8 PLANEACIÓN

La planeación [46, 1, 55] es un tema de interés tradicional en la IA, que involucra razonar acerca de los efectos de las acciones y la secuencia en que éstas se aplican para lograr un efecto acumulativo deseado. En esta sesión desarrollaremos planificadores simples para ilustrar los principios de la planeación.

La Figura 8.1 muestra una tarea de planeación muy utilizada en IA, el mundo de los bloques. De hecho, ya hemos utilizado este escenario al hablar acerca de la lógica de primer orden. En este capítulo, el problema se usará para introducir una representación de conocimiento que nos permita razonar explícitamente acerca de los efectos de las acciones que nuestro programa puede ejecutar.

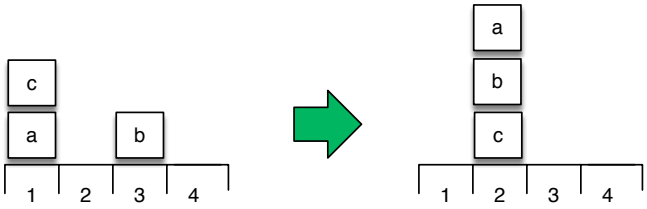

Figura 8.1: *El mundo de bloques, revisitado.*

Aunque el mundo de los bloques puede verse como un caso de los criticados mundos sintéticos usados en IA, debe observarse que encontrar un plan óptimo en estos ambientes es un problema NP-duro [58]. Además, la configuración original del problema, puede redefinirse para incluir aquellas características deseables del medio ambiente, que no están presentes en su formulación original. Por ejemplo: no determinismo, dinamismo, observación parcial, etc. La sección 2.3, discute que características pueden estar presentes al considerar un medio ambiente.

Si coincidimos con Patrick Winston [1], en que el sujeto de estudio de la IA son los algoritmos construidos a partir de restricciones expuestas por representaciones que dan soporte a modelos del razonamiento, la percepción y la acción; este capítulo introduce en nuestro curso, los componentes de acción y, de manera muy limitada, el de percepción.

Veamos un ejemplo de como las técnicas de representación y razonamiento basados en programas definitivos, pueden ser usadas para hacer algo que parece planeación, pero no lo es. ¿Cómo puedo escribir un programa que nos diga como salir del laberinto de la figura 8.2? Adoptando el *modus operandi* seguido hasta ahora, tal cuestión implica decidir qué representación del laberinto debo adoptar, para explotar alguno de los métodos de razonamiento que hemos visto.

El laberinto puede representarse como un grafo de conectividad, que establece que posiciones son contiguas. Usaremos los predicados `conecta/2` y `conectado/2` para ello. El segundo, simplemente establece que el primero es una relación simétrica:

```
1 %%%conectado/2
2 %%%  computa si la Pos1 esta conectada con la Pos2
3
4 conectado(Pos1, Pos2) :- conecta(Pos1, Pos2).
5 conectado(Pos1, Pos2) :- conecta(Pos2, Pos1).
6
7 %%%Datos del laberinto (adyacencia)
```

1 La definición que sigue se puede encontrar en el curso de Winston sobre Introducción a la IA en MIT: https://ai6034.mit.edu/wiki/index.php?title=Main_Page

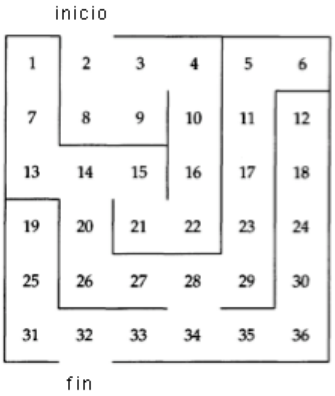

Figura 8.2: Un laberinto.

```
8
9  conecta(inicio,2).
10 conecta(1,7).
11 conecta(2,8).
12 conecta(2,3). %%%agrega varias soluciones.
13 conecta(3,4).
14 conecta(3,9).
```

Observen una de las ventajas de la representación en primer orden adoptada: `conectado/2` reduce el tamaño de nuestra base de conocimientos a la mitad. Si la representación fuese proposicional, la simetría de `conecta/2` tendría que expresarse por extensión, es decir, incluyendo en la base de conocimientos todas las relaciones de `conecta/2` en ambos sentidos. La relación `conectado/2` reduce media base de conocimiento en una regla (línea 5).

