

# 11

## PLANEACIÓN

La planeación es un tema de interés tradicional en Inteligencia Artificial, que involucra razonar acerca de los efectos de las acciones y la secuencia en que estas se aplican para lograr un efecto acumulativo dado. En esta sesión desarrollaremos planificadores simples para ilustrar los principios de la planeación.

La Figura 32 muestra una tarea de planeación muy utilizada en Inteligencia Artificial, el mundo de los bloques. De hecho, ya hemos utilizado este escenario al hablar acerca de las búsquedas en espacios de soluciones (ver Capítulo 8, página 77). Sin embargo, en este capítulo, el problema usará una representación que nos permita razonar explícitamente acerca de los efectos de las acciones que nuestro programa puede ejecutar.

### 11.1 ACCIONES

Asumiremos que las acciones causan que el mundo cambie, de forma que éstas inducen cambios entre los estados con que el programa representa al mundo. Ahora bien, una acción generalmente no cambia todo el estado actual de las cosas, solo algunos de los componentes del estado. Una buena representación de las acciones debería tomar en cuenta esta “focalización” en el efecto de las acciones.

Para facilitar razonar acerca de estos efectos focalizados de las acciones, un **estado** será representado como una lista de relaciones que son actualmente verdaderas. Y claro, la lista estado incluye solo aquellas relaciones que son relevantes para nuestro problema de planificación. Para el caso del mundo de los bloques, tales relaciones son *en/2* y *libre/1* con la semántica intuitiva. Por ejemplo, el estado del mundo de cubos de la izquierda, en la Figura 32, será:

[ libre(2), libre(4), libre(c), libre(b), en(a, 1), en(b, 3), en(c, a) ]

Cada **acción** posible es entonces definida en términos de las condiciones que deben observarse en el estado, para que esta pueda ser ejecutada y sus efectos esperados, específicamente:

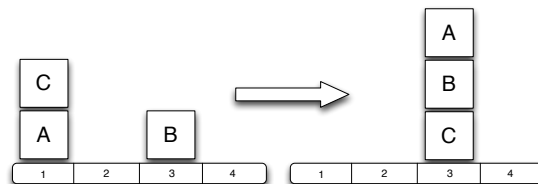


Figura 32: El mundo de bloques, revisitado.

- **Precondición.** Las condiciones que debe satisfacerse, para que la acción pueda ejecutarse.
- **Agregar.** Es una lista de observaciones que, se espera, ocurran después de ejecutarse la acción.
- **Borrar.** Es una lista de observaciones que, se espera, dejen de ser verdaderas después de ejecutarse la acción.

Las precondiciones, pueden definirse por un procedimiento:

```
1 | can(Acc,Cond).
```

que expresa que la acción *Acc* solo puede (*can*) ejecutarse en situaciones de las condiciones *Cond* se sostienen.

Los efectos de una acción pueden definirse de manera similar, por dos procedimientos:

```
1 | add(Acc,ListaAdd).
2 | del(Acc,ListaDel).
```

donde *ListaAdd* y *ListaDel* corresponden a las listas de observaciones agregar y borrar, respectivamente.

En el dominio del mundo de los bloques, la única acción posible será:

```
1 | mover(Bloque,De,A).
```

La definición completa de esta acción es como sigue:

```
1 | can( mover(Bloque,De,A),
2 |       [ libre(Bloque), libre(A),
3 |         en( Bloque,De) ] ) :-
4 |     bloque(Bloque),
5 |     objeto(A),
6 |     A \== Bloque,
7 |     objeto(De),
8 |     De \== A,
9 |     Bloque \== De.
10 |
11 | add( mover(X,De,A), [ en(X,A), libre(De) ] ).
12 |
13 | del( mover(X,De,A), [ en(X,De), libre(A) ] ).
```

De manera que para poder mover un bloque *Bloque* de la posición *De* a la posición *A*, es necesario que el bloque *Bloque* y la posición *A* estén despejados, lo mismo que el bloque *Bloque* esté en la posición *De*. El resto del procedimiento *cond/2* establece restricciones extras: que *Bloque* sea un bloque, y *A* y *De* sean objetos en el universo de discurso; que *A* sea diferente de *Bloque* (no mover el bloque sobre si mismo); que se debe mover el bloque a una nueva posición (*A* es diferente de *De*); y no mover el bloque de sí mismo (*Bloque* es diferente de *De*). Las definiciones de *add/2* y *del/2* completan la especificación de *mover/3*.

Las siguientes definiciones especifican un escenario en el mundo de los bloques:

```

1 objeto( X ) :-
2   lugar( X )
3   ;
4   bloque( X ).
5
6 bloque( a ).
7 bloque( b ).
8 bloque( c ).
9
10 lugar( 1 ).
11 lugar( 2 ).
12 lugar( 3 ).
13 lugar( 4 ).
14
15 estado1( [ despejado(2), despejado(4), despejado(b),
16           despejado(c), en(a,1), en(b,3), en(c,a) ] ).
17
18 metas1([en(a, b)]).

```

Esta definición de las acciones, establece también el espacio de planes posibles, por lo que se le conoce como *espacio de planeación*. Las metas del planeador se definen en términos de una lista de observaciones que se deben cumplir.

Ahora veremos como a partir de esta representación, es posible derivar los planes mediante un procedimiento conocido como análisis medios-fines.

## 11.2 ANÁLISIS MEDIOS-FINES

Consideremos que el mundo de los bloques se encuentra en el estado inicial especificado anteriormente como estado1. Sea la meta del planeador  $en(a, b)$ . Su trabajo consiste entonces en encontrar una secuencia de acciones que satisfagan esta meta. Un planeador típico razonaría de la siguiente forma:

1. Encontrar una acción que satisfaga  $en(a, b)$ . Al buscar en la relación *add*, encontramos que tal acción es de la forma  $mover(a, De, b)$  a partir de cualquier objeto *De*. Tal acción deberá formar parte de nuestro plan, pero no podemos ejecutarla inmediatamente dado nuestro estado inicial.
2. Hacer posible la acción  $mover(a, De, b)$ . Al buscar en la relación *can* encontramos que la condición para ejecutar esta acción es:

$$[despejado(a), despejado(b), en(a, De)]$$

en el estado inicial tenemos que  $despejado(b)$  y que  $en(a, De)$  para  $De/1$ ; pero no que  $despejado(a)$ , así que el planeador se concentra en esta fórmula como su nueva meta.

