.e.* instanciação de variáveis) ao plano. Isso permite que a descrição de ações nesse planejador contenha variáveis. Por outro lado, para que os planejadores baseados em STRIPS pudessem

[1] A base de conhecimento do PROLOG é organizada como uma tabela de *hashing*.

aceitar operadores contendo variáveis, um procedimento específico para codesignação de variáveis teria que ser implementado [61]. Então, para facilitar a comparação, implementamos o ABP como uma versão proposicional do AECP (como é feito na maioria das análises de desempenho encontradas na literatura de planejamento [31, 3, 27, 30, 28]). Como veremos, entretanto, essa mudança afeta a checagem de consistência do resíduo negativo, permitindo que ela seja mais eficiente no ABP do que no AECP.

M5: Consistência do resíduo negativo

No AECP, a consistência do resíduo negativo deve ser checada toda vez que o resíduo abdutivo é modificado. Por exemplo, para satisfazer a submeta $holdsAt(at(there), t)$, o AECP poderia postular uma ocorrência da ação $walk(X, there)$, num instante de tempo t' anterior a t. Para tanto, os literais $happens(walk(X, there), t')$ e $before(t', t)$ teriam que ser incluídos no resíduo abdutivo e o literal $clipped(t', at(there), t)$ teria que ser incluído no resíduo negativo. Então, para garantir a consistência dessas inclusões, o AECP teria que garantir que o novo evento postulado não afeta a validade de nenhum literal *clipped* previamente estabelecido (através de negação por falha), bem como que a validade do novo literal *clipped*, recém-estabelecido, não seja violada por nenhum evento previamente postulado. Isso é necessário porque, no AECP, a reutilização de passos (eventos) parcialmente instanciados pode causar novas codesignações de variáveis, fazendo com que um passo já existente no resíduo abdutivo passe a ameaçar vínculos causais já estabelecidos.

Cada literal $clipped(t_1, f, t_2)$ existente no resíduo negativo equivale a um vínculo causal[2] $t_1 \rightarrow f@t_2$ e a checagem de consistência desse resíduo equivale ao tratamento de conflitos no planejamento de ordem parcial. Então, no AECP, se o plano contém $|\mathcal{S}|$ passos e $|\mathcal{L}|$ vínculos, temos que checar $|\mathcal{S}| \times |\mathcal{L}|$ possíveis ameaças. Logo, como $|\mathcal{L}|$ pode ser expresso em função de $|\mathcal{S}|$, segue que a consistência do resíduo negativo no AECP consome tempo $O(|\mathcal{S}|^2)$.

No caso do ABP, entretanto, como ele é proposicional, não há possibilidade de que a reutilização de um passo existente no resíduo abdutivo faça surgir uma ameça entre passos e vínculos pré-estabelecidos. Então, quando um passo existente no plano

[2]Note que cada ocorrência de ação no cálculo de eventos é identificada univocamente por uma marca de tempo t_i. Assim, há um isomorfismo entre marcas de tempo e rótulos de passos usados, respectivamente, no cálculo de eventos e na representação STRIPS.

é selecionado para suportar uma submeta, apenas o novo literal *clipped* adicionado precisa ser protegido. Caso um novo passo seja adicionado, temos que checar apenas a consistência do literal *clipped* recém adicionado, com relação às ações postuladas no resíduo abdutivo, e dos literais *clipped*'s já provados, com relação ao passo recém adicionado. Assim, diferentemente do que ocorre no AECP, no ABP o tratamento de conflitos é incremental e tem complexidade $O(|\mathcal{S}|)$.

M6: Planejamento abdutivo sistemático e redundante

A partir do ABP, resultante das modificações descritas até aqui, implementamos mais dois sistemas de planejamento:

- SABP: versão sistemática do ABP;
- RABP: versão redundante do SABP.

A versão sistemática (*vide* subseção 3.4.1), que denominamos SABP, foi obtida modificando-se o axioma [SEC3] do cálculo de eventos simplificado, de modo que um fluente F fosse considerado ameaçado dentro de um intervalo de tempo $[T_1, T2]$ – $clipped(T_1, F, T_2)$ – não apenas quando da ocorrência de uma ação que termina sua validade nesse intervalo, mas também quando da ocorrência de uma ação que a inicia.

$$clipped(T_1, F, T_2) \leftrightarrow \qquad \text{[SEC3']}$$
$$\exists A, T[happens(A, T) \wedge T_1 \prec T \wedge T \prec T_2 \wedge$$
$$(terminates(A, F, T) \vee initiates(A, F, T))]$$

Para obtermos a versão redundante (*vide* subseção 3.4.2), que denominamos RABP, nenhuma modificação precisou ser feita na axiomatização do cálculo de eventos simplificado proposto. De fato, a única mudança feita foi com relação à estratégia de seleção de submetas. No ABP, assim como no SABP, submetas são selecionadas e então removidas da lista de submetas a serem atingidas. Isso é possível porque esses planejadores coletam literais *clipped* (que correspondem a vínculos causais) cuja validade é constantemente checada (o que corresponde a uma estratégia de proteção de submetas já atingidas). Entretanto, como nenhuma política de proteção é adotada no planejamento redundante (ou seja, literais *clipped* não são coletados), a implementação do RABP requer uma pequena modificação no meta-interpretador abdutivo.

Para implementar no RABP uma estratégia de seleção de submetas equivalente àquela utilizada no TWEAK, baseada no MTC, usamos a idéia de *projeção temporal*: fazemos o meta-interpretador executar o plano correntemente em construção, sem permitir que qualquer modificação seja feita nele, e selecionamos a primeira submeta não necessariamente verdadeira encontrada. Em outras palavras, selecionamos uma submeta ϕ tal que o literal $holdsAt(\phi, t)$ seja possivelmente falso no instante t, posterior ao instante em que a última ação no plano é executada.

Note que, como não há proteção de submetas no planejamento redundante, uma submeta já estabelecida pode ser violada e ter que ser restabelecida futuramente. Conseqüentemente, as submetas selecionadas para estabelecimento no RABP não podem ser removidas da lista de submetas a serem atingidas. Ademais, por não coletar literais *clipped* provados através de negação por falha, o RABP não precisa checar a consistência do resíduo negativo toda vez que o resíduo abdutivo é modificado (*vide* subseção 4.1.2). Em compensação, a cada refinamento do plano, o RABP precisa checar novamente a validade de cada submeta já estabelecida e, caso alguma delas tenha sido violada (em conseqüência da falta de proteção aos literais *clipped*), ele deverá restabelecê-la novamente.

Assim como ocorre no TWEAK, é possível que no RABP uma mesma submeta tenha que ser estabelecida diversas vezes e que, por outro lado, a adição de um único passo satisfaça diversas submetas, simultaneamente.

5.2 Experimento I: correspondência entre o POP e o ABP

Através desse experimento, corroboramos a conjectura de *Shanahan* [57], que afirma que *o planejamento de ordem parcial como busca no espaço de planos é isomorfo ao planejamento abdutivo no cálculo de eventos*, e também mostramos que, usando abdução e cálculo de eventos, um sistema de planejamento lógico pode ser tão eficiente quanto um sistema de planejamento algorítmico baseado em STRIPS.

5.2.1 Domínios de teste

Para obter problemas de planejamento dentro das diferentes classificações apresentadas na subseção 3.1.3, *Barret & Weld* [3] criaram uma família de domínios artificiais denominada $D^m S^n$. Nessa nomenclatura, a parte D^m indica que cada operador do domínio

remove precondições de m outros operadores e a parte S^n, que n passos são necessários para atingir uma submeta particular. A tabela 5.2 especifica alguns domínios dessa família.

domínio	*esquema de operador*
D^0S^1	$oper(act:a_i, pre:\{p_i\}, add:\{g_i\}, del:\{\})$
D^1S^1	$oper(act:a_i, pre:\{p_i\}, add:\{g_i\}, del:\{p_{i-1}\})$
D^mS^1	$oper(act:a_i, pre:\{p_i\}, add:\{g_i\}, del:\{p_j \mid j<i\})$
D^mS^2	$oper(act:a_i^1, pre:\{p_i\}, add:\{q_i\}, del:\{\})$ $oper(act:a_i^2, pre:\{q_i\}, add:\{g_i\}, del:\{p_j \mid j<i\})$

Tabela 5.2: *Domínios artificiais propostos por Barret & Weld.*

No domínio D^0S^1, cada operador a_i adiciona uma submeta g_i se sua precondição p_i está satisfeita. Como os operadores têm efeitos positivos exclusivos, e não há efeitos negativos, os problemas nesse domínio são *independentes*. Conseqüentemente, tanto a ordem em que as submetas são estabelecidas quanto a ordem em que os passos do plano são executados é irrelevante para o desempenho do sistema de planejamento. Ademais, como nesse domínio não há nenhum tipo de interação entre operadores, *o tempo gasto pelo planejador é totalmente dedicado ao estabelecimento de submetas.*

Os domínios D^1S^1 e D^mS^1, com poucos e muitos efeitos negativos, respectivamente, propiciam problemas *serializáveis*. Esses efeitos negativos fazem com que os passos em cada domínio interajam uns com os outros, forçando o planejador a tratar os conflitos resultantes dessas interações. No domínio D^1S^1, além de adicionar uma submeta g_i quando sua precondição p_i está satisfeita, cada operador a_i também remove a precondição p_{i-1} do operador a_{i-1}. O domínio D^mS^1 é muito semelhante ao domínio D^1S^1; exceto pelo fato que cada operador a_i remove a precondição do operador a_j, para $j<i$.

Finalmente, em D^mS^2 temos problemas *não-serializáveis*. Nesse domínio, criado a partir de D^mS^1, cada submeta precisa de dois passos para ser atingida: o primeiro deles, a_i^1, requer p_i como precondição e produz q_i como efeito; o segundo, a_i^2, requer q_i como precondição e produz g_i como efeito. Além disso, cada operador a_i^2 remove a precondição p_j do operador a_j^1, para $j<i$. Nesse domínio, como há muita interação entre operadores, *o tempo gasto pelo planejador é dominado pelo tratamento de conflitos.*

5.2.2 Metodologia

Para avaliar o desempenho relativo dos planejadores POP e ABP, utilizamos os domínios artificiais da família D^mS^n, que compreendem problemas de diversas complexidades. Com isso, visamos garantir que os resultados empíricos obtidos fossem independentes das idiossincrasias de um domínio particular.

Para cada um dos domínios considerados (D^0S^1, D^1S^1, D^mS^1 e D^mS^2), foram definidos 6 operadores[3] e foram gerados 30 problemas distintos, cada um deles consistindo de uma permutação aleatória de 6 precondições iniciais $(p_1, \ldots, p_6)$ e de um arranjo simples de 1 a 6 submetas $(g_1, \ldots, g_6)$, selecionadas aleatoriamente.

Para cada tamanho de problema (número de submetas), foram geradas 5 instâncias distintas, que foram resolvidas por cada um dos planejadores. Então, a partir dos resultados obtidos com esses testes, calculamos o tempo médio gasto por cada planejador para resolver instâncias de um mesmo tamanho. Com esse cuidado em gerar problemas aleatórios e calcular tempos médios, visamos garantir que o desempenho dos planejadores não fosse influenciado, de maneira tendenciosa, pela ordem específica das submetas em cada problema de planejamento, para cada tamanho de instância.