El caso es que ahora sabemos que `inicio` está conectado con 2, y 2 con 3, etc. Por lo que podemos hacer una búsqueda recursiva para encontrar un camino que me lleve de `inicio` a `fin` en el laberinto representado como un grafo de conectividad. Dada mi posición actual, podré moverme a una posición que satisface `conectado/2` y, para evitar ciclos infinitos, que no haya sido visitada anteriormente:

```
1  %%%sol/0
2  %%% computa los caminos de solución para el laberinto.
3
4  sol :- camino([inicio], Sol), write(Sol).
5
6  %%%camino/2
7  %%%  si el camino llego al fin, regresa el camino de fin a inicio.
8  %%%  en caso contrario busca ir a una posición que no se haya visitado
9  %%%  anteriormente (\+ miembro)
10
11 camino([fin|RestoDelCamino], [fin|RestoDelCamino]).
12 camino([PosActual|RestoDelCamino], Sol) :-
13   conectado(PosActual,PosSiguiente),
14   \+ miembro(PosSiguiente, RestoDelCamino),
15   camino([PosSiguiente,PosActual|RestoDelCamino], Sol).
```

Como suele ser el caso, yo como programador solo estoy definiendo lo que es un camino solución en el laberinto: Una secuencia de pasos conectados de `inicio` a `fin`, sin volver sobre mis pasos (línea 14). La solución se encuentra vía la aplicación de la regla de resolución-SLD de Prolog, es decir, por una búsqueda en profundidad del camino solución, con reconsideración. La relación `miembro/2` está definida en el programa; si no quieren definir su versión de ella, pueden usar `member/2`, que está definida en todos los Prolog. La consulta a este programa es como sigue:

```
1 ?- sol.
2 [fin,32,33,34,28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,inicio]
3 true ;
```

```
4 [fin,32,33,34,28,27,26,20,14,15,21,22,16,10,4,3,2,inicio]
5 true ;
6 false.
```

Aunque los dos caminos encontrados por nuestro programa, pueden verse como secuencias de decisiones que nos sacan del laberinto, dificilmente pueden verse como un plan ¿Porqué?

Un proceso de planeación suele verse como un razonamiento explícito acerca de los efectos de las acciones y la secuencia en que estas se aplican para lograr un efecto acumulativo dado [18]. Ghallab, Nau y Traverso [55] define la planeación como un proceso deliberativo abstracto y explícito, que elige y organiza acciones anticipando sus efectos; la meta es alcanzar una serie de objetivos predefinidos de la mejor manera posible. En este caso, ni las acciones, ni las metas son consideradas explícitamente. Tampoco hay una consideración sobre el mejor plan posible. En lo que sigue, definiremos procesos basados en búsqueda que si toman en cuenta estos factores, y por lo tanto, planean.

El capítulo está organizado de la siguiente manera: Primero se aborda un formalismo para definir las acciones con las que se construirán los planes y las metas del proceso. Posteriormente, abordaremos una estrategia medios-fines para la construcción de planes, que iremos refinando a lo largo del capítulo. La primer mejora a nuestro planificador será la consideración de metas protegidas. Luego explotaremos la búsqueda en amplitud para generar planes más cortos. Posteriormente, abordaremos la regresión basada en metas para guiar la búsqueda hacia atrás que lleva a cabo el planeador. Todos los planificadores abordados hasta ahora, son ciegos, es decir, no tienen información sobre el costo o beneficio de las acciones a considerar en el plan. La siguiente mejora es introducir esta información via una búsqueda primero el mejor. Terminaremos con algunas consideraciones sobre el uso de variables en los planes y la planeación no lineal.