3. Volvemos a buscar en la relación *add* para encontrar una acción que satisfaga  $despejado(a)$ . Tal acción tiene la forma  $mover(Bloque, a, A)$ . La condición para ejecutar esta acción es:

$$[despejado(Bloque), despejado(A), en(Bloque, a)]$$

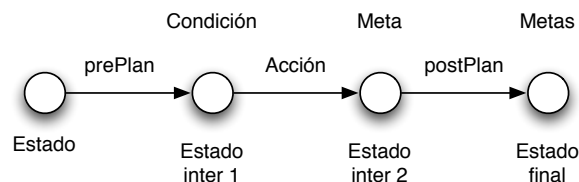


Figura 33: Análisis medios-fines

la cual se satisface en nuestro estado inicial para  $\text{Boque}/c$  y  $A/2$ . De forma que  $\text{mover}(c, a, 2)$  puede ejecutarse en el estado inicial, modificando el estado del problema de la siguiente manera:

- Eliminar del estado inicial las relaciones que la acción borra.
- Incluir las relaciones que la acción agrega al estado inicial del problema.

esto produce la lista:

$[\text{despejado}(a), \text{despejado}(b), \text{despejado}(c), \text{despejado}(4), \text{en}(a, 1), \text{en}(b, 3), \text{en}(c, 2)]$

4. Ahora podemos ejecutar la acción  $\text{mover}(a, 1, b)$ , con lo que la meta plantada se satisface. El plan encontrado es:

$[\text{mover}(c, a, 2), \text{mover}(a, 1, b)]$

Este estilo de razonamiento se conoce como *análisis medios-fines*. Observen que el ejemplo planteado el plan se encontró directamente, sin necesidad de reconsiderar (no hicimos *backtracking* en ningún momento). Esto ilustra como el proceso de razonar sobre el efecto de las acciones y las metas guían la planeación en una dirección adecuada. Desafortunadamente, no siempre se puede evitar la reconsideración. De hecho, ocurre lo contrario: la explosión combinatoria y la búsqueda son típicas en la planeación.

El principio de planeación por análisis medios-fines se ilustra en la figura 33. Puede plantearse como sigue: Para resolver una lista de Metas a partir de un Estado<sub>0</sub> inicial, que lleven a un Estado<sub>f</sub> final, hacer:

Si todas las Metas son verdaderas en Estado<sub>0</sub>, entonces Estado<sub>f</sub> = Estado<sub>0</sub>. En cualquier otro caso:

1. Seleccionar una Meta no solucionada en Metas.
2. Encontrar una Acción que agregue Meta al estado actual.
3. Hacer posible Acción resolviendo can para obtener el estado intermedio Estado<sub>1</sub>.
4. Aplicar la Acción en el estado Estado<sub>1</sub> para obtener el estado intermedio Estado<sub>2</sub> donde Meta se cumple.
5. Resolver Metas en el estado Estado<sub>2</sub> para llegar a Estado<sub>f</sub>.

El código del planeador medios fines es como sigue:

```

1  plan( Estado, Metas, [], Estado) :-
2      satisfecho( Estado, Metas).
3
4  plan(Estado, Metas, Plan, EstadoFinal) :-
5      append( PrePlan, [Accion | PostPlan], Plan),
6      seleccionar( Estado, Metas, Meta),
7      lograr( Accion, Meta),
8      can( Accion, Condicion),
9      plan( Estado, Condicion, PrePlan, EstadoInter1),
10     aplicar( EstadoInter1, Accion, EstadoInter2),
11     plan( EstadoInter2, Metas, PostPlan, EstadoFinal).
12
13     satisfecho( _, []).
14
15     satisfecho(Estado, [Meta | Metas]) :-
16         member(Meta, Estado),
17         satisfecho(Estado, Metas).
18
19     seleccionar(Estado, Metas, Meta) :-
20         member(Meta, Metas),
21         not(member(Meta, Estado)).
22
23     lograr(Accion, Meta) :-
24         agregar(Accion, Metas),
25         member(Meta, Metas).
26
27     aplicar(Estado, Accion, NewEstado) :-
28         borrar(Accion, ListaBorrar),
29         borrar_todos(Estado, ListaBorrar, Estado1), !,
30         agregar(Accion, ListaAgregar),
31         append(ListaAgregar, Estado1, NewEstado).
32
33     borrar_todos([], _, []).
34
35     borrar_todos([X | L1], L2, Diff) :-
36         member(X, L2), !,
37         borrar_todos(L1, L2, Diff).
38
39     borrar_todos([X | L1], L2, [X | Diff]) :-
40         borrar_todos(L1, L2, Diff).

