Planeamento em Inteligência Artificial

Carlos Perestrelo, Eládio Nóbrega, Élio Cró, Emanuel Mendonça

Universidade da Madeira,
9000-080 Funchal, Madeira, Portugal
{a2019802,a2029002,a2030102,a2031302}@max.uma.pt

Abstract. O Planeamento é um aspecto importante dos sistemas inteligentes e um dos problemas centrais da área de Inteligência Artificial. O planeamento permite essencialmente que seja aumentada a flexibilidade e autonomia dos sistemas ao determinar qual a sequencia de acções mais prováveis de atingir a solução. Esta é uma área de grande investigação nas últimas décadas, as técnicas de planeamento são aplicadas na robótica, agentes autónomos, missões espaciais e muito mais.

1 Introdução

A inteligência artificial é um ramo da ciência computacional que lida com a obtenção de soluções para problemas complexos de uma forma que se aproxime à dos humanos. Isto é conseguido através da análise das características do raciocínio humano e aplicando-as como algoritmos que uma máquina possa executar. Embora esteja geralmente associada à ciência da computação, a inteligência artificial recebe contributos de outras áreas como a Matemática, Biologia, Psicologia e Filosofia.

Os computadores são máquinas orientadas para a resolução de problemas matemáticos utilizando regras pré-programadas, isto faz com que tenham muito dificuldade em compreender os problemas que lhes são apresentados e principalmente, dificuldade em se adaptar a novas situações. A Inteligência Artificial pretende melhorar o comportamento das máquinas perante situações complexas.

1.1 O Planeamento em Inteligência Artificial

No mundo da Inteligência Artificial, a área do Planeamento é uma das que tem visto maior interesse por parte dos investigadores.

O Planeamento é essencialmente a tarefa de descobrir a sequência correcta de acções que nos permita atingir uma solução. Este lida com aspectos da formalização, implementação e avaliação de algoritmos que permitem construir um plano, permitindo assim reduzir o espaço de busca, resolver conflitos entre objectivos, fornecer uma base para corrigir erros.

Existem várias formas de planeamento, entre as quais, o Planeamento Linear e Não-linear, o Planeamento de Ordem Parcial, o Planeamento Heurístico, o Planeamento Hierárquico, o Planeamento Condicional e o Planeamento Probabilístico.
Iremos analisar algumas formas de planeamento heurístico e hierárquico.

1.2 Planeamento Clássico

Quando se pretende criar um plano para a resolução de um determinado problema muitas vezes surgem problemas como:

O próprio problema não está bem definido,
Não existe um acordo na preferência dos objectivos a alcançar
Não se conhecem os resultados de determinadas acções
Existem restrições de tempo/recursos
Agentes externos podem interferir na determinação da solução

De forma a reduzir a incerteza e a tornar o planeamento mais fácil é necessário:

Representar explicitamente os estados, objectivos e acções – permitindo centralizar a acção em determinados objectivos em detrimento de outros e fazer aplicar algoritmos específicos
Decompor os objectivos – aplicar a técnica de 'dividir para conquistar' na realização do plano

Isto permite-nos obter um ambiente de Planeamento Clássico, que vai simplificar a determinação de soluções e aplicar certos algoritmos. Para simplificar todo o processo é necessário:

Conhecer o estado inicial
Ter apenas acções determinísticas
Um agente único, sem eventos externos
Representação simples das acções
Inexistência de acções concorrentes
Não ter limitações de recursos

2 Agentes

Um agente é basicamente tudo aquilo que pode ser visto como percepcionando o meio com os seus sentidos e a actuar sobre o meio através das suas acções.

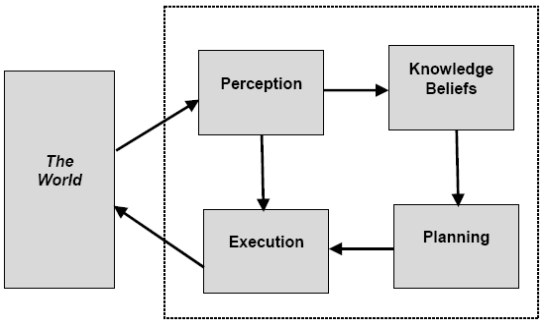


Fig. 1. Arquitectura de um agente

Os agentes podem seleccionar as suas acções não apenas com base nas suas percepções ou no modelo interno do estado concorrente. Temos assim dois tipos de agentes:

Agentes que resolvem problemas – agentes que planeiam para a frente, considerando as consequências das acções antes de actuar.

Agentes baseados em conhecimento – agentes que seleccionam as acções com base nas representações lógicas e explicitas, no estado corrente e nos efeitos das acções.

Um agente de planeamento funcionará da seguinte forma

1. Gera um objectivo a atingir
2. Gera um plano para atingir o objectivo, partindo do estado actual
3. Executa o plano até este estar terminado
4. Começa novamente com um novo objectivo

3 Mundo dos Blocos

O Mundo dos Blocos é um dos exemplos clássicos para o Planeamento em Artificial, será utilizado em alguns dos exemplos ao longo deste trabalho. O objectivo do problema é empilhar um conjunto de blocos assentes numa superfície numa certa ordem a partir de um estado inicial. Para tal os blocos devem ser movidos um de cada vez de uma pilha para a outra, sendo apenas possível mover um bloco de cada vez.

O problema requer que:

Todos os blocos têm o mesmo tamanho

A superfície é fixa, e a posição dos blocos nela é irrelevante

Pode haver um número qualquer de blocos na mesa

Os blocos são erguidos e pousados por um braço

O braço pode pegar em qualquer bloco numa dada altura

Fig. 2. O Mundo dos Blocos

Veremos seguidamente com resolver o problema do mundo dos blocos utilizando o cálculo situacional e o algoritmo STRIPS.

4 Cálculo Situacional

O Cálculo Situacional é um formalismo utilizado para representar problemas de planeamento e consiste em aumentar a lógica de primeira ordem de maneira a que possa ser utilizada para **raciocinar sobre acções no tempo**, utilizando para tal **variáveis de situação**, as quais especificam o tempo.

No Cálculo Situacional, o mundo é considerado como uma sequência de situações onde cada afirmação passa a ser feita em relação a uma determinada situação, sendo neste caso **situação** definida como uma imagem do mundo num intervalo de tempo em que não há mudanças.

Por exemplo: em vez de $\mathbf{P(x_1, x_2, ..., x_n)}$ utilizamos $\mathbf{P(x_1, x_2, ..., x_n, s)}$ para dizer que **P** é verdade na situação **s**.

Nesta abordagem ao problema do planeamento é utilizada uma função especial **resultado(a, s)** a qual recebe a situação actual **s** e uma acção **a** e devolve a nova situação resultante de executar **a** em **s**.

Por exemplo: a acção **mover_agente_posição_y** pode ser representada por $\forall_{x,y,s}$: em(Agente,x,s) $\wedge$ ~na_cadeira(Agente,s) $\Rightarrow$ em(Agente, y, resultado(mover(Agente, y),s))

Exemplo de representação para um problema de empilhamento:

Estado inicial:

On(A,B,S0) $\wedge$ On(B,C,S0) $\wedge$ On(C,Fl,S0) $\wedge$ Clear(A,S0) $\wedge$ Clear(F2,S0)

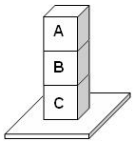


Fig. 3. Estado Inicial

Operadores:

$\forall_{x, y, z, s}$: On(x, y, s) $\wedge$ Clear(x, s) $\wedge$ Clear(z, s) $\wedge$ x $\neq$ z $\Rightarrow$ On(x, z, result(move(x, y, z), s))

Estado final:
$\exists s$: On(A, C, s) $\wedge$ On(B, A)

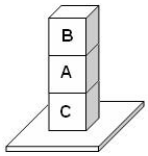


Fig. 4. Estado Final

A aplicação de operadores faz-nos progredir de situação em situação.