8.1 ESTADOS, ACCIONES Y METAS

Para mayor claridad, asumiremos que el mundo de los bloques, nuestro medio ambiente, es observable y determinista. Esto es, las acciones de nuestro programa son los únicos factores de cambio; y todo cambio puede ser **percibido** por el programa. Como se ha mencionado, estas restricciones podrán relajarse posteriormente, de ser necesario.

Percepción

Ahora bien, una acción generalmente tiene efectos locales, en el sentido que no afecta todo el medio ambiente. Una buena representación de las acciones debería tomar esto en cuenta. Para facilitar razonar acerca de estos efectos focalizados de las acciones, un **estado** del medio ambiente será representado como una lista de relaciones válidas en un momento dado. Esta lista estado solo incluirá aquellas relaciones que son relevantes [2] para nuestro problema de planificación.

Estado

Ejemplo 8.1. *Para el caso del mundo de los bloques, las relaciones a considerar son en/2 y libre/1, con su semántica intuitiva. El estado del mundo de cubos de la izquierda, en la Figura 8.1, sería:*

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

Cada **acción** posible se define entonces en términos de las condiciones que deben observarse en el estado actual, para que ésta pueda ser ejecutada; y de sus efectos esperados. Específicamente:

Acción

- **Condiciones.** Una lista de las condiciones que debe satisfacerse, para que la acción pueda ejecutarse.

2 Qué es relevante, es la cuestión en el centro del llamado *frame problem*, identificado inicialmente por McCarthy y Hayes [81]. Aunque el problema fue resuelto originalmente por Reiter, Scherl y Levesque [109], discuten ésta y otras soluciones.

- **Agregar.** Una lista de observaciones que, se espera, ocurran después de ejecutarse la acción.
- **Borrar.** Una lista de observaciones que, se espera, dejen de ser verdaderas después de ejecutarse la acción.

Ejemplo 8.2. *En el dominio del mundo de los bloques, la única acción posible será mover/3. La definición completa de esta acción es como sigue:*

```
%%%Acción mover en mundo de bloques

precond(mover(Bloque,De,A),
  [libre(Bloque),libre(A),en(Bloque,De)]) :-
    bloque(Bloque),
    objeto(A),
    A\==Bloque,
    objeto(De),
    De\==A,
    Bloque\==De.

agregar(mover(Bloque,De,A),[en(Bloque,A),libre(De)]).

borrar(mover(Bloque,De,A),[en(Bloque,De),libre(A)]).
```

De manera que para poder mover un *Bloque* de la posición *De* a la posición *A*, es necesario que el *Bloque* y la posición *A* estén libres; y que el bloque *Bloque* esté en la posición *De*. El resto de *precond*/2 establece otro tipo de restricciones: que *Bloque* sea un bloque, y *A* y *De* sean objetos en el universo de discurso; que *A* sea diferente de *Bloque*; que se debe mover el bloque a una nueva posición (*A* es diferente de *De*); y no mover el bloque de si mismo (*Bloque* es diferente de *De*). Las definiciones de *agregar*/2 y *borrar*/2 completan la especificación de *mover*/3.