```

Para invocar al planeador, ejecutamos en Prolog la siguiente meta:

```

1  ?- estado1(E), metas1(M), plan(E,M,P,Efinal).
2  E = [despejado(2), despejado(4), despejado(b),
3      despejado(c), en(a, 1), en(b, 3), en(c, a)],
4  M = [en(a, b)],
5  P = [mover(c, a, 2), mover(a, 1, b)],
6  Efinal = [en(a, b), despejado(1), en(c, 2), despejado(a),
7          despejado(4), despejado(c), en(b, 3)]

```

### 11.3 METAS PROTEGIDAS

Consideren ahora la siguiente llamada a plan/4:

```

1  ?- estado1(E), plan(E,[en(a,b),en(b,c)],Plan,_).
2  E = [despejado(2), despejado(4), despejado(b),
3       despejado(c), en(a, 1), en(b, 3), en(c, a)],
4  Plan = [mover(b, 3, c),
5          mover(b, c, 3),
6          mover(c, a, 2),
7          mover(a, 1, b),
8          mover(a, b, 1),
9          mover(b, 3, c),
10         mover(a, 1, b)]

```

Aunque el plan resultante cumple con su cometido, no es precisamente elegante. De hecho, existe un plan de tres movimientos para lograr las metas de este caso! Esto se debe a que el mundo de los bloques es más complejo de lo que parece, debido a la combinatoria. En este problema, el planeador tiene acceso a más opciones entre diferentes acciones que tienen sentido bajo el análisis medios-fines. Más opciones, significa mayor complejidad combinatoria.

Regresemos al ejemplo, lo que sucede es que el planeador persigue diferentes metas en diferentes etapas de la construcción del plan. Por ejemplo:

mover(b,3,c)	satisfacer en(b,c)
mover(b,c,3)	satisfacer clear(c) y ejecutar siguiente acción
mover(c,a,2)	satisfacer clear(a) y mover(a,1,b)
mover(a,1,b)	satisfacer on(a,b)
mover(a,b,1)	satisfacer clear(b) y mover(b,3,c)
mover(b,3,c)	satisfacer en(b,c) otra vez
mover(a,1,b)	satisfacer en(a,b) otra vez

Lo que esta tabla muestra es que a veces el planeador destruye metas que ya había satisfecho. El planeador logra fácilmente satisfacer una de las dos metas planteadas, en(b,c) pero la destruye al buscar como satisfacer la otra meta en(a,b). Lo peor es que esta forma desorganizada de seleccionar las metas, puede incluso llevar al fracaso en la búsqueda del plan, como en el siguiente ejemplo:

```

1  ?- estado1(E), plan(E,[despejado(2), despejado(3)], Plan, _).
2  ERROR: Out of local stack

```

Hagan un trace de esta corrida, para saber porque la meta falla.

Una idea evidente para evitar este comportamiento en nuestro planeador, es mantener una lista de metas protegidas, de forma que las acciones que destruyen estas metas no puedan ser seleccionadas. De forma que el planeador medios-fines con metas protegidas se define como:

```

1  plan_metas_protegidas(EstadoInicial, Metas, Plan, EstadoFinal) :-
2      plan_mp(EstadoInicial, Metas, [], Plan, EstadoFinal).
3
4  plan_mp(Estado, Metas, _, [], Estado) :-

```

```

5   satisfecho(Estado, Metas).
6
7   plan_mp(Estado, Metas, Protegido, Plan, EstadoFinal) :-
8     append( PrePlan, [Accion | PostPlan], Plan),
9     seleccionar( Estado, Metas, Meta),
10    lograr( Accion, Meta),
11    precond( Accion, Condicion),
12    preservar(Accion, Protegido),
13    plan_mp( Estado, Condicion, Protegido, PrePlan,
14            EstadoInter1),
15    aplicar( EstadoInter1, Accion, EstadoInter2),
16    plan_mp( EstadoInter2, Metas, [Meta|Protegido],
17            PostPlan, EstadoFinal).
18
19    preservar(Accion, Metas) :-
20      borrar(Accion, ListaBorrar),
21      not( (member(Meta, ListaBorrar),
22            member(Meta, Metas))).

```

De forma que si ejecutamos la consulta:

```

1   ?- estado1(E), plan_metas_protegidas(E, [despejado(2),
2     despejado(3)], P, _).
3   E = [despejado(2), despejado(4), despejado(b),
4     despejado(c), en(a, 1), en(b, 3), en(c, a)],
5   P = [mover(b, 3, 2), mover(b, 2, 4)]

```

obtenemos una solución, aunque sigue sin ser la mejor. Un sólo movimiento mover(b, 2, 4) era necesario para cumplir con las metas planeadas.

Los planes innecesariamente largos son resultado de la estrategia de búsqueda usada por nuestro planeador.

## 11.4 ASPECTOS PROCEDIMENTALES DE LA BÚSQUEDA EN AMPLITUD

Los planeadores implementados usan esencialmente una estrategia de búsqueda primero en profundidad, pero no por completo. Para poder estudiar lo que está pasando, debemos poner atención al orden en que se generan los planes candidatos. La meta

```

1   append(PrePlan, [Accion|PostPlan], Plan)

```

es central en este aspecto. La variable Plan no está instanciada cuando esta meta es alcanzada. El predicado append/3 genera al reconsiderar, candidatos alternativos para PrePlan en el siguiente orden:

```

1   PrePlan = [];
2   PrePlan = [_];
3   PrePlan = [_,_];
4   PrePlan = [_,-,-];
5   ...

```

Candidatos cortos para PrePlan son los primeros. PrePlan establece una condición para Accion. Esto permite encontrar una acción cuya condición puede satisfacerse por un plan tan corto como sea posible (simulando búsqueda primero en amplitud). Por otra parte, la lista candidato para PostPlan está totalmente no instanciada, y por tanto su longitud es ilimitada. Por tanto, la estrategia de búsqueda resultante es globalmente primero en profundidad, y localmente primero en amplitud. Con respecto al encadenamiento hacia adelante de las acciones que se agregan al plan emergente, se trata de una búsqueda primero en profundidad. Cada acción es validada por un PrePlan, este plan es por otra parte, buscado primero en amplitud.