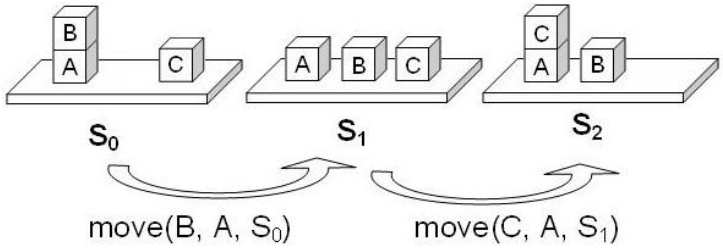


Fig. 5. Progressão das situações

4.1 Limitações do Cálculo Situacional

O Cálculo Situacional possui algumas limitações, que de seguida descrevemos:

4.1.2 Problema da Qualificação

As acções no cálculo situacional descrevem apenas aquilo que **mudam** e não aquilo que **não mudam**, o que provoca que não saibamos quais as condições para que uma dada acção garantidamente se possa executar.

4.1.3 Problema da Persistência

A determinação do que é que ainda é verdade na situação resultante consiste em acrescentar axiomas de persistência que declarem explicitamente aquilo que não muda após a acção.
Exemplo: quando pela acção
$\forall_{x,y,s}$: em(Agente,x,s) $\wedge$ ~na_cadeira(Agente,s) $\Rightarrow$ em(Agente, y, resultado(mover(Agente, y),s)),
o Agente muda de posição, os outros objectos do mundo mantêm-se no mesmo sítio. Devemos então inserir um axioma de persistência que diga isso mesmo para cada objecto:

$\forall$x,y,s: em(Bananas,x,s) $\Rightarrow$ em(Bananas, y, resultado(mover(Agente, y),s))

O problema com esta abordagem resulta do facto de ser necessário um número elevado de axiomas de persistência. Por exemplo: de cada vez que um novo predicado é acrescentado torna-se necessário acrescentar axiomas de persistência que o referenciem a todos os operadores já existentes.

4.1.4 Problema Inferencial

O facto de ser "transportado" um valor que não se altera ao longo de uma cadeia de inferência longa, em vez de, registar apenas o que se altera levanta os chamados problemas de inferência.

4.1.5 Outras limitações

As situações são pontos instantâneos no tempo, como tal, são vocacionados para mundos em que ocorre uma acção de cada vez.

Se existirem múltiplas acções em simultâneo ou múltiplos agentes no mundo, são necessários que se definam **acções compostas**, mas quando as acções têm durações diferentes ou os seus efeitos dependem da duração, o cálculo situacional não pode ser usado.

Resumindo, o Cálculo Situacional apesar de tudo, é suficiente para muitos domínios de planeamento, mas não para os mais interessantes.

5 Linguagem STRIPS

De maneira a evitar os problemas do cálculo situacional, a generalidade dos planeadores usam uma linguagem lógica simplificada chamada STRIPS (ou uma descendente desta). O sistema STRIPS (STanford Research Institute Problem Solver) foi a primeira proposta de solução para o problema de planeamento integrando um sistema forma e foi apresentado por Fikes e Nilsson em 1971.

Esta linguagem mantém muita da expressividade do cálculo situacional, mas como é uma linguagem mais restritiva, diminui o número de soluções onde procurar.

Esta linguagem recorre a algoritmos específicos, especialmente concebidos para processar as linguagens restritas de forma eficiente.

5.1 Características da linguagem STRIPS

As acções são representadas de forma simplificada relativamente à representação em Lógica.

Representação de um estado ou situação.

Um estado (inicial ou não) é representado por uma conjunção de literais sem variáveis, não havendo, por isso, quantificadores. Não existe também, uma representação explícita do tempo nem regras de inferência explícitas, que eram limitações que o Cálculo Situacional possuía. Por exemplo:

Em(Casa) ∧ ~Ter(Leite) ∧ ~Ter(Bananas) ∧ ~Ter(Berbequim).

Representação do objectivo

Os objectivos são também representados por conjunções de literais, podendo, no entanto, conter também variáveis que se assume estarem universalmente quantificadas. Por exemplo o objectivo de estar num sítio que venda leite pode ser representado por:

Em(x) ∧ Vende(x, Leite).

Representação dos operadores

Os operadores em STRIPS são constituídos por três partes: uma descrição da acção, uma pré-condição e um efeito. A descrição da acção é constituída por um termo que refere uma possível acção. Por exemplo, **coloca(x)**.

A pré-condição é uma conjunção de literais positivos que têm que se verificar imediatamente antes da acção ser executada para que o operador possa ser aplicável, ou seja, define as situações em que uma acção é aplicável. Por exemplo:

em(x) ∧ ter(x).

Nesta linguagem os operadores podem ser representados de duas formas, forma descritiva e forma gráfica.

Exemplo: forma escrita:

Nome: pickup(x)
Pré-condições: ontable(x), clear(x), handempty
Add List: holding(x)
Delete List: ontable(x), clear(x), handempty

Exemplo:

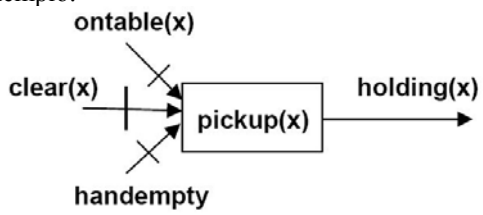


Fig. 6. Forma Gráfica

Os operadores do tipo STRIPS resolvem o problema da persistência assumindo que tudo o que é verdadeiro antes de uma acção ser executada continua a ser verdadeiro após a sua execução a não ser que esteja na **Delete List** e tudo o que não era verdadeiro antes da acção ser executada continua a não ser verdadeiro após a sua execução a não ser que faça parte da **Add List**.

5.2 Algoritmo STRIPS

O STRIPS é um planeador regressivo baseado numa pilha de objectivos que em vez de utilizar o algoritmo regressivo para gerar estados anteriores, fazendo o encadea-

mento para trás deste o estado final até ao inicial, usa uma **pilha de objectivos** para manter o conjunto de objectivos que ainda não foi satisfeito.

Ele mantém uma representação do estado actual que é inicializada com o estado inicial.

Abordagem:

1. Escolhe-se uma ordem para tentar satisfazer os objectivos.
2. Quando um objectivo é retirado da pilha, se não for verdade no estado actual, insere-se na pilha um operador que o adicione.
3. Seguidamente inserem-se na pilha as pré-condições do operador que vamos tentar satisfazer seguindo o mesmo método.
4. Quando todas as pré-condições estão satisfeitas (portanto fora da pilha) retira-se o operador da pilha e verifica-se se todas as suas pré-condições ainda são verdade no estado actual.
5. Se isso se verificar executa-se a acção criando-se um novo estado (actualiza-se a representação do estado actual).

Algoritmo:

```
strips(I, G1^G2^...^Gn, Ops)
        // I é o estado incial
        // G1^G2^...^Gn corresponde à conjunção dos objectivos finais
        // Ops é a lista de operadores
push(achieve(G1^G2^...^Gn), Goal_Stack)
loop:
        if empty(Goal_Stack) then return(success)
        N = pop(Goal_Stack)
        if N tiver a forma "achieve(g1, g2, ..., gn)" then
                if N for verdade no estado I then goto loop
                if todas as ordens dos gi já tiverem sido tentadas then return(failure)
                push(N, Goal_Stack)
                escolher uma nova ordem para os gi
                push(achieve(gi), Goal_Stack) na ordem escolhida
        else
                if N tiver a forma "achieve(g)" then
                        if N for verdade no estado I then goto loop
                        escolher um operador O de Ops que adicione g
                        if não houver then return(failure)
                        escolher atribuições de variáveis que façam O adicionar g
                        push(apply(O), Goal_Stack)
                        push(achieve(preconditions(O)), Goal_Stack)
                else
                        if N tiver a forma "apply(O)" then
                                I = apply(O, I) //criar novo estado
goto loop
```

Se a resolução de um objectivo posterior fizer com que um objectivo anterior deixe de estar satisfeito então o algoritmo falha.