5.2.3 Testes realizados

Usando os domínios artificiais da família D^mS^n, realizamos dois testes:

- primeiro, observamos como o tamanho do espaço de busca explorado pelos sistemas aumenta à medida em que aumentamos o número de submetas nos problemas;
- depois, observamos como o tempo médio de CPU consumido pelos sistemas aumenta à medida em que aumentamos o número de submetas nos problemas.

Verificação do tamanho do espaço de busca

Conforme vimos na subseção 3.4.3 – *análise do fator de ramificação* – um novo nó é gerado e visitado toda vez que o plano em construção é refinado, ou seja, sempre que um novo pré-requisito é satisfeito ou que um conflito é resolvido. Sabemos que a satisfação de um pré-requisito $\phi@\alpha_c$ por um passo α_p requer a adição da restrição $\alpha_p \prec \alpha_c$ ao conjunto de restrições de ordem causal $\mathcal{O}$ do plano em construção (*vide* tabela 3.3 – linha

[3] Exceto no caso do domínio D^mS^2, para o qual foram definidos 6 pares de operadores.

3.1). Da mesma forma, para resolver um conflito entre um passo α_t e um vínculo causal $\alpha_p \rightarrow \phi@\alpha_c$, precisamos antecipar o passo ameaçador, adicionando $\alpha_t \prec \alpha_p$ a $\mathcal{O}$, ou então postergá-lo, adicionando $\alpha_c \prec \alpha_t$ a $\mathcal{O}$ (*vide* tabela 3.3 – linhas 4.1 e 4.2). Portanto, o número de nós visitados durante a busca corresponde diretamente ao número vezes que uma restrição de ordem temporal é adicionada ao plano em construção.

Sendo assim, para determinar o tamanho do espaço de busca explorado pelo POP, alteramos sua implementação de modo que um contador fosse incrementado sempre que uma restrição de ordem temporal fosse adicionada ao conjunto $\mathcal{O}$. Analogamente, no meta-interpretador abdutivo, um novo nó foi contabilizado a cada vez que um literal $before$ foi adicionado ao resíduo abdutivo.

Os resultados dos testes realizados com esses sistemas, devidamente alterados para determinar o número de nós visitados durante a busca, são apresentados na figura 5.1.

Verificação do tempo médio de CPU

Para garantir que a comparação entres os desempenhos dos sistemas POP e APB fosse realmente justa, tentamos implementá-los da maneira mais equivalente possível, sempre que isso não implicasse em contrariar a abordagem em cada um deles (algorítmica ou lógica). Entre os cuidados que tomamos nesse sentido, além daqueles já descritos na seção 5.1, destacamos os seguintes:

- Usamos a mesma linguagem de programação em ambas as implementações.
- Representamos os componentes do plano (*vide* tabela 5.3) de maneira equivalente e empregamos as mesmas estruturas de dados (termos e listas) para agrupá-los.
- Usamos os mesmos recursos de programação (*e.g.* recursividade, retrocesso, corte, *etc.*) ao implementar procedimentos equivalentes nos dois sistemas (*e.g.* refinamento do plano, escolha de ações, tratamento de conflitos, *etc.*).
- Compartilhamos o mesmo procedimento de cômputo do fecho transitivo da relação de ordem temporal, necessário para consistência de planos nos dois sistemas.

Os tempos gastos pelos planejadores para solucionar problemas nos domínios da família $D^m S^n$ foram obtidos através do uso do predicado `cputime`, nativo do compilador SWI-PROLOG. De posse desses valores, calculamos o tempo médio de CPU consumido em cada sistema de planejamento, para cada tamanho de instância. Os resultados obtidos dessa maneira são apresentados na figura 5.2.

componente	POP	ABP
passo	$step(\tau_1, \alpha)$	$happens(\alpha_1, \tau)$
ordem temporal	$\tau_1 < \tau_2$	$before(\tau_1, \tau_2)$
vínculo causal	$link(\tau_1, \phi, \tau_2)$	$clipped(\tau_1, \phi, \tau_2)$

Tabela 5.3: *Correspondência entre os componentes do plano no* POP *e no* ABP

5.2.4 Análise dos resultados

Nessa subseção, analisamos os resultados obtidos com os testes realizados com problemas nos domínios artificiais da família $D^m S^n$.

Equivalência entre os métodos e abordagens

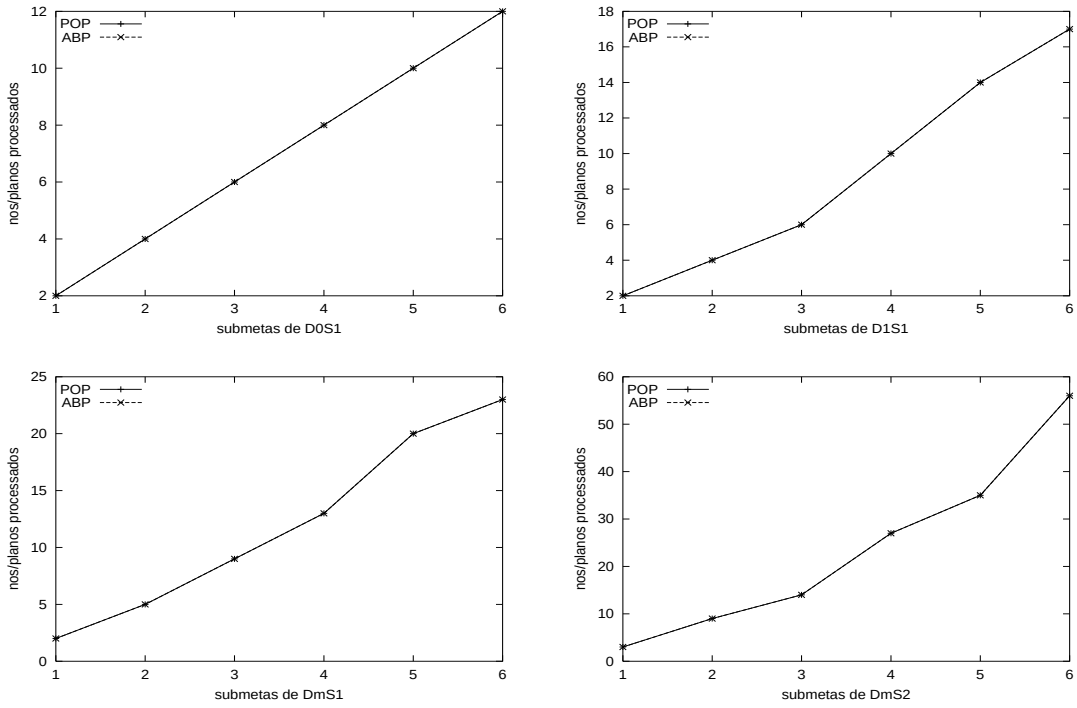


Figura 5.1: *Espaço de busca para testes em domínios da família $D^m S^n$.*

A figura 5.1 apresenta curvas que mostram como cresce o número de nós processados pelos planejadores POP e APB, em função do número de submetas, para problemas nos

diversos domínios considerados: D^0S^1 (esquerda superior), D^1S^1 (direita superior), D^mS^1 (esquerda inferior) e D^mS^2 (direita inferior).

Como podemos observar, em cada um desses domínios, as curvas obtidas para os sistemas comparados estão sobrepostas. Isso significa que os planejadores POP e ABP geram e exploram espaços de busca de tamanhos idênticos, para problemas idênticos, a despeito da complexidade desses problemas.

Um aspecto importante, que não pode ser observado através das curvas apresentadas na figura 5.1, é que a equivalência entre os espaços de busca explorados pelos planejadores não é apenas quantitativa, mas também qualitativa. Ou seja, eles não apenas visitam o mesmo número de nós, mas *visitam os mesmos nós, na mesma ordem.* Esse fato foi constatado a partir da análise das soluções correspondentes, encontradas pelos planejadores, para cada um dos 120 problemas resolvidos no teste realizado.

Para tornar essa idéia mais clara, vamos tomar como exemplo os planos[4] a seguir, encontrados como solução para o problema da anomalia de *Sussman*, respectivamente, pelo POP e pelo ABP:

```
plan([step(42,unstack(c,a)), step(40,stack(b,c)), step(18,stack(a,b))],
     [42<40,...], [link(i,on(c,a),42), ...])

res([happens(unstack(c,a),42), happens(stack(b,c),40), happens(stack(a,b),18)],
    [before(42,40),...], [clipped(0,on(c,a),42),...])
```

Examinando esses planos, (*vide* correpondência estabelecida na tabela 5.3) percebemos que o POP rotula passos de forma completamente análoga àquela em que o ABP *skolemiza* variáveis temporais nas ocorrências de eventos abduzidas. (Note que antes de encontrar o primeiro passo pertencente à solução do problema, `step(18,stack(a,b))`, o POP fez 17 tentativas fracassadas. Equivalentemente, o sistema ABP fez o mesmo número de tentativas antes de abduzir com sucesso o literal `happens(stack(a,b),18)`.) De fato, o rastreamento da execução dos sistemas mostra que eles geram e visitam os mesmos nós, na mesma ordem.

Esse resultado confirma, claramente, a conjectura de que planejamento de ordem parcial como busca no espaço de planos é isomorfo ao raciocínio abdutivo no cálculo de eventos. Como vimos na subseção 4.4.4, *Shanahan* [57] estabeleceu essa conjectura

[4] As partes omitidas, embora idênticas, não são relevantes à nossa discussão.

a partir de uma simples inspeção do código do meta-interpretador abdutivo, que foi especializado para o cálculo de eventos, tentando identificar os passos do algoritmo de planejamento de ordem parcial. Agora, com esse experimento, comprovamos empiricamente que tal correspondência realmente existe e que os métodos de planejamento em cada uma das abordagens são completamente equivalentes.

Desempenho dos planejadores

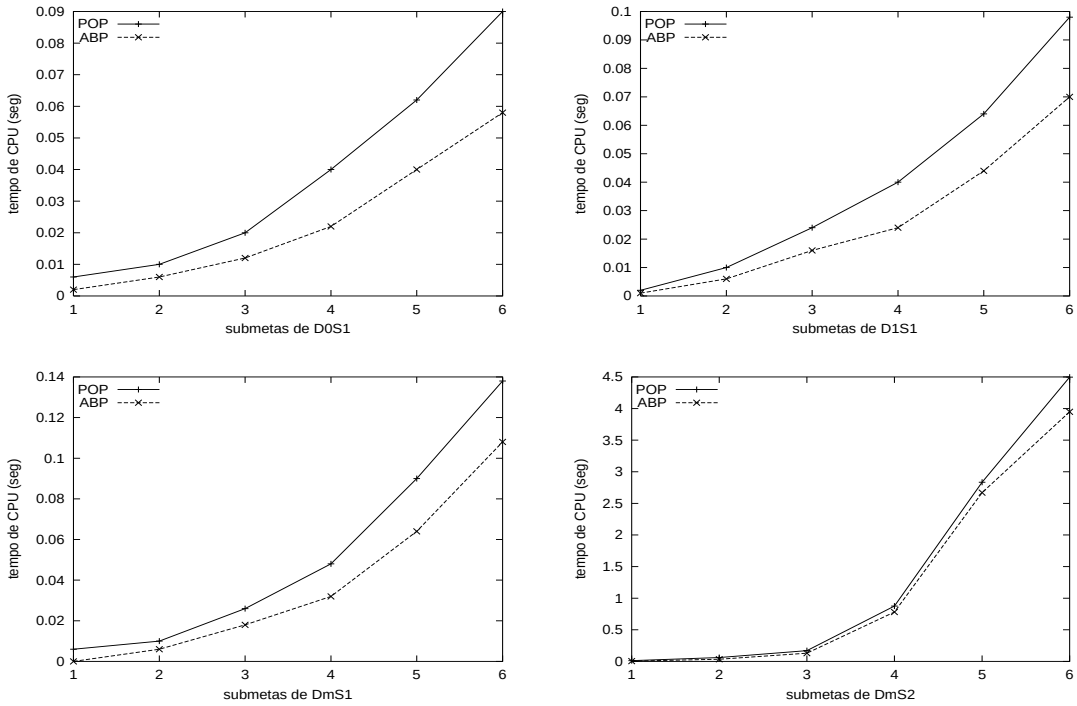


Figura 5.2: *Consumo de CPU para testes em domínios da família* $D^m S^n$.