Una forma de minimizar la longitud de los planes es forzar al planeador, en su parte de búsqueda en amplitud, de forma que los planes cortos sean considerados antes que los largos. Podemos imponer esta estrategia embebiendo nuestro planificador en un procedimiento que genere planes candidatos ordenados por tamaño creciente. Por ejemplo:

```

1  plan_primeros_amplitud(Estado, Metas, Plan, EstadoFinal) :-
2      candidato(Plan),
3      plan(Estado, Metas, Plan, EstadoFinal).
4
5
6  candidato([]).
7
8  candidato([Primero|Resto]) :-
9      candidato(Resto).
```

El mismo efecto puede lograrse de manera más elegante, insertando el generador de planes directamente en el procedimiento plan/4 de forma que:

```

1  plan_metas_protegidas_amplitud(EstadoInicial, Metas, Plan,
2      EstadoFinal) :-
3      plan_mp_amplitud(EstadoInicial, Metas, [], Plan,
4      EstadoFinal).
5
6  plan_mp_amplitud(Estado, Metas, _, [], Estado) :-
7      satisfecho(Estado, Metas).
8
9  plan_mp_amplitud(Estado, Metas, Protegido, Plan, EstadoFinal) :-
10     append(Plan, _, _),
11     append( PrePlan, [Accion | PostPlan], Plan),
12     seleccionar( Estado, Metas, Meta),
13     lograr( Accion, Meta),
14     precond( Accion, Condicion),
15     preservar(Accion, Protegido),
16     plan_mp_amplitud( Estado, Condicion, Protegido, PrePlan,
17         EstadoInter1),
18     aplicar( EstadoInter1, Accion, EstadoInter2),
19     plan_mp_amplitud( EstadoInter2, Metas, [Meta|Protegido],
20         PostPlan, EstadoFinal).
```

Y por tanto podemos volver a computar la meta original, encontrando esta vez el plan más corto:

```

1  ?- estado1(E),
2      plan_metas_protegidas_amplitud(E,[despejado(2),
3                                          despejado(3)], Plan,_).
4  E = [despejado(2), despejado(4), despejado(b),
5        despejado(c), en(a, 1), en(b, 3), en(c, a)],
6  Plan = [mover(b, 3, 4)]

```

Este resultado es óptimo, sin embargo la meta:

```

1  ?- estado1(E),
2      plan_metas_protegidas_amplitud(E,[en(a,b), en(b,c)], Plan, _).
3  E = [despejado(2), despejado(4), despejado(b),
4        despejado(c), en(a, 1), en(b, 3), en(c, a)],
5  Plan = [mover(c, a, 2),
6          mover(b, 3, a),
7          mover(b, a, c),
8          mover(a, 1, b)]

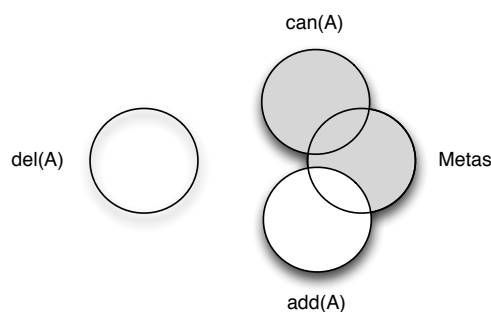
```

sigue siendo problemática. Este resultado se obtiene con y sin protección de metas siguiendo la estrategia primero en amplitud. El segundo movimiento del plan parece superfluo y aparentemente no tiene sentido. Investiguemos porque se le incluye en el plan y porque aún en el caso de la búsqueda primero en amplitud, el plan resultante está lejos del óptimo.

Dos preguntas son interesantes en este problema: ¿Qué razones encuentra el planeador para construir este curioso plan? y ¿Por qué el planeador no encuentra el plan óptimo e incluye la acción mover(b,3,a)? Atendamos la primer pregunta. La última acción mover(a,1,b) atiende la meta en(a,b). Los tres primeros movimientos están al servicio de cumplir las condiciones de esta acción, en particular la condición despejado(a). El tercer movimiento despeja a y una condición de este movimiento es en(b,a). Esto se cumple gracias al curioso segundo movimiento mover(b,3,a). Esto ilustra la clase de exóticos planes que pueden emerger durante un razonamiento medios-fines.

Con respecto a la segunda pregunta, ¿Por qué después de mover(c,a,2), el planeador no considera inmediatamente mover(b,3,c), lo que conduce a un plan óptimo? La razón es que el planeador estaba trabajando en la meta en(a,b) todo el tiempo. La acción que nos interesa es totalmente superflua para esta meta, y por lo tanto no es considerada. La cuarta acción logra en(a,b) y por pura suerte en(b,c)! Este último resultado no es una decisión planeada de nuestro sistema.

De lo anterior se sigue, que el procedimiento medios-fines, tal y como lo hemos implementado es **incompleto**, no sugiere todas las acciones relevantes para el proceso de planificación. Esto se debe a la localidad con que se computan las soluciones. Solo se sugerirán acciones relevantes para la meta actual del sistema. La solución al problema está en este enunciado: se debe permitir la interacción entre metas en el proceso de planificación. Antes de pasar al siguiente tema, consideren que al introducir la estrategia primero en amplitud para buscar planes más cortos, hemos elevando considerablemente el tiempo de computación necesario para hallar una solución.



**Figura 34:** Relaciones entre conjuntos de condiciones en la regresión de metas vía la acción  $A$ . El área sombreada representa las metas  $Metas_0$  resultado de la regresión. Observen que la intersección entre  $Metas$  y la lista borrar de  $A$  debe ser vacía.

### 11.5 REGRESIÓN DE METAS

Supongan que estamos interesados en una lista de metas  $Metas$  que se cumplen en cierto estado  $E$ . Sea el estado anterior a  $E$ ,  $E_0$  y la acción ejecutada en  $E_0$ ,  $A$ . ¿Qué metas  $Metas_0$  tienen que cumplirse en  $E_0$  para que  $Metas$  se cumpla en  $E$ ?

$Metas_0$  debe tener las siguientes propiedades:

1. La acción  $A$  debe ser posible en  $E_0$ , por lo que  $Metas_0$  debe implicar la condición para  $A$ .
2. Para cada meta  $M$  en  $Metas$ , se cumple que:
  - la acción  $A$  agrega  $M$ ; ó
  - $M \in Metas_0$  y  $A$  no borra  $M$ .