O algoritmo verifica se houve **interacções** entre objectivos fazendo a reverificação de cada conjunção de objectivos

Este planeador é um planeador regressivo porque os objectivos e sub-objectivos vão sendo gerados por encadeamento para trás, apesar de que no entanto, os operadores vão sendo aplicados na direcção contrária.

Exemplo de interacção:

Vamos utilizar o "mundo dos blocos" para exemplificar a representação das acções no STRIPS.

STACK(x,y)

P: clear(y), holding(x)
D: clear(y), holding(x)
A: armempty, on(x,y)

UNSTACK(x,y)

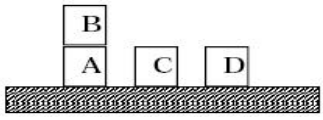
P: on(x,y), clear(x), armempty
D: on(x,y), armempty
A: holding(x), clear(y)

PUTDOWN(x)

P: holding(x)
D: holding(x)
A: armempty, ontable(x)

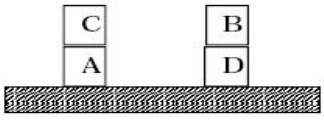
PICKUP(x)

P: ontable(x), clear(x),armempty
D: ontable(x), armempty
A: holding(x)



Estado Inicial

on(B,A)
clear(B)
clear(C)
clear(D)
ontable(A)
ontable(C)
ontable(D)
armempty



Estado Final

on(C,A)
on(B,D)
ontable(A)
ontable(D)

Fig. 7. Estado inicial e estado final:

Aplicação do algoritmo STRIPS:

Vamos partir do estado final até atingir o estado inicial.

Comecemos por utilizar o operador Stack instanciado com C e A, seguido do operador Unstack. Como não sabemos onde o bloco estava, vamos supor que estava em cima do bloco B, instanciando o operador com x = B.

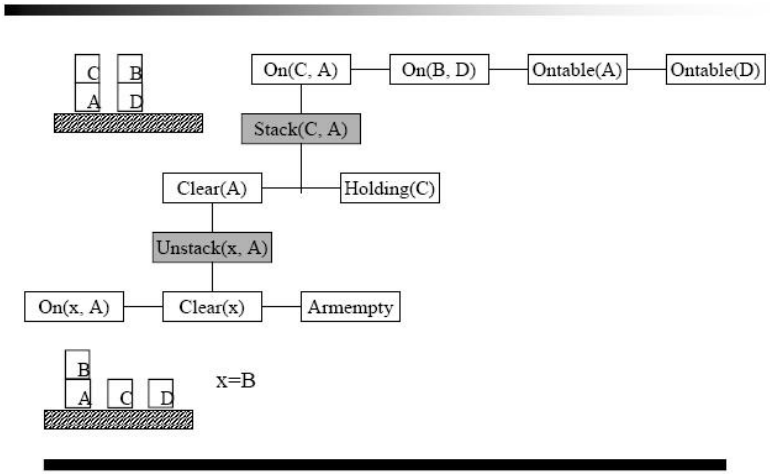


Fig. 8. Primeiro conjunto de passos da resolução

A representação do estado actual é o seguinte:

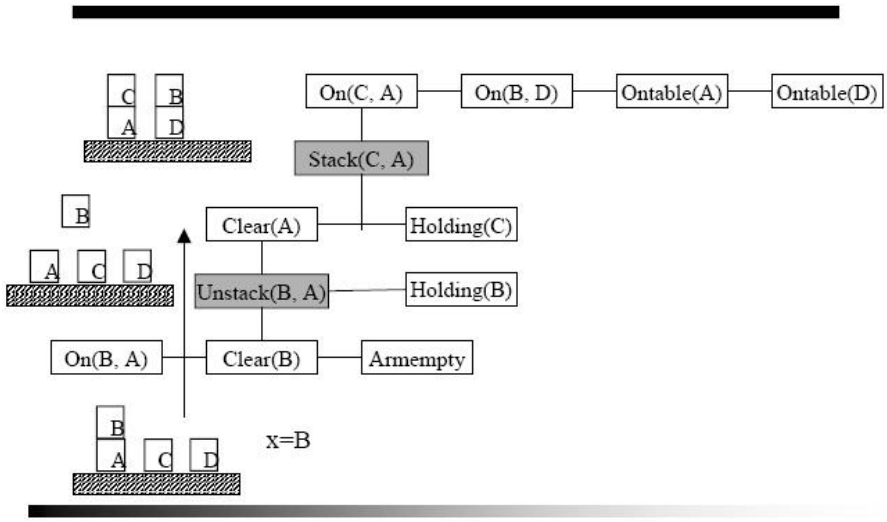


Fig. 9. Estado actual

Vamos agora explorar os ramos restantes, utilizando o operador Pick(C).

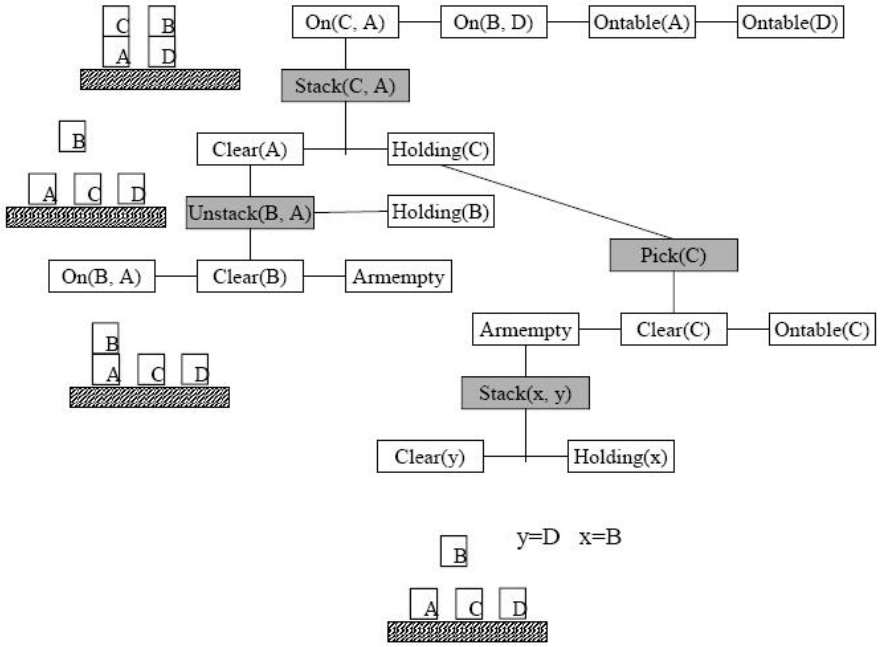


Fig. 10. Exploração dos ramos com Pick(C)

Instanciamos o operador Stack, com y = D e x = B e obtemos o estado final.

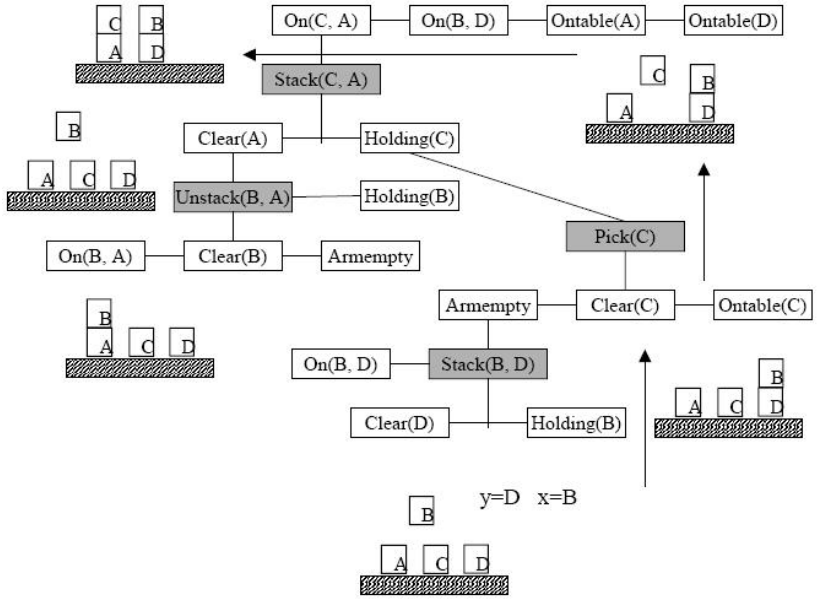


Fig. 11. Estado Final da resolução

Resumindo, os planeadores tipo-STRIPS possuem as seguintes vantagens: permitem manter o espaço de pesquisa reduzido (um objectivo de cada vez), bom se os objectivos são independentes e é um algoritmo Correcto. Em relação às desvantagens, pode produzir soluções sub-óptimas e não é completo.