A diferencia de la búsqueda ciega para encontrar un camino solución en el laberinto de la introducción al capítulo, los procesos de planeación suelen requerir de información sobre el dominio de aplicación. A esta información se le suele conocer como **ontología** del problema, es decir, lo que sé del dominio sobre el cual quiero elaborar planes. En el caso del mundo de los bloques la ontología define lo que es un `objeto/1`, un `bloque/1` y un `lugar/1` en la mesa:

Ontología

Ejemplo 8.3. *La ontología del mundo de los bloques:*

```
%%%Ontología

objeto(X) :-  lugar(X); bloque(X).

bloque(a).
bloque(b).
bloque(c).

lugar(1).
lugar(2).
lugar(3).
lugar(4).
```

Nuestras **metas** serán representadas como una lista de observaciones que deberían darse luego de ejecutar el plan. Con estas definiciones es posible establecer el estado inicial y metas en el mundo de los bloques:

Meta

Ejemplo 8.4. *El escenario mostrado en la Figura 8.1 y la meta de colocar el bloque a en el bloque b, pueden representarse con las siguientes relaciones:*

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

metas([en(a,b)]).
```

Observen que la definición de las acciones, establece también el espacio de planes posibles, conocido como **espacio de planeación**. Ahora veremos como a partir de esta representación, es posible derivar los planes mediante un procedimiento conocido como análisis medios-fines.

Espacio de planeación

8.2 ANÁLISIS MEDIOS-FINES

Sean el estado y las metas iniciales los definidos en el ejemplo 8.4. El trabajo de un planeador consiste entonces en encontrar una secuencia de acciones que satisfagan las metas iniciales. Un planeador típico razonaría de la siguiente forma:

1. Encontrar una acción que satisfaga `en(a,b)`. Para ello usamos la relación *agregar*/2, encontrando que tal acción es de la forma `mover(a,De,b)`. Esta acción deberá formar parte de nuestro plan, pero no podemos ejecutarla inmediatamente dado nuestro estado inicial.

2. Hacer posible la ejecución de la acción `mover(a,De,b)`. Para ello usamos la relación *precond*/2 encontrando que, las condiciones para poder ejecutar esta acción son:

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

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

3. Volvemos a buscar en la relación *agregar*/2 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), despeado(A), en(Bloque,a) ]
```

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

   - Eliminamos del estado inicial las relaciones que la acción borra.
   - Incluímos las relaciones que la acción agrega al estado inicial del problema.

   esto produce la lista:

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

4. Ahora podemos ejecutar la acción `mover(a,1,b)`, con lo que la meta inicial se satisface. El plan encontrado es:

```
[ mover(c,a,2), mover(a,1,b) ]
```

Este estilo de razonamiento se conoce como **análisis medios-fines**. Observen que el ejemplo planteado el plan se encuentra 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.

Análisis medios-fines

El principio general de planeación por análisis medios-fines se ilustra en la figura 8.3. Puede describirse como sigue: Para resolver una lista de *Metas* a partir de un estado inicial Edo_0, y alcanzar un estado final Edo_f, hacer lo siguiente: Si todas las *Metas* son verdaderas en Edo_0, entonces $Edo_f = Edo_0$. En cualquier otro caso:

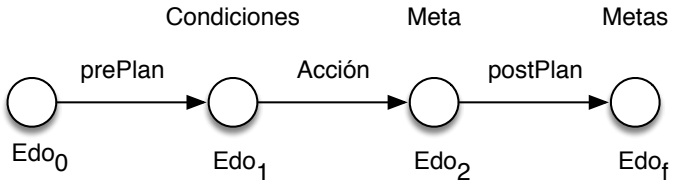

Figura 8.3: Análisis medios-fines

1. Seleccionar una *Meta* no solucionada en *Metas*.
2. Encontrar una *Accion* que agregue *Meta* al estado actual.
3. Hacer posible la ejecución de *Accion*, resolviendo su *Condicion* para obtener el estado intermedio Edo_1.
4. Aplicar la *Accion* en el estado Edo_1, para obtener el estado intermedio Edo_2 donde *Meta* se cumple.
5. Resolver *Metas* en el estado Edo_2 para llegar a Edo_f.