El cómputo para determinar  $Metas_0$  a partir de  $Metas$  y la acción  $A$  se conoce como **regresión de metas**. Por supuesto, sólo estamos interesados en aquellas acciones que agregan alguna meta  $M$  a  $Metas$ . Las relaciones entre varios conjuntos de metas y condiciones se ilustra en la figura 34

El mecanismo de regresión de metas puede usarse como planeador de la siguiente manera. Para satisfacer una lista de metas  $Metas$  a partir de un estado  $EstadoInicial$ , hacer: Si  $Metas$  se cumple en  $EstadoInicial$ , entonces el plan vacío es suficiente; en cualquier otro caso, seleccionar una meta  $M \in Metas$  y una acción  $A$  que agregue  $M$ ; entonces computar la regresión de  $Metas$  vía  $A$  obteniendo así  $NuevasMetas$  y buscar un plan para satisfacer  $NuevasMetas$  desde  $EstadoInicial$ .

El procedimiento puede mejorarse si observamos que algunas combinaciones de metas son imposibles. Por ejemplo  $en(a, b)$  y  $despejado(b)$  no pueden satisfacerse al mismo tiempo. Esto se puede formular vía la relación:

```
1 | imposible(Meta, Metas).
```

que indica que la  $Meta$  es imposible en combinación con las  $Metas$ . Para el caso del mundo de los bloques la incompatibilidad entre las metas se define como:

```
1 | imposible(en(X, X), ...).
2 |
```

```

3 imposible(en(X,Y), Metas) :-
4   member(despejado(Y),Metas)
5   ;
6   member(en(X,Y1),Metas), Y1 \== Y
7   ;
8   member(en(X1,Y),Metas) X1 \== X.
9
10 imposible(despejado(X),Metas) :-
11   member(en(_,X),Metas).

```

El resto del planeador es como sigue:

```

1 plan(Estado, Metas, []) :-
2   satisfecho(Estado, Metas).
3
4 plan(Estado, Metas, Plan) :-
5   append( PrePlan, [Accion], Plan),
6   seleccionar( Estado, Metas, Meta),
7   lograr(Accion, Meta),
8   precond(Accion, Condicion),
9   preservar(Accion, Metas),
10  regresion(Metas, Accion, MetasReg),
11  plan(Estado, MetasReg, PrePlan).
12
13 satisfecho(Estado, Metas) :-
14   borrar_todos(Metas,Estado,[]).
15
16 seleccionar(_, Metas, Meta) :-
17   member( Meta, Metas).
18
19 lograr( Accion, Meta) :-
20   agregar( Accion, Metas),
21   member( Meta, Metas).
22
23 borrar_todos( [], _, []).
24
25 borrar_todos( [X | L1], L2, Diff) :-
26   member( X, L2), !,
27   borrar_todos( L1, L2, Diff).
28
29 borrar_todos( [X | L1], L2, [X | Diff]) :-
30   borrar_todos( L1, L2, Diff).
31
32 preservar(Accion,Metas) :-
33   borrar(Accion,ListaBorrar),
34   not( (member(Meta,ListaBorrar),
35         member(Meta,Metas))).
36
37 regresion(Metas, Accion, MetasReg) :-
38   agregar(Accion, NuevasRels),
39   borrar_todos(Metas, NuevasRels, RestoMetas),
40   precond(Accion, Condicion),
41   agregarNuevo(Condicion,RestoMetas,MetasReg).
42
43 agregarNuevo([],L,L).

```

```

44
45  agregarNuevo([Meta|_],Metas,_) :-
46    imposible(Meta,Metas),
47    !,
48    fail.
49
50  agregarNuevo([X|L1],L2,L3) :-
51    member(X,L2), !,
52    agregarNuevo(L1,L2,L3).
53
54  agregarNuevo([X|L1],L2,[X|L3]) :-
55    agregarNuevo(L1,L2,L3).

```

Ahora es posible encontrar el plan óptimo de tres movimientos para el problema del mundo de los bloques:

```

?- estado1(E), plan(E,[en(a,b),en(b,c)],P).
E = [despejado(2), despejado(4), despejado(b),
     despejado(c), en(a, 1), en(b, 3), en(c, a)],
P = [mover(c, a, 2), mover(b, 3, c), mover(a, 1, b)]

```

## 11.6 COMBINANDO MEDIOS FINES CON PRIMERO EL MEJOR

Los planeadores construidos hasta ahora hacen uso de estrategias de búsqueda básicas: primero en profundidad, o primero en amplitud, o una combinación de ambas. Estas estrategias son totalmente desinformadas, en el sentido que no pueden usar información del dominio del problema para guiar su selección entre alternativas posibles. En consecuencia, estos planeadores son sumamente ineficientes, salvo en casos muy especiales. Existen diversas maneras de introducir una guía heurística, basada en el dominio del problema, en nuestros planeadores. Algunos lugares donde esto puede hacerse son:

- En la relación `seleccionar(Estado, Metas, Meta)` que decide el orden en que las metas serán procesadas. Por ejemplo, una guía en el mundo de los bloques es que las torres deben estar bien cimentadas, de forma que la relación `en/2` más arriba de la torre, debería resolverse al último (o primero en el planeador por regresión, que soluciona el plan en orden inverso). Otra guía es que las metas que ya se cumplen en el medio ambiente, deberían postergarse.
- En la relación `lograr(Accion, Meta)` que decide que acción alternativa será intentada para lograr una meta dada. Observen que nuestro planeador también genera alternativas al procesar `precond/2`. Por ejemplo, algunas acciones son “mejores” porque satisfacen más de una meta simultáneamente. También, con base en la experiencia, podemos saber que cierta condición es más fácil de satisfacer que otras.
- Decisiones acerca de qué conjunto de regresión de metas debe considerarse a continuación. Por ejemplo, seguir trabajando en el que parezca más fácil de resolver, buscando así el plan más corto.