Apesar de o STRIPS não ser perfeito, serviu como base de desenvolvimento a muitos outros planeadores que tentaram melhorar o seu desempenho e agregar maior capacidade de representação à linguagem.

6 Anomalia de Sussman

Em 1973, Sussman apresentou o planeador HACKER, que foi um descendente directo do STRIPS de Fikes e Nilsson. Foi neste artigo que Sussman apresentou a "Anomalia de Sussman": No mundo dos bloco onde o estado inicial é dado por:

sobre(C, A) ∧ sobre(A, Mesa) ∧ sobre(B, Mesa)

E o estado final por:

sobre(C, Mesa) ∧ sobre(A, B) ∧ sobre(B, C)

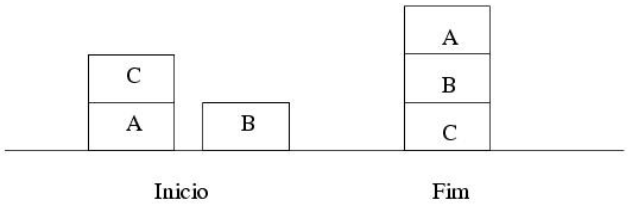


Fig. 12. Problema do mundo dos blocos conhecido como "Anomalia de Sussman"

O problema era considerado anómalo porque os planeadores "não intercalados" do início dos anos 1970 não o conseguiam resolver. Sussman descobriu que se os objectivos parciais (subgoals) são independentes, então podem ser sequencialmente alcançados, independentemente da ordem.

O próprio HACKER conseguiu resolver este problema simples. O STRIPS também não conseguia encontrar uma solução sem redundâncias para este problema, uma vez que, tratava os sub-objectivos isoladamente, em vez de os intercalar.

Algum tempo depois, foram desenvolvidos sistemas subsequentes como WARPLAN e INTERPLAN abordavam directamente o problema da Anomalia de Sussman através da reordenação das acções do plano.

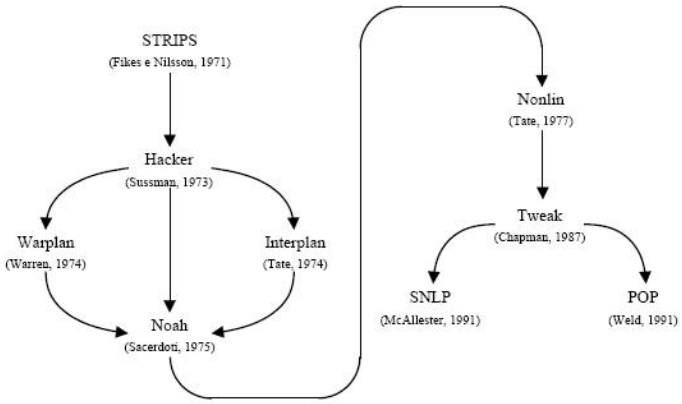


Fig. 13. Hierarquia com alguns planeadores que utilizam representações derivadas de STRIPS

Hoje em dia, são utilizados planeadores que permitam a intercalação de passos para resolver problemas deste tipo, pois lidam de forma mais adequada com as dificuldades levantadas pelas interacções entre objectivos, como por exemplo o POP (Partial-order Planner).

7 Algoritmos de Planeamento

A resolução de problemas complexos da vida real com os algoritmos standard (Depth-first, A*, etc.) mostra que estes são ineficientes. Os problemas detectados nestes algoritmos são os seguintes:

O problema do ruído, que permite que na resolução de um problema sejam considerados acções irrelevantes para atingir um determinado objectivo. Ex: Consideremos a tarefa de comprar um certo livro numa loja on-line. Se existe uma acção para cada nº de ISBN (código de identificação do livro), então se tivermos 10 milhões de livros, um algoritmo de procura teria de percorrer essas 10 milhões de acções para encontrar o livro correcto. Um agente de planeamento inteligente deveria associar de um objectivo ter_livro(ISBN-101234567) à acção comprar(ISBN-101234567) directamente.

Conseguir encontrar uma boa função heurística. A função heurística tem de retornar um valor bastante realista, o que normalmente não acontece (a função heurística calcula o custo de tomar uma acção num determinado estado, se tivermos que percorrer todas as acções possíveis).

Estes algoritmos não podem tirar vantagem da decomposição do problema em sub problemas, com o intuito de resolver os sub-problemas em separado e juntar os resultados no fim.

Agora vamos abordar um conjunto de algoritmos de planeamento que permitem ter resultados mais satisfatórios que os algoritmos de procura tradicionais tais como o Deph-first e o A*.

7.1 Procura num espaço de estados

Como temos numa acção pré-condições e efeitos podemos realizar uma procura em duas direcções. Podemos efectuar a procura desde o estado inicial até atingirmos o objectivo, ou podemos iniciar a procura no objectivo e ir andando para trás. À primeira procura chamamos de "*Planeamento em progressão*" e à segunda de "*Planeamento em regressão*".

Para efeitos de estudo dos algoritmos de procura num espaço de estados vamos considerar o seguinte problema, descrito na linguagem STRIPS:

Init(At(C1,SFO) ∧ At(C2,JFK) ∧ At(P1,SFO) ∧ At(P2,JFK)
∧ Cargo(C1) ∧ Cargo(C2) ∧ Plane(P1) ∧ Plane(P2)
∧ Airport(JFK) ∧ Airport (SFO)
Goal(At(C1,JFK) ∧ At(C2,SFO))
Action(Load(c,p,a),
PRECOND: At(c,a) ∧ At(p,a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)
EFFECT: ¬At(c,a) ∧ At(c,p))
Action(Unload(c,p,a),
PRECOND: In(c,p) ∧ At(p,a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)
EFFECT: At(c,a) ∧ ¬In(c,p))

Action(Fly(p,from,to), PRECOND: At(p,from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to) EFFECT: ¬At(p,from) ∧ At(p,to)))

Table 1. Problema de planeamento involvendo o transporte de carga entre aeroportos.

7.1.1 Procura em progressão

A formulação de problemas de planeamento com este algoritmo é feita da seguinte forma:

O estado inicial da procura é o estado inicial do problema de planeamento.

Cada estado será um conjunto de literais positivos.

As acções que se aplicam a um estado são as em que as pré-condições são satisfeitas.

O estado sucessor resultante de uma acção é gerado adicionado os literais positivos e apagando os literais negativos

O teste do objectivo verifica se o estado satisfaz o objectivo do problema de planeamento.

O custo de cada acção é geralmente 1, este custo pode ser colocado pelos utilizadores.