El código del planeador medios fines es como sigue:

```
%%%Planeador medios fines

plan(Edo,Metas,[],Edo) :-
  metas_satisfechas(Edo,Metas).

plan(Edo,Metas,Plan,EdoFinal)  :-
    append(PrePlan,[Accion|PostPlan],Plan),
    seleccionar(Edo,Metas,Meta),
    lograr(Accion,Meta),
    precond(Accion,Condicion),
    plan(Edo,Condicion,PrePlan,EdoInter1),
    aplicar(EdoInter1,Accion,EdoInter2),
    plan(EdoInter2,Metas,PostPlan,EdoFinal).

metas_satisfechas(_,[]).

metas_satisfechas(Edo,[Meta|Metas]) :-
    member(Meta,Edo),
    metas_satisfechas(Edo,Metas).

seleccionar(Edo,Metas,Meta)  :-
    member(Meta,Metas),
    not(member(Meta,Edo)).

lograr(Accion,Meta)  :-
    agregar(Accion,Metas),
    member(Meta,Metas).

aplicar(Edo,Accion,NewEdo)  :-
    borrar(Accion,ListaBorrar),
    borrar_todos(Edo,ListaBorrar,Edo1),!,
    agregar(Accion,ListaAgregar),
    append(ListaAgregar,Edo1,NewEdo).

% Borra de [X|L1] todos los elementos que aparecen en L2
% dando como resultado Diff.

borrar_todos([],_,[]).

borrar_todos([X|L1],L2,Diff)  :-
    member(X,L2),!,
    borrar_todos(L1,L2,Diff).

borrar_todos([X|L1],L2,[X|Diff])  :-
```

Como suele ser el caso, la implementación en Prolog del análisis medios fines ($plan/4$), es prácticamente una traducción directa de la descripción del proceso en castellano. El resto del código implementa los predicados auxiliares necesarios. Para invocar al planeador, ejecutamos en Prolog la siguiente meta:

```
?- estado(Edo0), metas(M), plan(Edo0,M,P,EdoF).
Edo0 = [libre(2), libre(4), libre(b), libre(c), en(a, 1), en(b, 3), en(c, a)],
M = [en(a, b)],
P = [mover(c, a, 2), mover(a, 1, b)],
EdoF = [en(a, b), libre(1), en(c, 2), libre(a), libre(4), libre(c), en(b, 3)]
```

8.3 METAS PROTEGIDAS

Consideren ahora la siguiente consulta a $plan/4$:

```
?- estado(Edo0), plan(Edo0,[en(a,b),en(b,c)],P,_).
Edo0 = [libre(2), libre(4), libre(b), libre(c), en(a, 1), en(b, 3), en(c, a)],
P = [ mover(b, 3, c), mover(b, c, 3), mover(c, a, 2), mover(a, 1, b),
       mover(a, b, 1), mover(b, 3, c), mover(a, 1, b) ]
```

Aunque el plan resultante cumple con su cometido, no es precisamente eficiente, ni 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.

Combinatoria

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

Acción	Objetivo medios-fines
$mover(b,3,c)$	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:

```
?- estado(E), plan(E,[libre(2), libre(3)], P, _).
ERROR: Out of local stack
```

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

Una manera de 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 al planear. El planeador medios-fines con metas protegidas se implementa de la siguiente manera:

Metas protegidas

```
%%%Medios fines con metas protegidas

plan_metas_protegidas(EdoInicial,Metas,Plan,EdoFinal):-
    plan_mp(EdoInicial,Metas,[],Plan,EdoFinal).

plan_mp(Edo,Metas,_,[],Edo) :-
```

```
7       metas_satisfechas(Edo,Metas).
8
9   plan_mp(Edo,Metas,Protegido,Plan,EdoFinal) :-
10      append(PrePlan,[Accion|PostPlan],Plan),
11      seleccionar(Edo,Metas,Meta),
12      lograr(Accion,Meta),
13      precond(Accion,Condicion),
14      preservar(Accion,Protegido),
15      plan_mp(Edo,Condicion,Protegido,PrePlan,EdoInter1),
16      aplicar(EdoInter1,Accion,EdoInter2),
17      plan_mp(EdoInter2,Metas,[Meta|Protegido],PostPlan,EdoFinal).
18
19  preservar(Accion,Metas) :-
20      borrar(Accion,ListaBorrar),
21      not( (member(Meta,ListaBorrar),
22      member(Meta,Metas))).
```