Esta última idea muestra como podemos imponer una estrategia primero el mejor en nuestro planeador. Esto implica computar un estimado heurístico de la dificultad de los diversos conjuntos de regresión de metas alternativas, para expandir el más promisorio.

Recuerden que para usar este tipo de estrategia es necesario especificar:

1. Una relación  $s/3$  entre nodos del espacio de búsqueda:  $s(\text{Nodo}_1, \text{Nodo}_2, \text{Costo})$ .
2. Los nodos meta en el espacio:  $\text{meta}(\text{Nodo})$ .
3. Una función heurística de la forma  $h(\text{Nodo}, \text{Hestimado})$ .
4. El nodo inicial de la búsqueda.

Una forma de definir estos requisitos es asumir que los conjuntos de regresión de metas son nodos en el espacio de búsqueda. Esto es, en el espacio de búsqueda hará una liga entre  $\text{Metas}_1$  y  $\text{Metas}_2$  si existe una acción  $A$  tal que:

1.  $A$  agrega alguna meta  $\in \text{Metas}_1$ .
2.  $A$  no destruye ninguna meta  $\in \text{Metas}_1$ .
3.  $\text{Metas}_2$  es el resultado de la regresión de  $\text{Metas}_1$  a través de  $A$ , tal y como definimos en nuestro planeador anterior:  $\text{regresion}(\text{Metas}_1, A, \text{Metas}_2)$ .

Por simplicidad, asumiremos que todas las acciones tienen el mismo costo, y en consecuencia asignaremos  $\text{Costo} = 1$  en todas las ligas del espacio de búsqueda. Por lo que la relación  $s/3$  se define como:

```

1  s(Metas1, Metas2) :-
2  member(Meta, Metas1),
3  lograr(Accion, Meta),
4  precondition(Accion, Cond),
5  preservar(Accion, Metas1),
6  regresion(Metas1, Accion, Metas2).
```

Cualquier conjunto de metas que sea verdadero en la situación inicial de un plan, es un nodo meta en el espacio de búsqueda. El nodo inicial de la búsqueda es la lista de metas que el plan debe lograr.

Aunque la representación anterior tiene todos los elementos requeridos, tiene un pequeño defecto. Esto se debe a que nuestra búsqueda primero el mejor encuentra un camino solución como una secuencia de estados y no incluye acciones entre los estados. Por ejemplo, la secuencia de estados (listas de metas) para lograr  $\text{en}(a, b)$  en el estado inicial que hemos estado usando es:

```

1  [ [despejado(c), despejado(2), en(c, a), despejado(b), en(a, 1)]
2  [despejado(a), despejado(b), en(a, 1)]
3  [en(a, b)] ]
```

El primer estado es verdadero por la situación inicial, el segundo es resultado de la acción  $\text{mover}(c, a, 2)$  y el tercero es resultado de la acción  $\text{mover}(a, 1, b)$ .

Observen que la búsqueda primero el mejor regresa el camino solución en el orden inverso. En nuestro caso es una ventaja, porque los planes son construidos en

la regresión hacía atrás, así que al final obtendremos la secuencia de acciones en el orden correcto. Sin embargo, es raro no tener mención explícita a las acciones en el plan, aunque puedan reconstruirse de las diferencias entre listas de metas. Podemos incluir las acciones en el camino solución fácilmente, basta con agregar a cada estado la acción que se sigue de él. De forma que los nodos del espacio de búsqueda tendrán la forma:

```
1 Metas -> Acción
```

Su implementación detallada es la siguiente:

```
1 :- op(300,xfy, ->).
2
3 s(Metas -> AccSiguiente, MetasNuevas -> Accion, 1) :-
4   member(Meta, Metas),
5   lograr(Accion, Meta),
6   precond(Accion, Cond),
7   preservar(Accion, Metas),
8   regresion(Metas, Accion, MetasNuevas).
9
10 meta(Metas -> Accion) :-
11   inicio(Estado),
12   satisfecho(Estado, Metas).
13
14 h(Metas -> Accion, H) :-
15   inicio(Estado),
16   borrar_todos(Metas, Estado, Insatisfecho),
17   length(Insatisfecho, H).
18
19 inicio([en(a,1), en(b,3), en(c,a), despejado(b), despejado(c),
20   despejado(2), despejado(4)]).
```