A figura 5.2 mostra o tempo médio de CPU gasto para resolver problemas nos diversos domínios da família $D^m S^n$. Como podemos observar, as curvas que representam o tempo gasto pelo planejador ABP estão muito próximas àquelas que representam o tempo gasto pelo planejador POP. Esse resultado confere ainda mais credibilidade à nossa comprovação empírica da conjectura de *Shanahan* [57]. A partir dele, podemos concluir que planejamento de ordem parcial e raciocínio abdutivo no cálculo de eventos são equivalentes não apenas conceitualmente, enquanto métodos de planejamento, mas

também em termos de desempenho, enquanto sistemas implementados.

Na verdade, em todos os domínios considerados nos testes realizados, o sistema ABP apresentou um desempenho ligeiramente melhor do que aquele observado para o sistema POP. Então, como esses sistemas implementam métodos equivalentes (conforme constatamos na análise anterior), essa diferença de desempenho pode ser justificada pela representação de ações empregada, que difere de uma abordagem para outra.

No cálculo de eventos (*vide* subseção 4.2.2), os efeitos das ações são descritos através dos predicados *initiates* e *terminates*, o que nos permite encontrar diretamente as ações que realizam ou que ameaçam a realização de uma determinada submeta. Por exemplo, para encontrar ações que produzem a submeta g como efeito, basta fazer a seguinte consulta:

```
?- initiates(A,g,T).
```

Para determinar que ações impedem a realização da submeta g, podemos fazer:

```
?- terminates(A,g,T).
```

Por outro lado, como na representação STRIPS os efeitos das ações são descritos através das listas *add* e *del*, para encontrar as ações que adicionam ou removem uma determinada submeta, o sistema tem que examinar essas listas, para cada operador existente no domínio. Por exemplo, com a consulta a seguir, podemos encontrar ações que produzem g como efeito (positivo ou negativo):

```
?- oper(Act,Pre,Add,Del), (member(g,Add) ; member(g,Del)).
```

Note que, diferentemente do que ocorre na consulta correspondente no cálculo de eventos, nessa consulta, a busca não é orientada pela submeta g. Para encontrar as ações de interesse, o sistema tem que recuperar a descrição de cada operador na base de conhecimento e, usando o predicado `member`, nativo do compilador SWI-PROLOG, verificar se a submeta desejada consta de uma de suas listas de efeitos.

Assim, acreditamos que o melhor desempenho do sistema ABP, em comparação ao POP, seja devido à sua representação de ações, baseada no cálculo de eventos, que possibilita uma busca mais eficiente. Essa justificativa explica também por quê a diferença entre os tempos consumidos pelos dois sistemas, para os mesmos problemas, se acentua à medida em que os domínios ficam mais complexos, conforme podemos observar na figura 5.3.

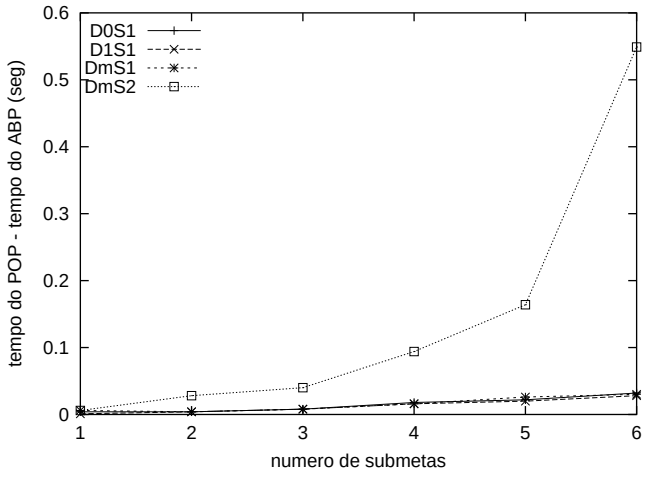


Figura 5.3: *Diferença entre tempos consumidos pelo* POP *e pelo* ABP*, em cada domínio.*

Nos domínios mais simples (*e.g.* D^0S^1), onde há pouca interação entre operadores, o tempo gasto pelos planejadores é praticamente todo dedicado ao estabelecimento de submetas. Então, quando o número de submetas estabelecidas é pequeno, a vantagem da representação de ações do cálculo de eventos não se destaca muito. Porém, em domínios mais complexos (*e.g.* D^mS^2), onde há muita interação entre operadores, o tempo consumido pelos planejadores é dominado pelo tratamento de conflitos. Nesse caso, como o tratamento de conflitos demanda muitas consultas à representação de ações, a diferença entre os desempenhos dos sistemas é acentuada (veja a figura 5.3). Enquanto essa diferença é da ordem de centésimos de segundos para problemas de menor complexidade (independentes e serializáveis), para problemas de maior complexidade (não-serializáveis), ela chega a ser da ordem de décimos de segundo.

5.3 Experimento II: sistematicidade *versus* redundância

Nesse experimento, mostramos que as versões sistemática e redundante do planejador abdutivo (SABP e RABP - *vide* página 90), apresentam o mesmo comportamento dos respectivos planejadores algorítmicos correspondentes (SNLP e TWEAK). Com isso, estendemos os resultados do experimento anterior e mostramos que o isomorfismo existente entre planejamento de ordem parcial e raciocínio abdutivo no cálculo de eventos

pode ser preservado para métodos de planejamento sistemáticos e redundantes, em ambas as abordagens.

Através desse mesmo experimento, também corroboramos a conjectura de *Knoblock & Yang* [30], segundo a qual *o desempenho de um planejador sistemático ou redundante está fortemente relacionado à razão entre o número de ameaças positivas e negativas no domínio considerado*, e mostramos que ela vale também para os sistemas de planejamento abdutivo baseado em cálculo de eventos.

5.3.1 Domínios de teste

Para que fosse possível controlar precisamente a razão entre o número de ameaças positivas e negativas existentes entre os passos de um plano, *Knoblock & Yang* [30] projetaram uma família de domínios artificiais denominada ART-$\#_{est}$-$\#_{clob}$. Nessa nomenclatura, $\#_{est}$ indica o número de ameaças positivas (*establishers*) e $\#_{clob}$ indica o número de ameaças negativas (*clobbers*).

ART-$\#_{est}$-$\#_{clob}$
$oper(act : a_i^1,$ $\quad pre : \{p_i\},$ $\quad add : \{q_i, p_{i+1}$ if $i < \#_{est}\},$ $\quad del : \{p_{i-1}$ if $0 < i \leq \#_{clob}\})$ $oper(act : a_i^2,$ $\quad pre : \{q_i\},$ $\quad add : \{g_i, q_{i+1}$ if $i < \#_{est}\},$ $\quad del : \{q_{i-1}$ if $0 < i \leq \#_{clob}\})$

Tabela 5.4: *Domínio artificial proposto por Knoblock & Yang.*

Nesses domínios (*vide* tabela 5.4), cada submeta g_i é atingida por um sub-plano composto por dois passos ordenados: o primeiro deles, a_i^1, exige p_i como precondição e produz q_i como efeito; o segundo, a_i^2, exige q_i como precondição e produz g_i como efeito. Ademais, cada um dos $\#_{est}$ primeiros pares de operadores adiciona as precondições para o par seguinte e, similarmente, cada um dos $\#_{clob}$ primeiros pares de operadores remove as precondições para o par anterior. Assim, através desses efeitos colaterais (*vide* figura[5]

[5]Nesse diagrama, círculos representam precondições e retângulos representam operadores; arestas

5.4), poderíamos então controlar a relação entre os fatores $\#_{est}$ e $\#_{clob}$.

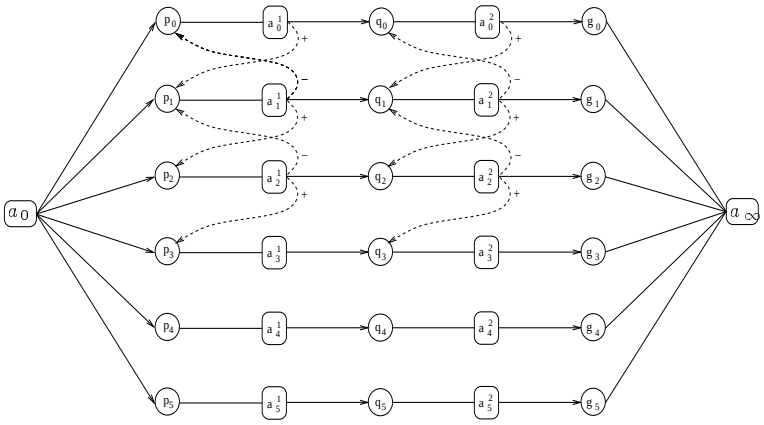


Figura 5.4: *Interações entre os operadores do domínio* ART-3_{est}-2_{clob}.

Realizando testes preliminares com esses domínios, entretanto, verificamos que na prática as ameaças positivas nunca surgem durante a construção de um plano. Esse fato pode ser constatado a partir da análise dos resultados apresentados na figura 5.5.

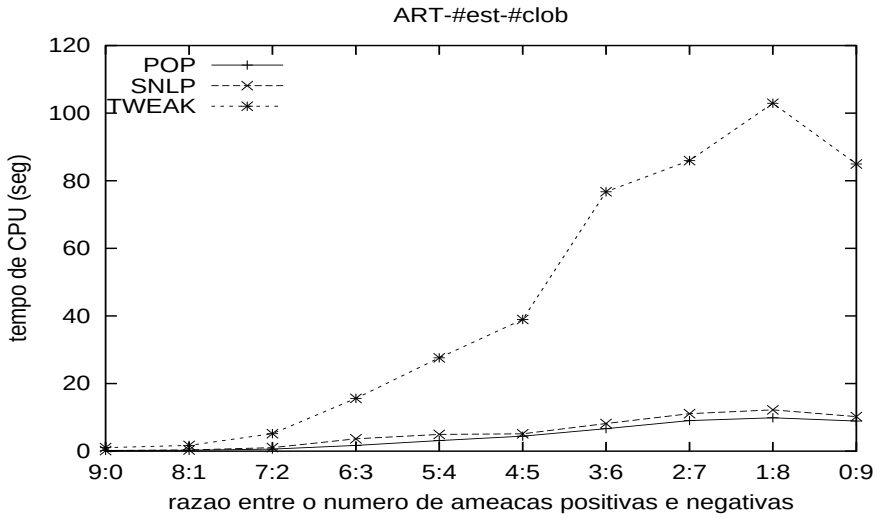


Figura 5.5: *Comportamento dos planejadores nos domínios* ART-$\#_{est}$-$\#_{clob}$.

associam precondições a operadores e arcos associam operadores a efeitos. Os rótulos + e − denotam, respectivamente, efeitos colaterais positivos e negativos. a_0 e a_∞ são passos virtuais (*vide* página 43).