De seguida mostramos a resolução do problema de transporte de carga:

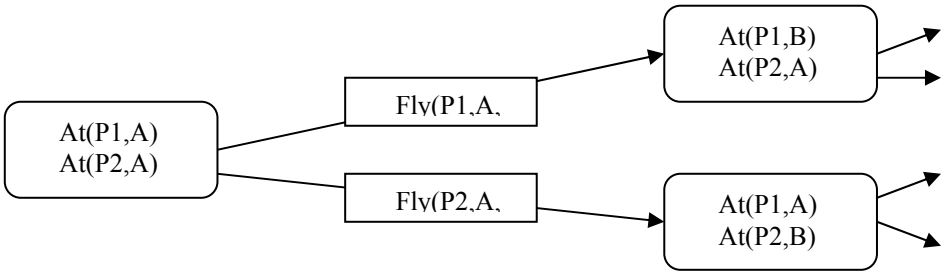


Fig. 14. Procura em progressão, em que a procura começa no estado inicial até ao objectivo

A conclusão que retiramos deste algoritmo é que é ineficiente devido aos seguintes factores:

Todas as acções são consideradas em cada estado, sendo analisadas acções irrelevantes para a resolução do problema

Quanto mais acções, mais caminhos possíveis a percorrer e em alguns casos podemos chegar a um nível aonde o problema não é computável. Imaginemos que no exemplo do transporte de carga entre aeroportos temos 10 aeroportos com 5 aviões em cada e 20 peças de carga para transportar. Temos 10*5*20 (1000) possíveis acções. Como cada avião pode voar para 9 aeroportos diferentes temos 10*5(50) aviões que podem realizar 1000 acções diferentes cada, menos para o aeroporto que pertencem, logo 50 – 9. A árvore terá 1000^41 nós a percorrer. Um numero bastante elevado!

7.1.2 Procura em regressão

A procura em regressão permite-nos considerar apenas as acções relevantes para a resolução do nosso problema. Consideramos uma acção como sendo relevante para um objectivo conjuntivo se essa acção tiver como efeito um dos elementos da conjunção. Por exemplo se um objectivo for ter 20 peças de carga num aeroporto B: At(C1,B) ∧ At(C2,B) ∧ At(C3,B) ∧ .. ∧At(C20,B) . Considerando a conjunção At(C1,B), temos de procurar uma acção que tenha esta última como efeito. Unload(C1,p,B) tem como efeito At(C1,B) logo podemos considerá-la como relevante(ver o exemplo de transporte de cargas entre aeroportos descrito em 1.1).

O processo de construir estados antecessores ao estado objectivo é o seguinte:
Dado um estado objectivo G, seja A um acção que é relevante e consistente(quando uma acção não elimina literais contidos no estado objectivo). O correspondente antecessor é calculado da seguinte forma:

Quaisquer efeitos positivos de A que aparecem em G são apagados.

Cada literal que seja pré-condição de A é adicionado. Caso este literal já exista ele é ignorado.

A procura acaba quando for gerado um estado antecessor que satisfaça o estado inicial do problema de planeamento.

De seguida mostramos a resolução do problema de transporte de carga:

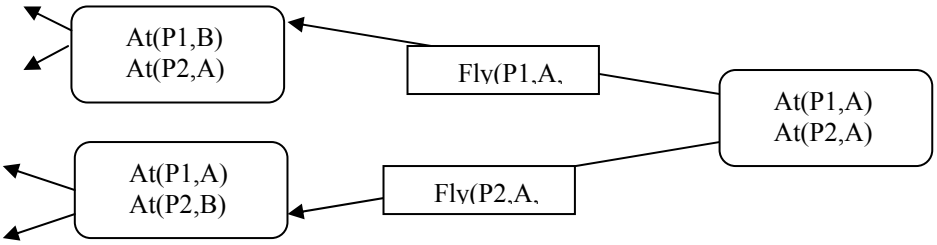


Fig. 15. Procura em regressão, em que a procura começa no estado objectivo até ao objectivo

7.1.3 Heurísticas para o planeamento num espaço de estados

As procuras em regressão e progressão são ineficientes sem uma boa função heurística. A função heurística estima a distância de um estado até um objectivo. Usando a linguagem STRIPS, o custo de cada acção é 1, logo a distância é o número de acções. Podemos usar duas aproximações para calcular uma heurística:

Derivar um **problema simplificado** da especificação do problema a resolver

Usar a assunção de independência do sub-objectivo: Neste caso o custo de resolução dum conjunto de sub-objectivos é aproximado pela soma dos custos de resolução de cada um dos sub-objectivos de forma independente.

Vamos ver agora como derivar problemas de planeamento simplificado.

A partir das representações explícitas de pré-condições e efeitos disponíveis, o processo irá funcionar modificando essas representações (pode-se comparar esta aproximação com os problemas de busca, onde a função sucessor é uma caixa-negra).

A ideia mais simples é simplificar o problema removendo todas as suas pré-condições das acções. Depois todas as acções serão aplicadas e qualquer literal pode ser atingido num passo (se for uma acção for aplicável, caso contrario, o objectivo é impossível).

Normalmente o número de passos necessários à resolução de um conjunto de objectivos é igual ao número de objectivos que ainda não foram alcançados, mas nem sempre isso acontece. Por exemplo, algumas acções podem conseguir vários objectivos.

Em muitos casos, a heurística mais exacta é obtida considerando pelo menos as interacções positivas levantando das acções que consiga múltiplos objectivos.

Em primeiro lugar removemos os efeitos negativos e, em seguida, contamos o número mínimo de acções necessárias de modo a que a união dessas acções satisfaça o objectivo.

Por exemplo:

Goal (A∧B∧C)
Action(X, EFFECT: A∧P)
Action(Y, EFFECT: B∧C∧Q)
Action(Z, EFFECT: B∧P∧Q)

O mínimo necessário para cobrir o objectivo {A, B, C} é dado pelas acções {X, Y}, logo a heurística retorna um custo de 2.

Também é possível gerar problemas simplificados removendo efeitos negativos sem remover pré-condições. Ou seja, se uma acção tem o efeito A ∧ ¬B no problema original, irá ter o efeito A no problema simplificado. O que significa que não temos que nos preocupar com as interacções negativas entre subplanos porque nenhuma acção pode apagar os literais atingidos por outra acção.

O custo da solução dada pelo problema simplificado é chamado **empty-delete-list heuristic**. A heurística é completamente exacta, mas a sua computação implica correr um algoritmo de planeamento.

A heurística descrita pode ser usada quer em progressões quer em regressões.

7.2 Planeamento de ordem parcial

As procuras num espaço de estados em progressão e em regressão são formas particulares da procura totalmente ordenada. Estes tipos de algoritmos exploram apenas sequências lineares de acções ligados directamente ao estado inicial ou ao objectivo.

Isto significa que eles não podem tirar proveito de decompor um problema em vários sub-problemas. Em vez de trabalharem em cada sub-problema separadamente, esses algoritmos têm de tomar decisões sobre as acções de sequência de todos os sub-problemas.

O objectivo deste algoritmo é separar os problemas em sub-problemas e assim ter a vantagem de flexibilidade na ordem em que construímos o plano. (ver melhor) O planeador pode trabalhar em decisões ou partes importantes primeiro em vez de trabalhar em todos os passos em ordem cronológica. (ver melhor)

Qualquer algoritmo de planeamento que consiga colocar duas accões num plano sem especificar qual deles vem primeiro é chamado de planeador de ordem parcial. Vamos mostrar um problema de colocar um par de sapatos. O problema é descrito formalmente da seguinte maneira:

Goal(RigthShoeOn ∧ LeftShoeOn)
Init()
Action(RightShoe, PRECOND: RightSockOn, EFFECT: RightShoeOn)
Action(RightSock, EFFECT: RightSockOn)
Action(LeftShoe, PRECOND: LeftSockOn, EFFECT: LeftShoeOn)
Action(LeftSock, EFFECT: LeftSockOn)

Table 2. Descrição Formal do Problema

O problema anterior seria resolvido da seguinte forma:

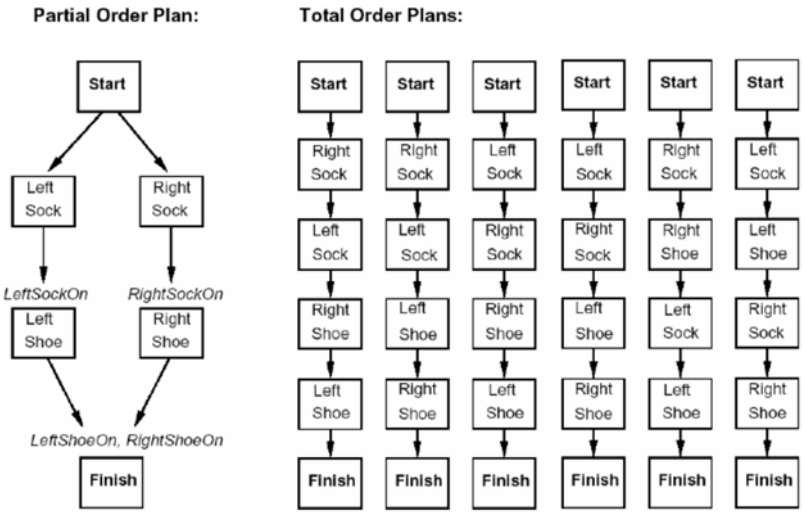


Fig. 16. Planeamento de Ordem Total vs Planeamento de Ordem Parcial

Cada plano é constituído por quatro componentes:

Actions

Constituem os vários passos do plano.

O plano vazio contem apenas as acções Start e Finish. Start não tem pré-condições e tem como efeito todos os literais no estado inicial do problema de planeamento. Finish não tem efeitos e tem como pré-condições os literais objectivo do problema de planeamento.

Ordering Constraints

Estes elementos definem limitações na ordem das acções. São da forma A < B, e lê-se como "A antes de B" e significa que a acção A tem que ser executada antes da B.

Qualquer ciclo como: A < B e B <A representam uma contradição por isso as regras que dão origem a ciclos não podem ser adicionadas ao plano.

Causal Links

Uma ligação causal entre duas acções A e B num plano é escrito como:

$$A \xrightarrow{P} B$$

e deve ser lido como "A atinge *p* por B". No exemplo anterior teríamos que a ligação causal

$$\text{RightSock} \xrightarrow{\text{RightSockOn}} \text{RightShoe}$$

garantia que RightSockOn é um efeito da acção RightStock e uma pré-condição de RightShoe. Também é garantido que RightSockOn deve permanecer a true desde a altura em que é executada a acção RightSock até a altura da acção RightShoe. Por isso é que alguns autores chamam aos"causal links" **intervalos de protecção.** No caso geral apresentado atrás, p está protegido, não podendo ser negado por nenhuma acção entre o intervalo de A até B.

Open preconditions

Uma pré-condição está aberta se não é atingível por alguma das acções do plano. Os planeadores irão trabalhar para reduzir o conjunto de pré-condições abertas até atingir um conjunto vazio, sem introduzirem contradições.

Do plano apresentado na figura anterior podemos identifica as seguintes componentes:

Acções: {RightSock, RightShoe, LeftSock, LeftShoe, Start, Finish}

Ordens: {RightSock < RightShoe, LeftSock < LetfShoe}

Ligações:

$$\text{RightSock} \xrightarrow{\text{RightSockOn}} \text{RightShoe}$$

$$\text{LeftSock} \xrightarrow{\text{LeftSockOn}} \text{LeftShoe}$$

$$\text{RightShoe} \xrightarrow{\text{RightShoeOn}} \text{Finish}$$

$$\text{LeftShoe} \xrightarrow{\text{LeftShoeOn}} \text{Finish}$$

Open Preconditions: {}.

Agora vamos resolver um problema usando o algoritmo de planeamento de ordem parcial. O objectivo é substituir um pneu furado por um sobresselente.

Init(At (Flag, Axle) At (Spare, Trunk))

Goal(At (Spare, Axle))

Action(Remove (Spare,Trunk),
PRECOND: At (Spare, Trunk)
EFFECT: ¬ At (Spare, Trunk) □ At (Spare, Ground)
)

Action(Remove (Flat, Axle),
PRECOND: At (Flat, Axle)
EFFECT: ¬ At (Flat, Axle) □ At (Flat, Ground)
)

Action(PutOn (Spare, Axle),
PRECOND: At (Spare, Ground) □ ¬ At (Flat, Axle)
EFFECT: ¬ At (Spare, Ground) □ At (Spare, Axle)
)

Action(LeaveOvernight,
PRECOND:
EFFECT: ¬ At (Spare, Ground) □ ¬ At (Spare, Axle) □ ¬ At (Spare, Trunk)
□ ¬ At (Flat, Ground) □ ¬ At (Flat, Axle)
)

Table 3. Descrição do Problema da Mudança de Pneu

A procura para a solução começa com o plano inicial, contendo a acção de Start com o efeito At(Spare,Trunk) ∧ At(Flat, Axle) e a acção Finish com a pré-condição At(Spare,Axle). A sequencia de eventos para resolver o problema é a seguinte:

1. Escolher a única pré-condição aberta, At(Spare,Axle) da acção Finish. Escolher a única acção aplicável. PutOn(Spare, Axle).
2. Escolher *At (Spare, Axle)*, pré-condição de *PutOn (Spare, Axle)*. Escolher a única acção aplicável, *Remove (Spare,Trunk)* para atingi-la. O plano resultante é mostrado na figura seguinte:

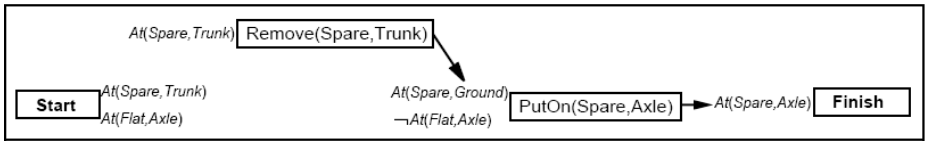


Fig. 17. Plano de Ordem Parcial (incompleto) para o Problema dos Pneus depois de escolher as acções para as duas primeiras pré-condições

3. Escolher ¬ *At (Flat, Axle)*, pré-condição de *PutOn (Spare, Axle)*. Apenas para ser contrário, escolhemos a acção *LeaveOvernight* que é melhor do que a acção *Remo-*

ve (Flat, Axle). Note-se que *LeaveOvernight* também tem o efeito ¬ *At (Spare, Ground)*, o que significa que entra em conflito com a ligação causal:

Remove(Spare, Trunk) $\xrightarrow{\text{At(Spare, Ground)}}$ PutOn (Spare, Axle).

Para resolver este conflito adicionamos uma nova limitação de ordem, colocando *LeaveOvernight* antes de *Remove(Spare, Trunk)*. O resultado será mostrado na figura que se segue:

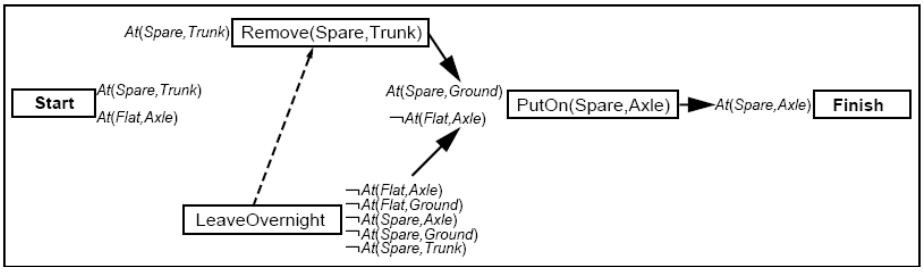


Fig. 18. Plano após escolher LeaveOvernight como acção para atingir ¬At(Flat, Axle)

4. A única pré-condição aberta que nos resta é *At(Spare, Trunk)*, pré-condição da acção *Remove(Spare, Trunk)*. A única acção que pode alcançá-la é a acção Start, mas a ligação causal do *Start* para o *Remove(Spare, Trunk)* está em conflito com *¬At(Spare, Trunk)*, efeito de *LeaveOvernight*. Desta vez não há nenhum modo de resolver o conflito com *LeaveOvernight.* Não podemos colocá-lo antes do *Start* nem nem depois do *Remove(Spare, Trunk)*. Por isso seremos obrigados a recuar, removendo a acção *Remove(Spare, Trunk)* e as últimas duas ligações causais, retornando para a figura resultante do ponto 2.