Ahora podemos usar nuestro viejo buscador primero el mejor:

```
1 primeroMejor(Inicio, Solucion) :-
2   expandir([], hoja(Inicio, 0/0), 9999, _, si, Solucion).
3
4
5   %% expandir(Camino, Arbol, Umbral, Arbol1, Solucionado, Solucion)
6   %% Camino es el recorrido entre Inicio y el nodo en Arbol
7   %% Arbol1 es Arbol expandido bajo el Umbral
8   %% Si la meta se encuentra, Solucion guarda el camino solución
9   %% y Solucionado = si
10
11 % Caso 1: la hoja con Nodo es una meta, construye una solución
12
13 expandir(Camino, hoja(Nodo, _), _, _, si, [Nodo|Camino]) :-
14   meta(Nodo).
15
16 % Caso 2: una hoja con f-valor menor o igual al Umbral
17 % Generar sucesores de Nodo y expandirlos bajo el Umbral
18
19 expandir(Camino, hoja(Nodo, F/G), Umbral, Arbol1, Solucionado, Sol) :-
```

```

20     F =< Umbral,
21     (bagof( M/C, (s(Nodo,M,C), not(member(M,Camino))), Succ),
22     !,      % Nodo tiene sucesores
23     listaSuccs(G,Succ,As), % Encuentras subárboles As
24     mejorF(As,F1),      % f-value of best successor
25     expandir(Camino,arbol(Nodo,F1/G,As),Umbral,Arbol1,
26     Solucionado,Sol)
27     ;
28     Solucionado = nunca % Nodo no tiene sucesores
29     ).
30
31 % Caso 3: Nodo interno con f-valor menor al Umbral
32 % Expandir el subárbol más promisorio con cuyo
33 % resultado, continuar/7 decidirá como proceder
34
35 expandir(Camino,arbol(Nodo,F/G,[A|As]),Umbral,Arbol1,
36 Solucionado,Sol) :-
37     F =< Umbral,
38     mejorF(As,MejorF), min(Umbral,MejorF,Umbral1),
39     expandir([Nodo|Camino],A,Umbral1,A1,Solucionado1,Sol),
40     continuar(Camino,arbol(Nodo,F/G,[A1|As]),Umbral,Arbol1,
41     Solucionado1,Solucionado,Sol).
42
43 % Caso 4: Nodo interno con subárboles vacío
44 % Punto muerto, el problema nunca será resuelto
45
46 expandir(_,arbol(_,_,[ ]),_,_ ,nunca,_ ) :- !.
47
48 % Caso 5: f-valor mayor que el Umbral
49 % Arbol no debe crecer
50
51 expandir(_,Arbol,Umbral,Arbol,no,_ ) :-
52     f(Arbol,F), F > Umbral.
53
54 %% continuar(Camino,Arbol,Umbral,NuevoArbol,SubarbolSolucionado,
55 %%           ArbolSolucionado,Solucion)
56
57 % Caso 1: el subarbol y el arbol están solucionados
58 % la solución está en Sol
59
60 continuar(_,_ ,_ ,_ ,si,si,Sol).
61
62 continuar(Camino,arbol(Nodo,F/G,[A1|As]),Umbral,Arbol1,no,
63 Solucionado,Sol) :-
64     insertarArbol(A1,As,NAs),
65     mejorF(NAs,F1),
66     expandir(Camino,arbol(Nodo,F1/G,NAs),Umbral,Arbol1,
67 Solucionado,Sol).
68
69 continuar(Camino,arbol(Nodo,F/G,[_ |As]),Umbral,Arbol1,nunca,
70 Solucionado,Sol) :-
71     mejorF(As,F1),
72     expandir(Camino,arbol(Nodo,F1/G,As),Umbral,Arbol1,
73 Solucionado,Sol).
74

```

```

75  %% listaSuccs(G0,[Nodo1/Costo1, ...], [hoja(MejorNodo,MejorF/G),
76  ...])
76  %%% hace una lista de árboles sucesores ordenados por F-valor
77
78  listaSuccs(_,[],[]).
79
80  listaSuccs(G0,[Nodo/C|NCs],As) :-
81      G is G0 + C,
82      h(Nodo,H), % Heuristic term h(N)
83      F is G + H,
84      listaSuccs(G0,NCs,As1),
85      insertarArbol(hoja(Nodo,F/G),As1,As).
86
87  %%%Inserta A en una lista de arboles As preservando el orden por f-
88  valor
89
90  insertarArbol(A,As,[A|As]) :-
91      f(A,F), mejorF(As,F1),
92      F =< F1, !.
93
94  insertarArbol(A,[A1|As], [A1|As1]) :-
95      insertarArbol(A,As,As1).
96
97  %%%Extraer f-valores
98
99  f(hoja(_,F/_),F). % f-valor de una hoja
100 f(arbol(_,F/_,_),F). % f-valor de un árbol
101
102 mejorF([A|_],F) :- f(A, F).
103 mejorF([], 9999).
104
105 min(X,Y,X) :- X =< Y, !.
106 min(_,Y,Y).

```

De forma que podemos procesar el plan con la siguiente llamada:

```

1  ?- primeroMejor([en(a,b), en(b,c)] -> stop, Plan).
2  Plan = [[despejado(2), en(c, a), despejado(c), en(b, 3),
3          despejado(b), en(a, 1)]->mover(c, a, 2),
4          [despejado(c), en(b, 3), despejado(a), despejado(b),
5          en(a, 1)]->mover(b, 3, c),
6          [despejado(a), despejado(b), en(a, 1), en(b, c)]
7          ->mover(a, 1, b),
8          [en(a, b), en(b, c)]->stop]

```

La acción nula stop es necesaria pues todos los nodos deben incluir una acción. Aunque la heurística usada es simple, el programa debe ser más rápido que las versiones anteriores. Eso si, el precio a pagar es una mayor utilización de la memoria, debido a que debemos mantener el conjunto de alternativas competitivas.

## 11.7 VARIABLES Y PLANES NO LINEALES

A manera de comentario final, consideraremos dos casos que pueden mejorar la eficiencia de los planificadores construidos en esta sesión. El primer caso consiste en permitir que las acciones y las metas contengan variables no instanciadas; el segundo caso es considerar que los planes no son lineales.

### 11.7.1 Acciones y metas no instanciadas

Las variables que ocurren en nuestros planeadores están siempre instanciadas. Esto se logra, por ejemplo en la relación `precond/2` cuyo cuerpo incluye la meta `block(Block)` entre otras. Este tipo de meta hace que `Block` siempre esté instanciada. Esto puede llevar a la generación de numerosos movimientos alternativos irrelevantes. Por ejemplo, cuando al planeador se le plantea como meta `despejar(a)`, éste utiliza `lograr/2` para generar movimientos que satisfagan `despejado(a)`:

```
1 mover(De,a,A)
```

Entonces se computan las condiciones necesarias para ejecutar esta acción:

```
1 precond(mover(De,a,A),Cond)
```

Lo cual fuerza, al reconsiderar, varias instancias alternativas para `De` y `A`:

```
1 mover(b,a,1)
2 mover(b,a,2)
3 mover(b,a,3)
4 mover(b,a,4)
5 mover(b,a,c)
6 mover(b,a,1)
7 mover(b,a,2)
```

Para hacer más eficiente este paso del planeador, es posible permitir variables no instanciadas en las metas. Para el ejemplo del mundo de los bloques, las condiciones de mover serían definidas como:

```
1 precond(mover(Bloque,De,A),
2         [despejado(Bloque),despejado(A),en(Bloque,De)]).
```

Si reconsideramos con esta nueva definición la situación inicial, la lista de condiciones computadas sería:

```
1 [despejado(Bloque),despejado(A),en(Bloque,a)]
```

Observen que esta lista de metas puede ser satisfecha inmediatamente en la situación inicial de nuestro ejemplo si `Bloque/c` y `A/2`. Esta mejora en eficiencia se logra postergando la decisión de cómo instanciar las variables, al momento en que ya se cuenta con más información para ello.