Observe que o planejador sistemático SNLP tem um comportamento praticamente idêntico àquele do POP. Porém, de acordo com a análise apresentada na página 62, em domínios onde o número de ameças positivas é muito maior que o número de ameças negativas (*e.g.* $9:0$, $8:1$, *etc.*), o SNLP deveria consumir muito mais tempo do que o POP. Investigando o motivo dessa discrepância, percebemos que isso acontece porque nossos planejadores (que usam busca em profundidade iterativa) encontram sempre planos minimais como solução. Sendo assim, se há dois ou mais operadores distintos que realizam a mesma submeta, apenas um deles será escolhido para compor o plano.

Por exemplo, considere o domínio ART-3_{est}-2_{clob}, cujas interações estão esquematizadas na figura 5.4, e suponha um problema cujo estado inicial é especificado pelo conjunto $\{p_0, \ldots, p_5\}$ e cuja meta é especificada pelo conjunto $\{g_0, \ldots, g_5\}$. Evidentemente, para que a submeta g_0 seja atingida, o passo a_0^2 terá que ser adicionado ao plano. Então, como a_0^2 também produz como efeito colateral a precondição q_1 para o passo a_1^2 (que atinge g_1), o passo a_1^1, cujo efeito principal é q_1, não será mais necessário. Analogamente, uma vez que o passo a_1^2 seja adicionado ao plano, o passo a_2^1 poderá ser descartado. Esse efeito "dominó" será observado até que não haja mais operadores distintos para uma mesma submeta. Em conseqüência disso, podemos concluir também que o número de passos nos planos deve aumentar à medida em que o número de ameaças positivas decrescer: quanto menos operadores com efeitos redundantes tivermos, menos operadores poderão ser descartados na construção dos planos.

A nova família de domínios artificiais proposta

Ao contrário do que esperávamos, o uso da família ART-$\#_{est}$-$\#_{clob}$ não nos permite controlar precisamente a razão entre ameaças positivas e negativas. Sendo assim, para que pudéssemos comprovar a conjectura de *Knoblock & Yang* [30], criamos uma nova família de domínios artificiais que denominamos[6] $A^x D^y S^2$. Conforme veremos, com essa nova família podemos, de fato, controlar precisamente a relação entre o número de ameaças positivas x (**A***dd*) e negativas y (**D***el*) em cada domínio. Diferentemente do que ocorre na família ART-$\#_{est}\#_{clob}$, na família $A^x D^y S^2$ proposta:

- há ameaças positivas ainda que os planos construídos sejam minimais e
- os planos têm um número constante de passos para problemas de tamanho fixo.

[6] Preferimos seguir a nomenclatura criada por *Barret & Weld* (*vide* página 91).

Similarmente aos domínios da família ART-$\#_{est}\#_{clob}$, num domínio $A^x D^y S^2$, cada submeta g_i pode ser atingida por um sub-plano composto por dois passos ordenados: o primeiro deles, a_i^1, tem p_i como precondição e produz q_i^1 e q_i^2 como efeitos; o segundo passo, a_i^2, tem q_i^1 e q_i^2 como precondições o e produz g_i como efeito (*vide* tabela 5.5).

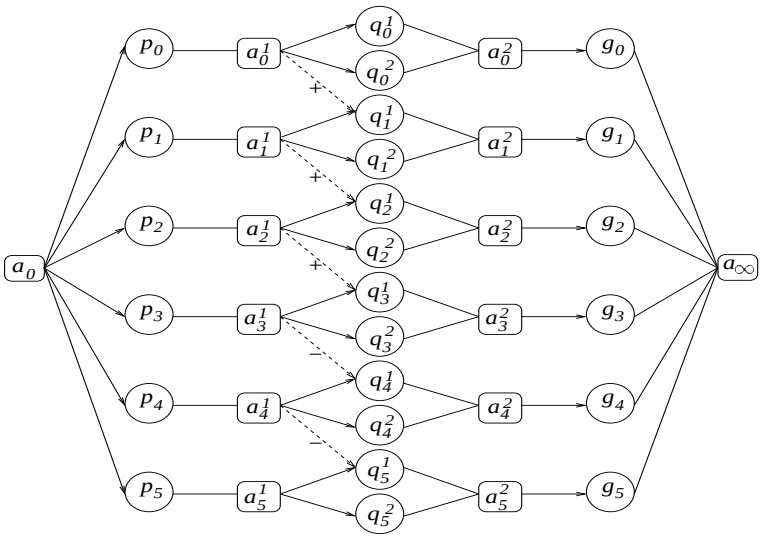


Figura 5.6: *Interações entre os operadores do domínio* $A^3 D^2 S^2$.

A novidade é que, num domínio da família $A^x D^y S^2$ composto por $n = x + y + 1$ operadores, os x primeiros operadores a_i^1 adicionam, e os próximos y operadores removem, a precondição q_{i+1} para o operador o a_{i+1}^2. Como um operador a_i^1 fornece como efeito colateral apenas a metade das precondições necessárias ao operador a_{i+1}^2 (veja a figura 5.6), toda submeta deverá ser atingida por um sub-plano com dois passos exclusivos (daí o sufixo S^2 no nome da família). Assim, podemos garantir que, independentemente do domínio escolhido em $A^x D^y S^2$, sempre que tivermos um problema que inclui todas as submetas do domínio, teremos também planos que incluem todos os operadores do domínio. Portanto, usando a nova família que propomos, garantimos que a razão entre o número de ameaças positivas e negativas poderá ser, de fato, precisamente controlada.

domínio	*esquemas de operadores*
$A^x D^y S^2$	$oper(act: a_i^1, pre: \{p_i\}, add: \{q_i^1, q_i^2, q_{i+1} \text{ if } i < x\}, del: \{q_{i+1} \text{ se } i \geq x\})$ $oper(act: a_i^2, pre: \{q_i^1, q_i^2\}, add: \{g_i\}, del: \{\})$

Tabela 5.5: *Novo domínio artificial proposto para esse experimento.*

5.3.2 Metodologia

Para observarmos de que forma os planejadores comparados se comportam à medida que variamos os domínios, ou seja, à medida que variamos a razão entre o número de ameaças positivas e negativas, mantivemos fixo o número de submetas nos problemas resolvidos. Em conseqüência disso, e das características dos domínios $A^x D^y S^2$ utilizados, o número de passos em todos os planos encontrados foi sempre o mesmo.

Para gerar os domínios de teste, variamos o número de ameaças positivas x e negativas y, de tal forma que o número total de ameaças $x+y$ se mantivesse constante. Mais especificamente, definimos 10 pares de operadores distintos e, então, simultaneamente, aumentamos o número de interações negativas entre eles à medida em que diminuímos o número de interações positivas. Dessa forma, obtivemos 10 domínios distintos ($A^9 D^0 S^2$, $A^8 D^1 S^2$, ..., $A^1 D^8 S^2$, $A^0 D^9 S^2$).

Também geramos 10 problemas distintos, cada um deles consistindo de uma permutação aleatória de 10 condições iniciais $(p_0, \ldots, p_9)$ e de uma permutação aleatória de 10 submetas $(g_0, \ldots, g_9)$. Esses mesmos problemas foram resolvidos por cada um dos sistemas comparados, em cada um dos diferentes domínios considerados.

Com esses cuidados, visamos garantir que os diferentes comportamentos observados fossem decorrentes exclusivamente dos diferentes métodos de planejamento implementados pelos sistemas comparados (sistemático ou redundante) e da maneira como a razão entre os números de ameaças positivas e negativas influencia esses métodos.

5.3.3 Testes realizados

Nos testes realizados com os domínios artificiais da família $A^x D^y S^2$, verificamos como o tempo de CPU consumido pelos sistemas comparados varia em função da razão $x:y$ (entre o número de ameaças positivas x e o número de ameaças negativas y) e do método de planejamento implementado (sistemático ou redundante). Os resultados

obtidos são apresentados na figura 5.7 e a sua confiabilidade é estabelecida segundo as mesmas bases já discutidas em nosso primeiro experimento (*vide* subseção 5.2.3).

5.3.4 Análise dos resultados

Nessa subseção, analisamos os resultados obtidos com os testes realizados com problemas nos domínios artificiais da família $A^x D^y S^2$.

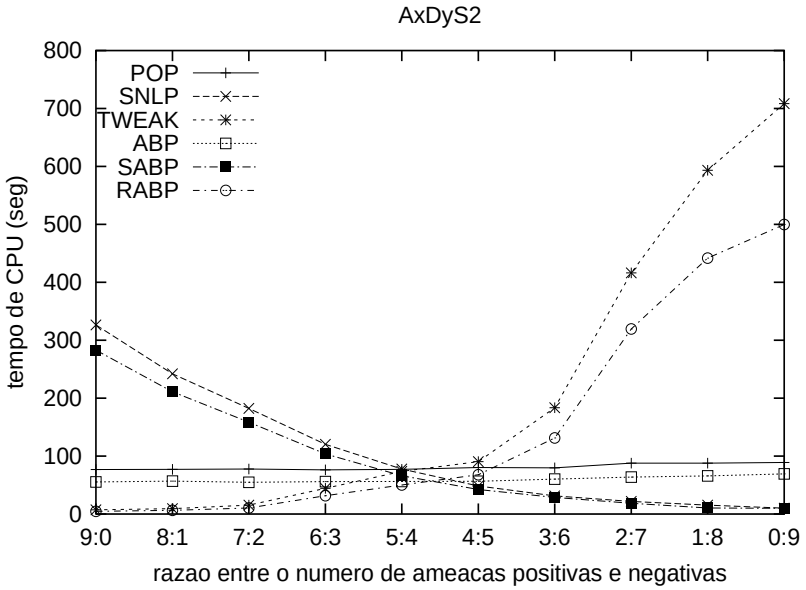


Figura 5.7: *Consumo de CPU para testes em domínios da família* $A^x D^y S^2$.

Correspondência entre comportamentos

Como podemos observar na figura 5.7, as curvas que representam o tempo gasto em sistemas que implementam métodos de planejamento correspondentes são muito similares; exceto pelo fato de que, como no Experimento I, o tempo gasto pelos planejadores lógicos é sempre um pouco menor do que aquele gasto pelos planejadores algorítmicos correspondentes. Como já foi explicado (*vide* página 98), entretanto, essa diferença de desempenho é devida à representação de ações no cálculo de eventos, que possibilita buscas mais eficientes na base de conhecimento.