Basicamente hemos agregado una restricción al momento de elegir la acción que satisface una *Meta* dada. Además de ser ejecutable, como en el caso del análisis medios-fines original, debe preservar las metas protegidas (línea 14). Una acción preserva una *Meta* data, si no hay intersección entre su lista *borrar*/2 y las metas de la planeación.

De forma que si ejecutamos la consulta:

```
1  ?- estado(Edo0), plan_metas_protegidas(Edo0,[libre(2), libre(3)], P, _).
2  Edo0 = [libre(2), libre(4), libre(b), libre(c), en(a, 1), en(b, 3), en(c, a)],
3  P = [mover(b, 3, 2), mover(b, 2, 4)]
```

obtenemos una solución, aunque ésta sigue sin ser la mejor. Un sólo movimiento `mover(b,3,4)` hubiese sido suficiente para cumplir con nuestras metas. Los planes **innecesariamente largos** son resultado de la estrategia de búsqueda usada por nuestro planeador.

Planes innecesariamente largos

8.4 PROCEDIMIENTOS PRIMERO EN AMPLITUD

Los planeadores implementados hasta ahora, 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* todavía 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  ...
```

Los candidatos cortos para *PrePlan* son los primeros en ser considerados. *PrePlan* establece las condiciones para que *Accion* pueda ejecutarse. Esto permite encontrar una acción cuyas condiciones pueden satisfacerse por un plan tan corto como sea posible, a la manera de una búsqueda primero en amplitud. Por otra parte, la lista de candidatos para el *PostPlan* está totalmente no instanciada, y por tanto su longitud es ilimitada. Por lo tanto, la **estrategia de búsqueda** resultante es globalmente primero en profundidad, y localmente primero en amplitud. Se trata de una búsqueda primero en profundidad con respecto al encadenamiento hacía adelante de las acciones que se agregan al plan emergente, donde cada acción es validada por un *PrePlan* cuya búsqueda se lleva a cabo primero en amplitud.

Estrategias de búsqueda

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:

```
%%%Planificador primero en amplitud

plan_primero_amplitud(Edo,Metas,Plan,EdoFinal) :-
    candidato(Plan),
    plan(Edo,Metas,Plan,EdoFinal).

candidato([]).

candidato([_|Resto]) :-
    candidato(Resto).
```

De forma que la siguiente consulta es posible:

```
?- estado(Edo0), plan_primero_amplitud(Edo0,[libre(2),libre(3)],P,_).
Edo0 = [libre(2), libre(4), libre(b), libre(c), en(a, 1), en(b, 3), en(c, a)],
P = [mover(b, 3, 4)]
```

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

```

%%%Planficador primero en amplitud elegante.

plan_metas_protegidas_amplitud(EdoInicial,Metas,Plan,EdoFinal):-
    plan_mp_amplitud(EdoInicial,Metas,[],Plan,EdoFinal).

plan_mp_amplitud(Edo,Metas,_,[],Edo) :-
    metas_satisfechas(Edo,Metas).

plan_mp_amplitud(Edo,Metas,Protegido,Plan,EdoFinal) :-
    append(Plan,_,_),
    append(PrePlan, [Accion|PostPlan],Plan),
    seleccionar(Edo,Metas,Meta),
    lograr(Accion,Meta),
    precond(Accion,Condicion),
    preservar(Accion,Protegido),
    plan_mp_amplitud(Edo,Condicion,Protegido,PrePlan,EdoInter1),
    aplicar(EdoInter1,Accion,EdoInter2),
```

La consulta es equivalente a la anterior:

```
?- estado(Edo0), plan_metas_protegidas_amplitud(Edo0,[libre(2),libre(3)],P,_).
Edo0 = [libre(2), libre(4), libre(b), libre(c), en(a, 1), en(b, 3), en(c, a)],
P = [mover(b, 3, 4)]
```

Este resultado es óptimo, sin embargo la meta:

```
?- estado(E), plan_metas_protegidas_amplitud(E,[en(a,b), en(b,c)], P, _).
E = [libre(2), libre(4), libre(b), libre(c), en(a, 1), en(b, 3), en(c, a)],
P = [mover(c, a, 2), mover(b, 3, a), mover(b, a, c), 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 por qué se le incluye en el plan y por qué aún en el caso de la búsqueda primero en amplitud, el plan resultante no es el 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 `libre(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.

Incompletez medios-fines

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 tema de la interacción entre metas, 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. Usen *time*/1 para evaluar el impacto en términos de inferencias necesarias para alcanzar la meta de construir la torre (de 6 a 192K inferencias en mi caso).

8.5 REGRESIÓN DE METAS

Supongan que estamos interesados en una lista de *Metas* que se deben cumplir en cierto estado E_f. Sea E_0 el estado anterior a E_f y A la acción considerada para ejecutarse en E_0 ¿Qué $Metas_0$ deben considerarse en E_0 para que *Metas* se cumpla en E_f? La figura 8.4 ilustra esto.

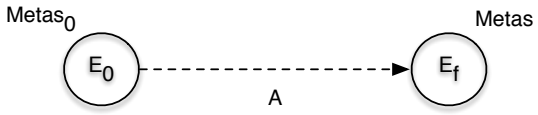

Figura 8.4: Planeación por regresión de metas.

$Metas_0$ debe tener las siguientes propiedades:

1. La acción A debe ser posible en E_0, por lo que $Metas_0$ debe incluir las condiciones de ejecución de A.
2. Para cada meta $M \in Metas$, se cumple que:
   - La acción A agrega M; ó
   - $M \in Metas_0$ y A no borra M.

Regresión de metas

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 los conjuntos que definen A y el de *Metas* se ilustran en la figura 8.5

El mecanismo de regresión de metas puede usarse como planeador de la siguiente manera. Para satisfacer una lista de *Metas* a partir de un estado inicial E_0, se procede como sigue: Si *Metas* se cumple en E_0, 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 E_0.

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

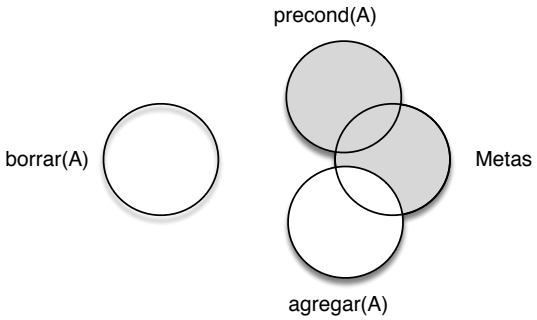

Figura 8.5: Relaciones entre los conjuntos que definen la acción A y las *Metas*. El área sombreada representa las $Metas_0$ resultado de la regresión. Observen que la intersección entre *Metas* y la lista borrar de A debe ser vacía.

```
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:

```
%%%metas incompatibles

imposible(en(X,X),_).

imposible(en(X,Y), Metas) :-
  member(despejado(Y),Metas)
  ;
  member(en(X,Y1),Metas), Y1 \== Y
  ;
  member(en(X1,Y),Metas), X1 \== X.

imposible(despejado(X),Metas) :-
  member(en(_,X),Metas).
```

El resto del planeador es como sigue:

```
%%%Planeador medios fines con regresión de metas

plan(Estado, Metas, []) :-
  satisfecho(Estado, Metas).