Este ejemplo ilustra el poder de la representación con variables, pero el precio a pagar es una mayor complejidad. Para empezar, nuestro intento por definir `precond`

para mover/3 es erróneo, pues permite movimientos como mover(c, a, c), que da como resultado que !el bloque c esté en el bloque c! Esto podría arreglarse si especificáramos que De y A deben ser diferentes:

```

1 | precondition(mover(Bloque, De, A),
2 |               [despejado(Bloque), despejado(A), en(Bloque, De),
3 |               diferente(Bloque, A), diferente(De, A),
4 |               diferente(Bloque, De)]).
```

donde diferente/2 significa que los dos argumentos no denotan al mismo objeto Prolog. Una condición como estas, no depende del estado del problema, de forma que no puede volverse verdadero mediante acción alguna, pero debe verificarse evaluando el predicado correspondiente. Una manera de manejar estas cuasi-metas es agregar al predicado satisfecho/2 la siguiente cláusula:

```

1 | satisfecho(Estado, [Meta|Metas]) :-
2 |     satisface(Meta),
3 |     satisfecho(Estado, Metas).
```

De forma que debemos definir también:

```

1 | satisface(diferente(X, Y))
```

Tal relación tiene éxito si X y Y no se corresponden. Si X y Y son lo mismo, la condición es falsa. Este caso debería tratarse con imposible, pues la condición deberá seguir siendo falsa, sin importar las acciones que serán adoptadas en el plan. En otro caso, estamos ante falta de información y satisface se debería postergar.

### 11.7.2 Planes no lineales

Un problema con nuestro planeador es que considera todos los posibles ordenes de las acciones, aún cuando las acciones son completamente independientes. Consideren el problema ilustrado en la figura 35, donde la meta es construir dos pilas de bloques que están de antemano bien separados. Las dos pilas puede construirse independientemente con los siguientes planes:

```

1 | Plan1 = [mover(b, a, c), mover(a, 1, b)]
2 | Plan2 = [mover(e, d, f), mover(d, 8, e)]
```

El punto importante aquí es que estos planes no interaccionan entre ellos, de forma que el orden de las acciones sólo es relevante dentro de cada plan. Tampoco importa si se ejecuta primero Plan1 o Plan2 y es incluso posible ejecutarlos de manera alternada, por ejemplo:

```

1 | [mover(b, a, c), mover(e, d, f), mover(d, 8, e), mover(a, 1, b)]
```

Sin embargo, nuestro planeador considerará las 24 permutaciones posibles de las cuatro acciones, aunque existan solo 4 alternativas: 2 permutaciones para cada uno de los planes. El problema se debe a que el planeador insiste en el orden total de las acciones en el plan. Una mejora se lograría si, en los casos donde el orden no es

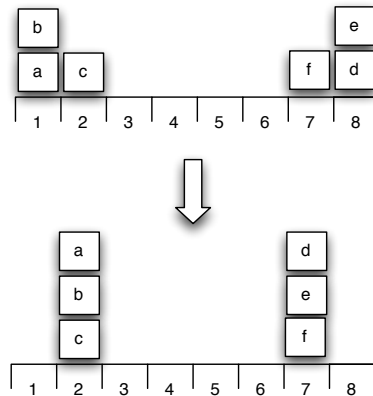


Figura 35: Una tarea de planeación con dos planes independientes

importante, la precedencia entre las acciones se mantiene indefinida. Entonces nuestros planes serán conjuntos de acciones parcialmente ordenadas. Los planeadores que aceptan este tipo de representación se conocen como planeadores **no lineales**.

Consideremos nuevamente el ejemplo de la figura 35. Analizando las metas en  $(a, b)$  y en  $(b, c)$  el planeador no lineal concluye que las siguientes dos acciones son necesarias en el plan:

```
1 | M1 = mover(a,X,b)
2 | M2 = mover(b,Y,c)
```

No hay otra forma de resolver ambas metas, pero el orden de estas acciones es aún indeterminado. Ahora consideren las condiciones de ambas acciones. La condición de mover  $(a, X, b)$  incluye  $dejado(a)$ , la cual no se satisface en la situación inicial, por lo que necesitamos una acción de la forma:

```
1 | M3 = mover(Bloque,a,A).
```

que precede a M1. Ahora tenemos una restricción en el orden de las acciones:

```
1 | antes(M3,M1)
```

Ahora revisamos si M3 y M1 pueden ser el mismo movimiento. Como este no es el caso, el plan tendrá que incluir tres movimientos. Ahora el planeador debe preguntarse si hay una permutación de  $[M1, M2, M3]$  tal que M3 preceda a M1, tal que la permutación es ejecutable en el estado inicial del problema y las metas se cumplen en el estado resultante. Dadas las restricciones de orden anteriores tres permutaciones de seis, cumplen con los requisitos:

```
1 | [M3,M1,M2]
2 | [M3,M2,M1]
3 | [M2,M3,M1]
```

Y de estas permutaciones, solo la del medio cumple con el requisito de ser ejecutable bajo la sustitución  $Bloque/c, A/2, X/1, Y/3$ . Como se puede intuir, la complejidad

computacional no puede ser evitada del todo por un planeador no lineal, pero puede ser aliviada considerablemente.