Assim, a partir desse resultado, podemos inferir que a equivalência existente entre

planejamento de ordem parcial (POP) e raciocínio abdutivo no cálculo de eventos (ABP) pode ser mantida também entre as versões sistemáticas (SNLP e SABP) e redundantes (TWEAK e RABP) correspondentes, dentro de cada abordagem.

Influência da razão $x:y$ no comportamento dos sistemas

Conforme havíamos previsto (*vide* subseção 3.4.4 - página 62), nossos testes comprovam que quando o número de ameaças positivas é muito maior do que o número de ameças negativas (*hipótese 1*), o TWEAK é o planejador com melhor desempenho e o SNLP é aquele com pior desempenho. Por outro lado, quando o número de ameaças negativas é muito maior do que o número de ameças positivas (*hipótese 2*), a situação se inverte: enquanto o SNLP é o sistema mais eficiente, o TWEAK é aquele que se sai pior.

Como podemos observar na figura 5.7, o planejador POP nunca tem o melhor desempenho; porém, ao contrário do SNLP e do TWEAK, cujos comportamentos variam em função da razão entre o número de ameças positivas e negativas, seu comportamento se mantém estável em todos os domínios.

Esse resultado confirma a conjectura estabelecida por *Knoblock & Yang* [30], que afirma que o desempenho de um sistema de planejamento depende não apenas da política de proteção adotada por ele (sistemática ou redundante), mas também das características do domínio considerado [30]. Ademais, como os planejadores abdutivos apresentam o mesmo comportamento dos planejadores algorítmicos testados, podemos inferir que essa conjectura vale também para sistemas de planejamento lógico.

Calibragem dos sistemas

Em vista dos resultados obtidos nesse experimento, acreditamos não ser possível implementar um sistema de planejamento baseado num único método (sistemático ou redundante), de tal modo que ele tenha sempre o melhor desempenho. Porém, seria possível que um sistema de planejamento "compilasse" os operadores de um domínio particular e determinasse, automaticamente, a razão entre o número de ameças positivas e negativas nesse domínio; então, com base nessa medida, o sistema poderia calibrar-se de modo a obter o melhor desempenho possível. No caso dos planejadores abdutivos, isso pode ser feito da seguinte maneira:

- Para problemas onde o número de ameaças negativas é muito maior do que

o número de ameças positivas, utilize o método de planejamento sistemático (SABP); ou seja, substitua o axioma [SEC3] pelo axioma [SEC3'], conforme mostramos na página 90.

- Para problemas onde o número de ameaças positivas é muito maior do que o número de ameças positivas, utilize o método de planejamento redundante (RABP); ou seja, substitua a estratégia de seleção de pré-requisitos baseada em fila por aquela baseada em projeção temporal, conforme discutimos na página 91.

- Para problemas onde o número de ameaças positivas é muito próximo ao número de ameaças negativas, utilize o método de planejamento padrão (ABP).

5.4 Considerações finais

Nesse capítulo, mostramos como implementar em PROLOG três planejadores lógicos que empregam *abdução* como regra de inferência e *cálculo de eventos circunscritivo* como formalismo para representar ações e raciocinar sobre seus efeitos: o ABP, sua versão sistemática, que denominamos SABP, e sua versão redundante, que denominamos RABP.

Usando esses planejadores lógicos e mais três planejadores algorítmicos baseados em STRIPS, que também implementamos em PROLOG, reproduzimos alguns resultados importantes da literatura de análise comparativa de sistemas de planejamento [31, 3, 27, 30, 28]. Com isso, mostramos que há uma estreita correspondência entre planejamento de ordem parcial (POP, SNLP e TWEAK) e raciocínio abdutivo no cálculo de eventos (ABP, SABP e RABP). Essa correspondência foi estabelecida tanto em termos de desempenho (consumo de CPU), quanto em termos de políticas de proteção e respectivos comportamentos de busca (sistemático ou redundante).

Finalmente, mostramos que as características específicas de um domínio particular, sobretudo a razão entre o número de ameaças positivas e negativas entre os operadores desse domínio, podem influenciar bastante no comportamento de um sistema de planejamento. Assim, concluímos que um sistema de planejamento deve ser calibrado, se desejamos que seu desempenho seja sempre o melhor possível. Isso pode ser feito avaliando-se o domínio de planejamento de interesse e selecionando-se um método de planejamento adequado para esse domínio, de acordo com as suas características.

Capítulo 6

Conclusão

Planos não substituem ações. (Mas,
garantem que elas sejam efetivas.)

Provérbio chinês (adaptado)

Nesse trabalho, apresentamos as principais idéias da área de planejamento em IA, tanto dentro da abordagem lógica (dedutiva e abdutiva), *i.e.* sistemas que sintetizam planos como efeito colateral da prova de teoremas, quanto dentro da abordagem algorítmica.

Com base no sistema de planejamento AECP (**A***bductive* **E***vent* **C***alculus* **P***lanner*), proposto por *Shanahan* [57], e nas idéias provenientes de diversos trabalhos de análise conceitual da tarefa de planejamento, especialmente [4, 5, 6], implementamos um sistema de planejamento lógico que emprega abdução como regra de inferência e cálculo de eventos como formalismo para descrever ações e raciocinar sobre seus efeitos. Esse sistema, que denominamos ABP (**A***bductive* **P***lanner*), difere do AECP, principalmente, por implementar do cálculo de eventos apenas aspectos relevantes ao planejamento clássico; ou seja, *tempo atômico*, *efeitos determinísticos* e *onisciência*. A partir do sistema ABP, implementamos outros dois sistemas: um sistemático, que denominamos SABP, e um redundante, que denominamos RABP.

A fim de analisar o desempenho da abordagem de planejamento abdutiva (*i.e.* dos sistemas ABP, SABP e RABP), implementamos três planejadores algorítmicos clássicos correspondentes (POP, SNLP e TWEAK). As implementações foram feitas de modo que todos os planejadores compartilhassem, sempre que possível, as mesmas estruturas de dados e os mesmos procedimentos básicos. A equivalência entre essas implementações foi demonstrada através de experimentos onde pudemos observar que, em

ambas as abordagens, sistemas (lógicos e algorítmicos) que implementam estratégias de planejamento correspondentes apresentam comportamentos idênticos. Esses experimentos também mostraram que a eficiência dos planejadores lógicos, assim como dos planejadores algorítmicos, não depende apenas da política de proteção de submetas adotada em cada um deles, mas também das características específicas do domínio de planejamento considerado; sobretudo, da relação existente entre o número de ameaças positivas e negativas existentes nesse domínio. Nossos experimentos reproduzem para os planejadores lógicos importantes resultados encontrados na literatura de análise de desempenho comparativa entre sistemas de planejamento [31, 3, 27, 30, 28].

Conforme mostramos, os planejadores algorítmicos POP, SNLP e TWEAK e suas correspondentes versões lógicas ABP, SABP e RABP consomem aproximadamente o mesmo tempo de CPU para problemas idênticos. Isso indica que, em termos de desempenho, não há vantagem significativa dos planejadores de ordem parcial avaliados com relação aos planejadores abdutivos baseados em cálculo de eventos. Por outro lado, além de serem mais facilmente estendidos, mantidos ou modificados, os planejadores lógicos têm a vantagem de nos permitir empregar uma linguagem mais expressiva para representação de conhecimento e de produzir sistemas inteligentes onde a capacidade de planejar possa ser facilmente integrada a outras habilidades que também dependem de raciocínio lógico (*e.g.* compreensão de linguagem natural, aprendizado de máquina, raciocínio espacial, controle de sistemas dinâmicos, *etc.*).

Apesar de em nossos experimentos não termos comparado os planejadores lógicos aos mais recentes e eficientes planejadores propostos na literatura de planejamento em IA (*e.g.* SATPLAN [29] e GRAPHPLAN [7]), acreditamos que os resultados obtidos representam um importante passo na direção de uma boa solução teórica para a tarefa de planejamento.

6.1 Principais contribuições

A principal contribuição desse trabalho é a comprovação empírica de que um planejador baseado em prova de teoremas pode ter comportamento e desempenho similares àqueles observados para aos planejadores algorítmicos baseados em STRIPS, sem adicionar qualquer complexidade extra na solução de problemas.

Outras contribuições importantes do nosso trabalho são as seguintes:

- Acrescentamos à literatura em português uma revisão dos principais conceitos da área de planejamento em IA, especialmente sobre planejamento abdutivo.
- Coletamos um extenso material bibliográfico sobre planejamento dedutivo, abdutivo e algorítmico.
- Implementamos em PROLOG três sistemas de planejamento algorítmico clássicos (POP, SNLP e TWEAK) e suas respectivas versões lógicas baseadas em abdução no cálculo de eventos (ABP, SABP e RABP). Pelo fato desses sistemas terem sido implementados numa mesma linguagem e compartilharem estruturas de dados e procedimentos comuns, eles proporcionam uma boa base para futuras extensões do nosso trabalho.
- Demonstramos, empiricamente, que raciocínio abdutivo no cálculo de eventos e planejamento de ordem parcial são isomorfos; ou seja, que os sistemas implementados segundo essas abordagens realizam os mesmos passos e exploram o mesmo espaço de busca quando os problemas resolvidos são idênticos.
- Provamos que um sistema de planejamento lógico, baseado em abdução e cálculo de eventos, pode ser tão eficiente quanto alguns sistemas de planejamento algorítmicos clássicos da literatura de IA.
- Mostramos como um planejador abdutivo baseado em cálculo de eventos pode ser modificado de modo a implementar métodos de planejamento sistemático e redundante, do mesmo modo que sistemas de planejamento algorítmicos clássicos.
- Projetamos uma família de domínios artificiais ($A^x D^y S^2$), através da qual podemos controlar precisamente a razão entre o número de ameaças positivas e negativas existentes entre os operadores de um determinado domínio.
- Estabelecemos uma correspondência entre sistemas de planejamento sistemático e redundante clássicos e planejamento abdutivo baseado em cálculo de eventos.
- Comprovamos que, como nos sistemas algorítmicos, nos sistemas lógicos a eficiência depende fortemente das características do domínio considerado.
- Mostramos como um sistema de planejamento abdutivo baseado em cálculo de eventos pode ser calibrado, de acordo com as características do domínio, de modo a ter sempre o melhor desempenho possível.

6.2 Trabalhos futuros

Entre as diversas extensões possíveis do nosso trabalho, *planejamento hierárquico* e *programação de agentes robóticos* são duas extensões especialmente interessantes.

6.2.1 Planejamento hierárquico

Planejamento hierárquico ou HTN (**H***ierarchical* **T***ask* **N***etwork*) [16] é um paradigma que contribui para a redução do espaço de busca, baseando-se na idéia de decomposição de tarefas em outras tarefas mais simples.

Em HTN, uma *tarefa primitiva* é uma tarefa que pode ser diretamente realizada, executando-se uma ação correspondente, enquanto uma *tarefa composta* é uma tarefa cuja realização requer a execução de várias outras tarefas (compostas ou primitivas).