   Basicamente, o planeador provou que *LeaveOvernight* não funciona como uma forma de mudar um pneu.

5. Considerar novamente ¬ *At (Flat, Axle)*, pré-condição de *PutOn (Spare, Axle)*. Desta vez iremos escolher *Remove (Flat, Axle)*.

6. Uma vez mais, escolhemos *At (Spare, Tire)*, pré-condição de *Remove(Spare, Trunk)* e escolhemos *Start* para alcançá-lo. Não havendo conflitos desta vez.

7. Escolher *At (Flat, Axle)*, pré-condição de *Remove (Flat, Axle)*, e escolher *Start* para alcançá-lo. Isto dá-nos um plano completo e consistente. Ou seja, uma solução, como nos mostra a figura seguinte:

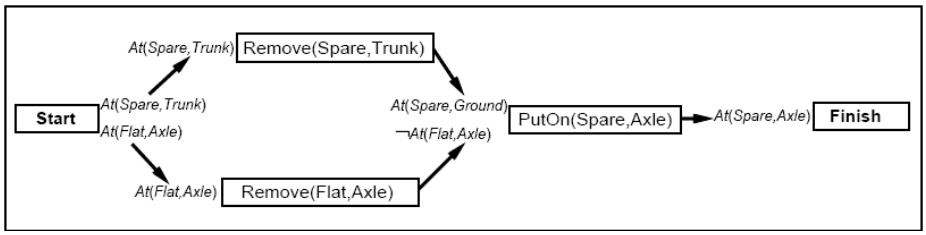


Fig. 19. Solução final para o problema dos pneus.

7.2.1 Heurísticas para o planeamento de ordem parcial

Comparado com o planeamento de ordem total, o planeamento de ordem parcial tem uma clara vantagem em ser capaz de decompor problemas em sub-problemas. Também tem uma desvantagem nisso que é o facto de não representar os estados directamente, por isso é difícil estimar a que distância um plano de ordem parcial está de alcançar um objectivo.

Actualmente, há menos conhecimento de como computar heurísticas exactas usando planeamento de ordem parcial do que para o planeamento de ordem total. A heurística mais óbvia é contar o número de pré-condições abertas distintas. Podendo ser melhorado subtraindo o número de pré-condições abertas que jogam literais no estado Start.

Como no caso da ordem total, este sobrestima o custo quando existem acções que alcançam vários objectivos e subestima o custo quando existem interacções negativas entre passos de um plano.

A função heurística é usada para escolher que plano devemos refinar. Dada esta escolha, o algoritmo gera sucessores baseados na selecção de uma única pré-condição aberta para trabalhar sobre ela.

A heurística **most-constrained-variable** de CSPs pode ser adaptada para algoritmos de planeamento e parece funcionar bem. A ideia é seleccionar a condição aberta que pode ser satisfeita num menor número de modos. Existem dois dos casos especiais nesta heurística. Primeiro, se uma condição aberta não pode ser alcançada por nenhuma das acções existentes, a heurística irá seleccioná-la; isto é uma boa ideia porque uma detecção precoce de impossibilidade pode salvar-nos de um grande trabalho. Segundo, se uma condição aberta pode ser alcançada de uma única forma, então deve ser seleccionada pois pode impor limitações adicionais noutras escolhas para ainda ser feito.

Estudos mostram que o uso destes dois casos especiais dá-nos um grande aumento da velocidade pois a computação de todos os casos possíveis para satisfazer todas as condições abertas tem um elevado custo e nem sempre traz grandes valias.

7.3 Planeamento com lógica proposicional

Nesta secção iremos fazer uma aproximação que consiste em efectuar um teste de satisfabilidade de uma frase lógica melhor do que provar um teorema. Iremos encontrar modelos de frases proposicionais que se assemelham com algo do tipo:

Estado inicial □ todas as descrições possíveis das acções□objectivo.

A frase irá conter símbolos proposicionais correspondentes a todas as acções que podem ocorrer. Um modelo que satisfaça a frase irá assinalar a *true* às acções que são parte de um plano correcto e *false* para as outras.

Se um problema de planeamento não tiver solução, então a frase não será satisfeita.

Descrevendo problemas de planeamento em lógica proposicional

O processo que se utiliza para traduzir os problemas STRIPS em lógica proposicional segue a seguinte ordem:

Devemos começar com um conjunto de axiomas que nos pareça razoável.

Verificamos se estes axiomas são permitidos para modelos.

Por fim, adicionamos mais axiomas, partindo do conjunto inicial.

Vamos começar com um exemplo simples de transporte aéreo . No estado inicial (time0), o avião P1 está em SFO e o avião P2 está em JFK. O objectivo é ter P1 em JFK e P2 em SFO; ou seja, os aviões devem trocar de lugar.

Em primeiro lugar precisamos identificar os símbolos proposicionais para as asserções em cada intervalo de tempo. A notação usada para o estado inicial será escrita da forma seguinte:

$$At(P1,SFO)^0 \wedge At(P2,JFK)^0$$

Uma vez que na lógica proposicional não existe a assumpção de um mundo fechado (closed-word), temos que especificar as proposições que não falsas no estado inicial.

Se algumas proposições forem desconhecidas no estado inicial, então podem não ser especificadas à esquerda (a assumpção de mundo aberto).

Neste exemplo especificamos desta forma:

$$\neg At(P1,JFK)^0 \wedge \neg At(P2,SFO)^0$$

O objectivo deve ser associado a um intervalo de tempo. Desde que não se saiba à partida quantos passos levamos até atingir um objectivo podemos assumir que o objectivo é verdadeiro no estado inicial, tempo T = 0, ou seja, assumimos $At(P1,SFO)^0 \wedge At(P2,JFK)^0$.

Se isso falhar tentamos novamente com T=1 e assim sucessivamente até encontrarmos a solução em que T seja o menor possível.

Para cada valor de T, o conhecimento base incluirá apenas frases que cubram o espaço de tempo entre 0 até T.

Para garantir que existe um fim, será definido um T_{max} de forma aleatória.

Este algoritmo será mostrado em seguida.

function SATPLAN(*problem*, T_{max}) **return** solution or failure
inputs: *problem*, a planning problem
T_{max}, an upper limit to the plan length
for T= 0 **to** Tmax **do**
cnf, *mapping* ← TRANSLATE-TO_SAT(*problem*, T)
assignment ← SAT-SOLVER(*cnf*)
if *assignment* is not null **then**
return EXTRACT-SOLUTION(*assignment*, *mapping*)
return *failure*

O problema de planeamento é transformado numa frase CNF. Se o algoritmo de satisfabilidade encontrar um modelo, então um plano é extraído procurando que símbolos proposicionais que dizem respeito a acções estão assinalados a true no modelo. Se não existir modelo então o processo é repetido com o objectivo a ser movido um passo para a frente.

O próximo objectivo é saber como codificar as descrições das acções em lógica proposicional. A aproximação mais directa é ter uma proposição por cada acção. Por exemplo: Fly(P1, JFK, SFO)^0 é verdade se o avião P_1 voa de SFO para JFK no momento inicial.

Um modo de escrever de forma proposicional os axiomas do estado seguinte seria, por exemplo:

$$At(P1,JFK)^1 \Leftrightarrow (At(P1,JFK)^0 \wedge \neg(Fly(P1,JFK,SFO)^0 \wedge At(P1,JFK)^0)) \vee (Fly(P1,SFO,JFK)^0 \wedge At(P1,SFO)^0)$$

Ou seja, o avião P_1 estará em JFK na hora 1 se estava em JFK no momento 0 e não levantou voo, ou se estava em SFO no momento 0 e voou para JFK. Precisamos de um axioma para cada avião, aeroporto e espaço de tempo. Cada aeroporto adicional irá aumentar as formas de viajar de ou para um dado aeroporto, adicionando também mais disjunções ao lado direito de cada axioma.