Em STRIPS, tarefas primitivas podem ser representadas por *operadores*, enquanto tarefas compostas podem ser representada por *métodos*, *i.e.* sub-planos que implementam operadores abstratos. O exemplo a seguir, baseado em *Russel & Norvig* [53], ilustra um método para construir uma casa.

$$
\begin{aligned}
&method(task : construct(house),\\
&\quad steps : \{\alpha_1 : build(foundation), \alpha_2 : build(frame), \alpha_3 : build(roof),\\
&\qquad \alpha_4 : build(walls), \alpha_5 : build(interior)\},\\
&\quad order : \{\alpha_1 \prec \alpha_2, \alpha_2 \prec \alpha_3, \alpha_2 \prec \alpha_4, \alpha_3 \prec \alpha_5, \alpha_4 \prec \alpha_5\},\\
&\quad links : \{\alpha_1 \rightarrow foundation@\alpha_2, \alpha_2 \rightarrow frame@\alpha_3, \alpha_2 \rightarrow frame@\alpha_4,\\
&\qquad \alpha_3 \rightarrow roof@\alpha_5, \alpha_3 \rightarrow walls@\alpha_5\})
\end{aligned}
$$

Dados um conjunto de operadores $\mathcal{A}$, especificando as precondições e os efeitos de cada tarefa primitiva, um conjunto de métodos $\mathcal{M}$, estabelecendo como realizar tarefas compostas, e uma tarefa inicial, representando o problema de planejamento a ser resolvido, um planejador HTN procede decompondo tarefas compostas em tarefas mais simples, e resolvendo os conflitos que surgem entre elas, até que um plano constituído exclusivamente por tarefas primitivas seja obtido.

Um planejador abdutivo hierárquico

No cálculo de eventos, tarefas compostas podem ser naturalmente definidas em termos de outras mais simples. Por exemplo, o método $construct(house)$ poderia ser especificado em cálculo de eventos da seguinte maneira:

$$
\begin{aligned}
&happens(construct(house), T_1, T_5) \leftarrow \\
&\quad happens(build(foundation), T_1) \wedge happens(build(frame), T_2) \wedge \\
&\quad happens(build(roof), T_3) \wedge happens(build(walls), T_4) \wedge \\
&\quad happens(build(interior), T_5) \wedge \\
&\quad (T_1 \prec T_2) \wedge (T_2 \prec T_3) \wedge (T_2 \prec T_4) \wedge (T_3 \prec T_5) \wedge (T_4 \prec T_5) \wedge \\
&\quad \neg clipped(T_1, foundation, T_2) \wedge \neg clipped(T_2, frame, T_3) \wedge \neg clipped(T_2, frame, T_4) \wedge \\
&\quad \neg clipped(T_3, roof, T_5) \wedge \neg clipped(T_3, walls, T_5)
\end{aligned}
$$

Assim, para implementarmos um planejador abdutivo hierárquico, poucas alterações seriam necessárias no meta-interpretador abdutivo. A versão binária de *happens* poderia ser empregada para denotar tarefas primitivas, enquanto a versão ternária denotaria tarefas compostas. Assim, ao encontrar um literal $happens(C, T_1, T_2)$, denotando uma tarefa composta C, o meta-interpretador o expandiria num sub-plano correspondente (possivelmente contendo outras tarefas compostas). Por outro lado, ao encontrar um literal $happens(P, T)$, denotando uma tarefa primitiva P, o meta-interpretador simplesmente o adicionaria ao resíduo abdutivo.

Planejamento hierárquico *versus* Planejamento baseado em estados

Uma das principais diferenças entre planejamento hierárquico e planejamento baseado em estados está no objetivo do planejamento. Planejadores baseados em estados visam *metas de satisfação*, ou seja, condições que devem ser satisfeitas num determinado estado do mundo. Qualquer plano que torne essas condições verdadeiras no estado final é considerado uma solução válida, não importando que ações ou estados intermediários ele contenha. Planejadores hierárquicos, entretanto, visam *metas de realização*, ou seja, tarefas que devem ser realizadas. Por exemplo, *"construir uma casa"* é uma meta de realização, enquanto *"ter uma casa"* é uma meta de obtenção; note que *"comprar uma casa"* satisfaz essa última, mas não a primeira.

De fato, metas de realização proporcionam um incremento na expressividade da representação. Por exemplo, *"ir a Nova York e voltar"* não pode ser expressa diretamente como uma meta de satisfação, já que os estados inicial e final são o mesmo [16]. Além disso, metas de realização reduzem o espaço de busca na medida em que a especificação de métodos restringe a atenção do planejador a expansões desejáveis de uma particular tarefa, em vez de permitir que a busca explore todas as seqüências de ações possíveis.

6.2.2 Uma linguagem para programação de agentes robóticos

Ao projetarmos uma linguagem para programação de agentes, devemos levar em conta os diferentes tipos de agentes que podem ser programados; desde aqueles puramente *deliberativos*, que planejam todas as suas ações, até aqueles puramente *reativos*, que simplesmente reagem aos estímulos que recebem de seu ambiente.

Na verdade, esses tipos extremos de agentes correspondem diretamente a dois estilos de representação de conhecimento, também extremos: o puramente *declarativo*, no qual descrevemos o "que" constitui o problema a ser resolvido pelo agente, e o puramente *operacional*, no qual descrevemos "como" o agente deve resolver um determinado problema. Enquanto o estilo puramente declarativo exige do agente grande capacidade de raciocínio e tomada de decisão, o estilo puramente operacional requer apenas que o agente seja capaz de executar uma seqüência de ações previamente programada (*i.e.* o programa de controle do agente).

Assim, agentes deliberativos têm como vantagem o fato de representarem um paradigma para resolução de problemas muito mais flexível (não-determinístico); por outro lado, agentes reativos têm como vantagem o fato de representarem um paradigma de resolução de problemas muito mais eficiente (determinístico). Na prática, entretanto, o ideal é manter um compromisso entre flexibilidade e eficiência, criando agentes que não apenas sejam capazes de planejar, mas que também sejam capazes de agir.

A linguagem GOLOG

O GOLOG (Alg**o**l in **Log**ic) [35], criado pelo *Grupo de Robótica Cognitiva* da Universidade de Toronto, é uma linguagem de programação lógica, baseada no cálculo de situações, especialmente projetada para a programação de agentes. Como uma linguagem de programação estruturada, GOLOG oferece estruturas de controle tais como

seqüência, *decisão* e *repetição* (*vide* tabela 6.1). Entretanto, diferentemente de programas escritos em linguagens de programação convencionais, ao serem executados, programas em GOLOG são decompostos em primitivas que denotam ações a serem executadas pelo agente. Ademais, como tais primitivas são formuladas através de axiomas do cálculo de situações, é possível raciocinar logicamente sobre seus efeitos.

De fato, GOLOG é uma tentativa de combinar ambos os estilos de representação de conhecimento (declarativo e operacional) numa mesma linguagem de programação, permitindo ao programador cobrir todo um espectro de possibilidades entre os extremos puramente deliberativo e puramente reativo.

```
:- op(950,xfy,[&]).  % seqüencia
:- op(500,xfy,[?]).  % teste (projeção temporal)
:- op(960,xfy,[|]).  % escolha não-determinística
:- op(960,xfy,[~]).  % negação por falha

exec(A1 & A2,S1,S3) :- exec(A1,S1,S2), exec(A2,S2,S3).
exec(P?,S,S):- holds(P,S).
exec(A1 | A2,S1,S2) :- exec(A1,S1,S2); exec(A2,S1,S2).
exec(if(P,A1,A2),S1,S2) :- exec(P? & A1 | ~P? & A2,S1,S2).
exec(star(E),S1,S2) :- S1=S2; copy_term(E,E1), exec(E & star(E1),S1,S2).
exec(while(P,A),S1,S2) :- copy_term(P,P1), exec(star(P? & A) & ~P1?,S1,S2).
exec(A,S1,S2) :- proc(A,A1), exec(A1,S1,S2).
exec(A,S,do(A,S)) :- prim(A), poss(A,S).

holds(A=A,_).
holds(~P,S) :- not holds(P,S).
```

Tabela 6.1: *Implementação simplificada de um interpretador* GOLOG *em* PROLOG.

Programas GOLOG são executados através de prova de teorema. O usuário fornece uma axiomatização $\mathcal{A}$, descrevendo as ações do agente no domínio (*conhecimento declarativo*), bem como um programa de controle c, especificando o comportamento esperado desse agente (*conhecimento operacional*). A partir disso, executar o programa equivale a provar que existe uma situação σ tal que $\mathcal{A} \models exec(c, s_0, \sigma)$. Então, se a situação σ encontrada pelo provador de teoremas é um termo da forma $do(a_n, do(\ldots, do(a_1, s_0)))$, a seqüência de ações $\langle a_1, \ldots, a_n \rangle$ correspondente é executada pelo agente.

Por exemplo, considere a axiomatização a seguir que descreve as ações de um agente que controla um elevador e a situação inicial do seu mundo:

```
holds(curfloor(4),s0).
holds(on(3),s0).
holds(on(5),s0).

poss(open,_).
poss(close,_).
poss(up(N),S) :- holds(curfloor(C),S), C<N.
poss(down(N),S) :- holds(curfloor(C),S), C>N.
poss(turnoff(N),S) :- holds(on(N),S).

holds(curfloor(N),do(up(N),S)).
holds(curfloor(N),do(down(N),S)).
holds(P,do(A,S)) :- holds(P,S), not affects(A,P).

affects(up(N),curfloor(M)) :- N\=M.
affects(down(N),curfloor(M)) :- N\=M.
affects(turnoff(N),on(N)).

prim(open).
prim(close).
prim(up(_)).
prim(down(_)).
prim(turnoff(_)).
```

Nesse domínio, o agente deve atender às chamadas[1] que são feitas pelos usuários, indicadas através do fluente $on(n)$. Esse comportamento desejado pode ser especificado pelo seguinte programa GOLOG:

```
proc(control,
  while(on(N), serve(N)) &
  park).

proc(serve(N),
  curfloor(C)? &
  if(C=N,
    open & turnoff(N) & close,
    up(N) | down(N))).

proc(park,
   curfloor(C)? &
   if(C=0,
     open,
     down(0) & open)).
```

[1] Não há distinção entre chamadas originadas externa ou internamente ao elevador.

Uma vez fornecidos a axiomatização do domínio e o programa de controle, podemos executar o interpretador GOLOG da seguinte maneira:

```
?- exec(control,s0,S).
S = do(open, do(down(0), do(close, do(turnoff(5), do(open, do(up(5), do(close,
do(turnoff(3), do(open, do(down(3), s0)))))))))
```

Proposta de uma nova linguagem de controle baseada em cálculo de eventos

Uma idéia interessante seria implementar uma linguagem similar ao GOLOG, que também empregasse símbolos extra-lógicos adicionais (*e.g.* `if`, `while`, *etc.*) para representar ações complexas, mas que usasse cálculo de eventos, em vez de cálculo de situações, para descrever ações primitivas. Além disso, usaríamos também um provador de teoremas que fosse capaz de realizar abdução.

Desta forma, em vez dos planos de ordem total produzidos com o GOLOG, a execução dos programas escritos nessa nova linguagem produziria planos de ordem parcial; tornando, assim, muito mais fácil integrar ação e deliberação.

Para entender melhor essa idéia, considere o seguinte cenário: *O elevador encontra-se parado no 5º andar e duas chamadas são originadas, respectivamente, no 9º e no 2º andares. Então, o agente formula um plano para atender a essas chamadas e iniciaria sua execução, subindo em direção ao 9º andar. No caminho, porém, uma nova chamada é originada no 8º andar. Em conseqüência desse evento, o agente reage "consertando" seu plano de execução, de modo que essa nova chamada seja também atendida.* Claramente, se o plano construído pelo agente for de ordem parcial, a adição dos passos necessários para atender à nova chamada será muito mais simples.

Um interpretador para a linguagem que propomos poderia ser obtido adaptando-se aquele apresentado para o GOLOG (*vide* tabela 6.1), de modo que tanto a projeção temporal quanto a satisfação de submetas fossem realizadas por um planejador abdutivo. Dessa forma, além de tornar a tarefa de replanejamento mais fácil, poderíamos também calibrar a linguagem para adotar um método de planejamento sistemático ou redundante, de acordo com as características do domínio considerado. Ademais, devido ao uso do cálculo de eventos, a linguagem poderia incorporar noções mais complexas tais como ações com duração no tempo e paralelismo.

6.2.3 Outras extensões

Outras extensões interessantes seriam as seguintes:

- comparar o planejador abdutivo ao UCPOP [48], um planejador cuja representação de ações – o ADL[46] – estende a expressividade do STRIPS, possibilitando o uso de precondições disjuntivas e efeitos condicionais;
- verificar como o sistema de planejamento abdutivo pode ser modificado para implementar *multi-contribuidores*, planejamento condicional, planejamento com consumo de recursos e planejamento com mudança contínua.

Referências Bibliográficas

[1] Apt, K. R. *Logic Programming*, In Handbook of Theoretical Computer Science, J. van Leeuwen, Elsevier Science Publishers B.V., pages 493-573, 1990.

[2] Bacchus, F. *The Role of Logic in Planning*, In Electronic Newsletter on Reasoning About Actions and Change (http://www.ida.liu.se/ext/etai/rac/nl00/005/sframe.html), Ed. Erik Sandewall, Issue 00005, 2000.

[3] Barrett, A. and Weld, D. *Partial Order Planning: Evaluating Possible Efficiency Gains*, Dept. of Computer Science and Engineering, University of Washigton, Technical Report 92-05-01, July, 1992.

[4] Barros, L. N., Valente, A. and Benjamins, V. R. *Modeling Planning Tasks.* In proceedings of: Third International Conference on Artificial Intelligence Planning Systems, AIPS'96, pages 11-18, 1996.

[5] Barros, L. N., Hendler, J. and Benjamins, V. R. *Par-KAP: a Knowledge Acquisition Tool for Building Practical Planning Systems.* In proceddings of: The Fifteenth International Joint Conference on Artificial Intelligence, IJCAI, 1997.

[6] Barros, L. N., and Santos, P. E. *The Nature of Knowledge in an Abductive Event Calculus Planner*, Proc. of 12th International Conference EKAW 2000, pages 328-343, 2000.

[7] Blum, A. and Furst, M. *Fast Planning Through Planning Graph Analysis*, Artificial Intelligence, 90, pages 281-300, 1997.

[8] Chapman, D. *Planning for Conjunctive Goals*, Artificial Intelligence, 32(3), pages 333-377, 1987.

[9] Chittaro, D. and Montanari, A. *Efficient Temporal Reasoning in the Cached Event Calculus*, Computacional Intelligence 12 (3), pages 359-382, 1996.

[10] Clark, K. *Negation as Failure*, In Herve Gallaire and Jack Minker, editors, Logic and Data Bases, pages 293-322, Plenum Press, NY, 1978.

[11] Colmerauer at al. *Un Système de Comunication Homme-machine en Français*, Technical Report, Groupe de Recherche en Intelligence Artificielle, Université d´Aix Marseille, pages 203-212, 1973.

[12] Cox, P. T., and Pietrzykowsky, T. *Causes for Events: Their Computation and Applications*, Proc. 8th International Conference on Automated Deduction, CADE'86, pages 608-621, 1986.

[13] Currie, K. and Tate, A. *O-Plan: the Open Planning Architecture*, Artificial Intelligence Vol. 52, pages 49-86, 1991.

[14] Denecker, M. and De Schreye, D. *SLDNFA: An Abductive Procedure for Normal Abductive Programs*, In Proceedings of the International Joint Conference and Symposium on Logic Program, K. R. Apt, ed., pages 686-700, 1992.

[15] Denecker, M., Missiaen, L. and Bruynooghe, M. *Temporal Reasoning with Abductive Event Calculus*, In Proceedings of the European Conference on Artificial Intelligence, pages 384-388, 1992.

[16] Erol, K. *Hierarchical Task network Planning: Formalization, Analisys and Implementation.* PhD Thesis, University of Maryland, 1995.

[17] Eshghi, K. *Abductive Planning with Event Calculus*, In Proceedings of the Fifth International Conference on Logic Programming, pages 562-579, 1988.

[18] Fikes, R. E., and Nilsson, N. J. *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving*, Artificial Intelligence, vol. 2 (3/4), pages 189-208, 1971.

[19] Finger, J. J. and Genesereth, M. R. *Residue: A Deductive Approach to Design Synthesis.*, Technical Report STAN-CS-85-1035, Department of Computer Science, Stanford University, Stanford, CA, 1985.

[20] FINGER, J. J. *Exploiting Constraints in Design Synthesis.*, PhD Thesis, Stanford University, Stanford, CA, 1987.

[21] GENESERETH, M.R. AND NILSSON, N. *Logical Foundations of Artificial Intelligence.*, Morgan Kaufmann, 1987.

[22] GEORGEFF, M. P. *Planning*, In Allen, J., Hendler, J. and Tate, A. (eds), Readings in Planning, Morgan Kaufmann, 1990.

[23] GREEN, C. *Application of Theorem Proving to Problem Solving*, Proc. IJCAI-69, pages 219-240, 1969.

[24] HANKS, S. AND MCDERMOTT, D. *Default Reasoning, Nonmonotonic Logic, and the Frame Problem*, Proc. AAAI'86, pages 328-333, 1986.

[25] HARVEY, W. D., GINSBERG, M. L. AND SMITH, D. E. *Deferring Conflict Resolution Retains Systematicity*, Proc. AAAI'93.

[26] KAKAS, A. C., KOWALSKI, R. A., AND TONI, F. *Abductive Logic Programming*, Journal of Logic and Computation, vol. 2, (6):719-770, 1993.

[27] KAMBHAMPATI, S. *Multi-contributor Causal Structures for Planning: A Formalization and Evaluation*, Artificial Intelligence, vol. 69, pages 235-278, 1992.

[28] KAMBHAMPATI, S. *Planning as Refinement Search: A Unified Framework for Evaluation Design Tradeoffs in Partial Order Planning*, Artificial Intelligence, vol. 76, pages 167-238, 1995.

[29] KAUTZ, H. AND SEL-MAN, B. *Planning as Satisfiability*, In Proc. of ECAI-92, pages 359-379, 1992.

[30] KNOBLOCK, C. A. AND YANG, Q. *Evaluating the Tradeoffs in Partial-Order Planning Algorithms*, In Proceedings of the Canadian Artificial Intelligence Conference, 1994.

[31] KORF, R. E. *Planning as Search: A Quantitative Approach*, Artificial Intelligence, vol. 33, pages 65-88, 1987.

[32] KORF, R. E. *Search: A Survey of Recent Results*, In H. Shrobe, editor, Exploring Artificial Intelligence, pages 197-237. Morgan Kaufmann, San Mateo, CA, 1988.

[33] KORF, R. E. *Linear-space Best-first Search: Summary of Results*, In Proc. 10th Nat. Conf. on A.I., pages 533-538, July 1992.

[34] KOWALSKI, R. A., AND SERGOT, M. J. *A Logic-based Calculus of Events*, New Generation Computing 4, pages 67-95, 1986.

[35] LEVESQUE, H. J., REITER, R., LESPÉRANCE, Y., LIN, F. AND SCHERL, R. B. *GOLOG: A Logic Programming Language for Dynamic Domains*, In Journal of Logic Programming, 31, pages 59-84, 1997.

[36] LEVI, G. AND RAMUNDO, D. *A formalization of metaprogramming for real*, In D. S. Warren, ed., Logic Programming, Proc. of the 10th International Conference on Logic Programming, pages 354-373, MIT Press, 1993.

[37] LIFSCHITZ, V. *On the Semantics of* STRIPS, In James Allen, James Hendler, and Austin Tate, editors, Readings in Planning, pages 523-531. Morgan kaufman, 1990.

[38] LIFSCHITZ, V. *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, 1994.

[39] MACALLESTER, D., AND ROSENBLITT, D. *Systematic Nonlinear Planning*, In Proc. 9th Nat. Conf. on A.I., pages 634-639, July 1991.

[40] MCCARTHY, J. *Situations, Actions and Causal Laws*, Memo 2, Stanford University Artificial Intelligence Project, Stanford, California, 1963.

[41] MCCARTHY, J. *Epistemological Problems of Artificial Intelligence*, In Proceedings of IJCAI-77, pages 1038-1044, 1977.

[42] MCCARTHY, J. *Applications of Circumscription to Formalizing Common Sense Knowledge*, Artificial Intelligence, 26(3):89-116, 1986.

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

[44] MCDERMOTT, D. V. *Regression Planning*, Int. Journal of Intelligent Systems, 6, pages 357-416, 1991.

[45] Nilsson, N. J. *Principles of Artificial Intelligence*, Tioga, Palo Alto, 1980.

[46] Pednault, E. P. D. *Formulating Multiagent, Dynamic-world Problems in the Classical Planning Framework*, In Georgeff, M. P. and Lansky, A. L., editors, Reasoning About Actions and Plans: Proceedings of the 1986 Workshop, pages 47-82, Timberline, Oregon. Morgan kaufmann, 1986.

[47] Peirce, C. S. *Collected Papers of Charles Sanders Peirce*, vol. 2, 1931-1958, Hartshorn et al., editors. Harvard University Press.

[48] Penberthy, J. S., and Weld, D. S. *UCPOP: A Sound, Complete, Partial Order Planner for ADL*, In Proceedings of KR-92, pages 103-114, 1992.

[49] Puget, J. F. *Apprentissage par Explications d'Échecs; Théories et Applications*, Ph.D. Thesis, LRI, 1990.

[50] Reiter, R. *On Closed Word Data Bases*, In H. Gallaire and J. Minker, Logic and Data Bases, pages 55-76, Plenum Press, NY, 1978.

[51] Robinson, J. A. *A Machine-oriented Logic based on the Resolution Principle*, Journal of the ACM, 12:23-41, 1965.

[52] Russell, S. *Efficient Memory Bounded Search Algorithms*, In Proceedings of the Tenth Conference on Artificial Intelligence, Vienna, Wiley, 1992.

[53] Russell, S., and Norvig, P. *Artificial Intelligence - A Modern Approach*, Prentice-Hall, 1995.

[54] Sacerdoti, E. *The Nonlinear Nature of Plans*, In Proceedings of IJCAI-75, pages 206-214, 1975.