Com estes axiomas poderemos correr algoritmos de satisfabilidade de modo a encontrar um plano. Neste caso, deverá ser um plano que atinge o objectivo quando T=1, ou seja, um plano no qual os aviões troquem de lugar.

Suponhamos que o KB é:

Estado inicial □ axiomas para o estado seguinte□objectivo^1 **(equação1)**

Que nos diz que o objectivo se verifica no momento T=1. Podemos verificar que a atribuição na qual $Fly(P1,SFO,JFK)^0 \wedge Fly(P2, JFK, SFO)^0$ são verdadeiras e todos os símbolos das outras acções são falsos num modelo do KB.

É possível que existam outros modelos que estejam de acordo com o algoritmo, mas nem todos esses modelos darão origem a planos satisfatórios. Vamos considerar um plano que é especificado pelas acções:

$$Fly(P1,SFO,JFK)^0 \wedge Fly(P1, JFK, SFO)^0 \wedge Fly(P2, JFK, SFO)^0$$

Este não é um bom plano porque P1 começa em SFO, por isso a acção $Fly(P1,JFK,SFO)^0$ nunca poderá ser satisfeita. No entanto o plano é um modelo da frase definida na **equação1.** Ou seja, está consistente com tudo o que definimos anteriormente.

Para perceber melhor a razão teremos que olhar cuidadosamente para os axiomas que definem o estado seguinte e verificar o que está definido para os casos das acçoes em que as pré-condições não são satisfeitas. Os axiomas prevêem correctamente que nada irá acontecer quando uma acção é executada, mas eles não dizem que uma acção não pode ser executada. Para evitar que sejam gerados planos com acções ilegais temos que adicionar axiomas que definem pré-condições, e obrigam a que para uma acção ocorrer é necessário que se verifiquem as pré-condições.

Por exemplo, precisaríamos de um axioma do tipo:

$$\text{Fly (P1, JFK, SFO)}^0 \Rightarrow \text{At(P1, JFK)}^0$$

Uma vez que temos no estado inicial: ¬At(P1,JFK)0 esta pré-condição garante-nos que todos os modelos têm Fly (P1, JFK, SFO)^0 a falso. Com a adição de **axiomas de pré-condição**, há apenas um modelo que satisfaz todos os axiomas quando o objectivo é atingido no momento T=1, neste caso seria o modelo em que P1 voa para JFK e P2 voa para SFO (neste caso teríamos duas acções paralelas).

No entanto mais problemas poderão surgir. Imaginemos que adicionamos um novo aeroporto (LAX). Com as regras definidas anteriormente seria possível definir um modelo em que:

$$Fly(P1,SFO,JFK)^0 \wedge Fly(P2, JFK, SFO)^0 \wedge Fly(P2, JFK, LAX)^0$$

Ou seja, um avião (P2) poderia viajar para dois lugares (SFO e LAX) ao mesmo tempo... e mesmo assim todos os axiomas de pré-condição seriam satisfeitos. Neste caso os axiomas determinariam que P2 estivesse no momento 1 em dois lugares diferentes: SFO e LAX e o objectivo seria satisfeito.

É obvio que para resolver estes casos precisaremos de mais axiomas, é neste contexto que surgem os **axiomas de exclusão de acções** que irão impedir acções simultâneas.

Uma possível solução seria um axioma do tipo:

$$\neg(Fly(P2, JFK, SFO)^0 \wedge Fly(P2, JFK, LAX)^0)$$

Isto obriga a que todos os planos estejam totalmente ordenados, perdendo a flexibilidade dos planos de ordem parcial. Também, por aumentar o número de espaços de tempo no plano o tempo de computação pode se maior.

Em vez que utilizarmos as exclusões totais podemos recorrer a exclusões parciais, ou seja impedir acções simultâneas apenas quando interferem umas com as outras.

As condições são as mesmas para as acções de exclusão mútua: ou seja, não podem ocorrer ao mesmo tempo se uma nega uma pré-condição ou efeito da outra. Por exemplo: $Fly(P2, JFK, SFO)^0 \wedge Fly(P2, JFK, LAX)^0$ não poderia acontecer pois

cada uma nega a pré-condição da outra. No entanto, $Fly(P1, SFO, JFK)^0 \wedge Fly(P2, JFK, SFO)^0$ poderia verificar-se pois os dois voos não interferem um com o outro.

Os axiomas de exclusão parecem ser muito limitativos. Em vez de dizer que dois aviões não podem voar para diferentes aeroportos ao mesmo tempo podemos dizer simplesmente que nenhum objecto pode estar em lugares diferentes ao mesmo tempo:

$$\forall p, x, y, t \quad x \neq y \Rightarrow \neg(At(p,x)^t \wedge At(p,y)^t)$$

Combinado com os axiomas de estado seguinte, implica que um avião não pode voar para dois aeroportos diferentes ao mesmo tempo. Factos como estes são chamados **state constraints** (limitações de estado). Na lógica proposicional , temos que escrever todas as instancias base de cada limitação de estado. As limitações de estado são normalmente muito mais compactas do que o uso de axiomas de exclusão de acções, mas nem sempre são fáceis de obter da discrição STRIPS original do problema.

É possível provar que este conjunto de axiomas é suficiente, no sentido que de não existiram mais soluções ambíguas ou impossíveis. Qualquer modelo que satisfaça a frase proposicional e todos os seus axiomas serão planos válidos para o modelo original. Ou seja, qualquer linearização do plano é uma sequência legal de acções que alcança o objectivo.

8 Conclusão

Neste trabalho abordamos vários problemas de planeamento e descrevemos as principais representações usadas nos problemas de planeamento e algumas aproximações algorítmicas para resolvê-los. Os pontos a reter são:

Sistemas de planeamento são "problem-solving algoritms" que operam em representações proposicionais explícitas (ou de primeira ordem) de estados e acções. Essas representações tornam possível a derivação de heurísticas efectivas e o desenvolvimento de algoritmos mais poderosos e flexíveis para revolver os problemas.

A linguagem STRIPS descreve as acções recorrendo às suas pré-condições e efeitos e descreve o estado inicial assim como estados-objectivo como conjunções de possíveis literais.

A busca no espaço dos estados pode operar na direcção em frente (progressão) ou em sentido contrário (regressão). As heurísticas efectivas podem ser derivadas assumindo a independência dos sub-objectivos e pela simplificação do problema de planeamento.

Os algoritmos POP (Partial-order planning) exploram o espaço dos planos sem seguir a uma sequência totalmente estabelecida das acções. Vão adicionando acções ao plano de modo a atingirem cada um dos sub-objectivos.

O algoritmo SATplan transforma um problema de planeamento num conjunto de axiomas proposicionais e aplica um algoritmo de satisfabilidade para encontrar o modelo que corresponde a um plano válido.

9 Referências

1. Russel, S., Norvig, P.: Artificial Intelligence – A Modern Approach (Second Edition), Part IV Planning – Chapter 11, Intelligent Agents – Chapter 2. http://aima.cs.berkeley.edu/.
2. Blythe, G., Ambite, J-L., Gil, Y.: CS541: Artificial Intelligence Planning. http://www.isi.edu/~blythe/cs541/.
3. Simmons, R., Veloso, M.: 15-887A: AI Planning, Execution, and Learning. http://www.cs.cmu.edu/~reids/planning/.
4. Hadad, M.: A summer Course: Artificial Intelligence Planning. http://cs.haifa.ac.il/~meirav/HaifaPlan.htm.
5. Khambhampati, S.: CSE: 574: Planning and Learning. http://rakaposhi.eas.asu.edu/planning-class.html/.