[55] Shanahan, M. P. *Prediction is Deduction but Explanation is Abduction*, In Proceedings IJCAI-89, pages 1055-1060, 1989.

[56] Shanahan, M. P. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*, MIT Press, Cambridge, 1997.

[57] Shanahan, M. P. *An Abductive Event Calculus Planner*, The Journal of Logic Programming, 44:207-239, 2000.

[58] Sterling, L. and Shapiro, E. *The Art of Prolog*, MIT Press, 1986.

[59] Tate, A. *Generating Projects Networks*, In Proceedings of 5th International Joint Conference on Artificial Intelligence, pages 888-893, 1977.

[60] Waldinger, R. *Achieving Several Goals Simultaneously*, In Machine Intelligence 8, Hellis Horwood Limited, Chichester, 1977.

[61] Weld, D. S. *An Introduction to Least Commitement Planning*, AI Magazine, 15(4), pages 27-61, Winter 1994.

Apêndice A

O planejador abdutivo ABP

O programa a seguir, codificado em SWI-PROLOG, implementa o planejador abdutivo ABP, com base no cálculo de eventos simplificado (para o planejamento clássico). Essa versão difere daquela usada em nossos experimentos apenas com relação às modificações que foram necessárias para contabilizar o espaço de busca explorado, bem como para realizar outras estatísticas (*e.g.* número de conflitos resolvidos) que foram analisadas durante o desenvolvimento desse trabalho.

```
abp(Problem) :-
   % consulta arquivo contendo a especificação do problema
   consult(Problem),
   % obtem a lista de submetas do problema
   goals(Goals),
   % inicia o contador de profundidade iterativa
   flag(depth,_,0),
   % guarda o tempo de CPU antes de iniciar a busca da solução
   Before is cputime,
   % repete a busca em profundidade iterativa
   repeat,
     % inicia o contador de marcas de tempo (para skolemização)
     flag(time,_,1),
     % incrementa o contador de profundidade iterativa
     flag(depth,D,D+1),
     % calcula profundidade corrente da busca
     Depth is D+1,
     % exibe a profundidade corrente da busca
     format('~nDepth:  ~d',Depth),
     % tenta solucionar o problema
     abp(Goals,res([],[],[]),res(H,B,N)),
   % ate encontrar um plano com no maximo Depth passos
```

```
   !,
   % obtem tempo de CPU após terminar a busca da solução
   After is cputime,
   % calcula o tempo consumido na busca
   Time is After-Before,
   % exibe o tempo consumido na busca
   format('~n~nCPU time:  ~2f sec',Time),
   % exibe uma linearização (ordenação topológica) para o plano de ordem parcial encontrado
   format('~n~nPlan:'),
   linearisation(H,B).

% resolve o problema vazio trivialmente

abp([],R,R).

% implementa o axioma [SEC1] do calculo de eventos simplificado,
% ou seja, holdsAt(F,T) ← initially(F) ∧ ¬clipped(0,F,T)

abp([holdsAt(F,T)|Gs],res(Hi,Bi,Ni),R) :-
   % verifica se o fluente F vale inicialmente
   initially(F),
   % e se não deixa de valer dentro do intervalo [0,T]
   naf(clipped(0,F,T),Hi,Bi,Bf),
   % coleta no resíduo negativo o literal provado através de negação por falha (naf)
   add(clipped(0,F,T),Ni,Nf),
   % continua tentando satisfazer demais submetas do problema
   abp(Gs,res(Hi,Bf,Nf),R).

% implementa o axioma [SEC2] do calculo de eventos simplificado, ou seja,
% holdsAt(F,T) ← happens(A,T1) ∧ initiates(A,F,T1) ∧ (T1 ≺ T) ∧ ¬clipped(T1,F,T)

abp([holdsAt(F,T)|Gi],res(Hi,Bi,Ni),Rf) :-
   % encontra uma ação do domínio que inicia o fluente F
   initiates(A,F,T1,Pr),
   % adiciona uma ocorrência dessa ação ao resíduo abdutivo
   add(happens(A,T1),Hi,Hf,Ni,Nt,Pr,Gi,Gf),
   % adiciona uma restrição de ordem causal ao resíduo abdutivo
   add(before(T1,T),Bi,B1),
   % garante que o fluente F não deixa de valer dentro do intervalo [T1,T]
   naf(clipped(T1,F,T),Hi,B1,B2),
   % garante que a nova ação não invalida negações por falha anteriores
   naf(Nt,[happens(A,T1)],B2,Bf),
   % coleta no resíduo negativo o literal provado através de negação por falha (naf)
   add(clipped(T1,F,T),Ni,Nf),
   % continua tentando satisfazer demais submetas do problema
   abp(Gf,res(Hf,Bf,Nf),Rf).

% postula uma ocorrência de ação - happens(A,T1) - no resíduo abdutivo
```

```
% mantém o resíduo inalterado
add(happens(A,T1),Hi,Hi,_,[],_,G,G) :-
   % se já existe uma ocorrência happens(A,T1) postulada no resíduo
   member(happens(A,T1),Hi).

% senão, adiciona uma nova ocorrência de ação ao resíduo abdutivo
add(happens(A,T1),Hi,[happens(A,T1)|Hi],Ni,Ni,Pre,Gi,Gf) :-
   % verifica se o limite de profundidade iterativa não vai ser excedido com a nova inclusão
   depth(Hi),
   % incrementa o contador de marcas de tempo (skolemização de variáveis temporais)
   flag(time,T1,T1+1),
   % adiciona precondições da nova ação ao conjunto de submetas a serem satisfeitas
   append(Gi,Pre,Gf).

% atualiza a relação de restrições de ordem temporal

% mantém o resíduo abdutivo inalterado
add(before(X,Y),B,B) :-
   % se X ≺ Y está no fecho transitivo de ≺,
   prec(X,Y,B).

% senão, adiciona a restrição X ≺ Y ao resíduo abdutivo
add(before(X,Y),B,[before(X,Y)|B]) :-
   % se isso for consistente com as restrições existentes
   X\=Y, not prec(Y,X,B).

% Adiciona um literal clipped a lista de negações provadas por falha

add(clipped(T1,F,T2),N,[clipped(T1,F,T2)|N]).

% verifica a consitência do resíduo negativo, refazendo
% a prova por falha (naf) de cada literal coletado nese resíduo

% consistência trivial se não há ocorrências de ações postuladas no resíduo abdutivo
naf(_,[],B,B).

% consistência trivial se não há literais já provados através de negação por falha
naf([],_,B,B).

% verifica a consistência de cada literal existente, para cada ocorrência de ação
naf([N|Ns],H,Bi,Bf) :-
   % verifica a consistência do primeiro literal provado por falha
   naf(N,H,Bi,B),
   % verifica a consistência dos demais literais provados por falha
   naf(Ns,H,B,Bf).


% implementa o axioma [SEC3] do calculo de eventos simplificado, ou seja,
% ¬clipped(T1,F,T2) ↔ ¬happens(A,T) ∨ ¬(T1 ≺ T) ∨ ¬(T ≺ T2) ∨ ¬terminates(A,F,T)

naf(clipped(T1,F,T2),[H|Hs],Bi,Bf) :-
```

```
   % verifica se ¬clipped(T1,F,T2) é consistente com relação à 1ª ação no resíduo abdutivo
   safe(clipped(T1,F,T2),H,Bi,B),
   % continua verificando a consistência de ¬clipped(T1,F,T2), com relação às demais ações
   naf(clipped(T1,F,T2),Hs,B,Bf).

% verifica a consistência de ¬clipped(T1,F,T2) com relação a uma ação happens(A,T)

% um fluente F está seguro com relação à sua ação produtora
safe(clipped(T,_,_), happens(_,T),B,B) :- !.

% um fluente F está seguro com relação à sua ação consumidora
safe(clipped(_,_,T), happens(_,T),B,B) :- !.

% um fluente F está seguro com relação a uma ação que não termina a sua validade
safe(clipped(_,F,_), happens(A,_),B,B) :- not terminates(A,F,_,_), !.

% um fluente F está seguro com relação a uma ação anterior àquela que o produz
safe(clipped(T1,_,_),happens(_,T),B,B) :- prec(T,T1,B), !.

% um fluente F está seguro com relação a uma ação posterior àquela que o consome
safe(clipped(_,_,T2),happens(_,T),B,B) :- prec(T2,T,B), !.

% altera o plano para postergar a ação que ameaça a validade de ¬clipped(T1,F,T2)
safe(clipped(_,_,T2),happens(_,T),B,[before(T2,T)|B]) :- not prec(T,T2,B).

% altera o plano para antecipar a ação que ameaça a validade de ¬clipped(T1,F,T2)
safe(clipped(T1,_,_),happens(_,T),B,[before(T,T1)|B]) :- not prec(T1,T,B).

% computa o fecho transitivo da relação de ordem temporal

% nenhum instante de tempo precede a si mesmo
prec(X,X,_) :- !, fail.

% o instante inicial 0 precede a todos os demais instantes de tempo
prec(0,_,_) :- !.

% o instante final t é precedido por todos os demais instantes de tempo
prec(_,t,_) :- !.

% o instante T1 precede T2 se isso foi estabelecido no resíduo abdutivo
prec(T1,T2,B) :- member(before(T1,T2),B), !.

% o instante T1 precede T2 se T1 ≺ T2 está no fecho transitivo de ≺
prec(T1,T2,B) :-
   % encontra T tal que T1 ≺ T
   member(before(T1,T),B),
   % verifica se T precede T2
   prec(T,T2,B).


% verifica se o limite de profundidade iterativa foi excedido

depth(H) :-
   % determina o número de ocorrência de ações no resíduo abdutivo
```

```
   length(H,L),
   % obtém a profundidade iterativa de busca corrente
   flag(depth,D,D),
   % verifica se o número de ações está dentro do limite
   L<D.

% encontra uma linearização (ordenação topológica) para um plano de ordem parcial

% linearização trivial se não há ações nem restrições de ordem
linearisation([],[]) :- nl, nl.

% se a ação E ocorre no instante Ti
linearisation([happens(E,Ti)|Hs],B) :-
   % que não é precedido por nenhum outro instante (exceto 0)
   not member(before(_,Ti),B), !,
   % exiba essa ação
   format(' n w',E),
   % remove da relação de ordem temporal todos os pares contento Ti
   del(before(Ti,_),B,Bs),
   % continua linearização com as demais ações
   linearisation(Hs,Bs).

% senão,
linearisation([H|Hs],B) :-
   % anexa essa ocorrência ao final da lista de ações
   append(Hs,[H],R),
   % e verifica a próxima ação na lista
   linearisation(R,B).


% remove de uma lista todas as ocorrências de um determinado elemento

% remoção trivial se a lista está vazia
del(_,[],[]).

% senão
del(E,[X|Y],[X|Z]) :-
   % se elemento a ser removido diferente do primeiro da lista
   E\=X, !,
   % continua a remoção com o restante da lista
   del(E,Y,Z).

% senão (elemento a ser removido é igual ao primeiro da lista)
del(E,[_|Y],Z) :-
   % remove esse elemento e continua a remoção no restante da lista
   del(E,Y,Z).
```

Os demais planejadores desenvolvidos como parte dessa dissertação podem ser encontrados no endereço `http://www.ime.usp.br/~slago`.