plan(Estado, Metas, Plan) :-
  append( PrePlan, [Accion], Plan),
  seleccionar( Estado, Metas, Meta),
  lograr(Accion, Meta),
  precond(Accion, Condicion),
  preservar(Accion, Metas),
  regresion(Metas, Accion, MetasReg),
  plan(Estado, MetasReg, PrePlan).

satisfecho(Estado, Metas) :-
  borrar_todos(Metas,Estado,[]).

seleccionar(_, Metas, Meta) :-
  member( Meta, Metas).

lograr( Accion, Meta) :-
  agregar( Accion, Metas),
  member( Meta, Metas).

borrar_todos( [], _, []).

borrar_todos( [X | L1], L2, Diff) :-
  member( X, L2), !,
  borrar_todos( L1, L2, Diff).

borrar_todos( [X | L1], L2, [X | Diff]) :-
  borrar_todos( L1, L2, Diff).
```

```
33
34 preservar(Accion,Metas) :-
35   borrar(Accion,ListaBorrar),
36   not( (member(Meta,ListaBorrar),
37         member(Meta,Metas))).
38
39 regresion(Metas, Accion, MetasReg) :-
40   agregar(Accion, NuevasRels),
41   borrar_todos(Metas, NuevasRels, RestoMetas),
42   precond(Accion, Condicion),
43   agregarNuevo(Condicion,RestoMetas,MetasReg).
44
45 agregarNuevo([],L,L).
46
47 agregarNuevo([Meta|_],Metas,_) :-
48   imposible(Meta,Metas),
49   !,
50   fail.
51
52 agregarNuevo([X|L1],L2,L3) :-
53   member(X,L2), !,
54   agregarNuevo(L1,L2,L3).
55
56 agregarNuevo([X|L1],L2,[X|L3]) :-
57   agregarNuevo(L1,L2,L3).
```

8.6 MEDIOS FINES CON BÚSQUEDA PRIMERO EL MEJOR

Los planeadores construídos hasta ahora hacen uso de estrategias de búsqueda básicas: primero en profundidad, 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:

Heurísticas

- 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 alternativos, 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: $s(Nodo_1, Nodo_2, Costo)$.
2. Los nodos meta en el espacio: $meta(Nodo)$.
3. Una función heurística de la forma $h(Nodo, Hestimado)$.
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 $Metas_1$ y $Metas_2$ si existe una acción A tal que:

1. A agrega alguna $meta \in Metas_1$.
2. A no destruye ninguna $meta \in Metas_1$
3. $Metas_2$ es el resultado de la regresión de $Metas_1$ a través de A, tal y como definimos en nuestro planeador anterior: $regresion(Metas_1, A, Metas_2)$.

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

```
s(Metas1,Metas2) :-
  member(Meta,Metas1),
  lograr(Accion,Meta),
  precond(Accion,Cond),
  preservar(Accion,Metas1),
  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 logra $en(a, b)$ en el estado inicial que hemos estado usando es:

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

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

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:

```
Metas -> Acción
```

Su implementación detallada es la siguiente:

```
:-  op(300,xfy, ->).

s(Metas -> AccSiguiente,MetasNuevas -> Accion, 1) :-
  member(Meta,Metas),
  lograr(Accion,Meta),
  precond(Accion,Cond),
  preservar(Accion,Metas),
```

```
  regresion(Metas,Accion,MetasNuevas).

meta(Metas -> Accion) :-
  inicio(Estado),
  satisfecho(Estado,Metas).

h(Metas -> Accion,H) :-
  inicio(Estado),
  borrar_todos(Metas,Estado,Insatisfecho),
  length(Instatisfecho,H).

inicio([en(a,1),en(b,3),en(c,a),despejado(b),despejado(c),
        despejado(2),despejado(4)]).
```

Ahora podemos usar nuestro viejo buscador primero el mejor:

```
primeroMejor(Inicio,Solucion) :-
  expandir([],hoja(Inicio,0/0),9999,_,si,Solucion).


%%% expandir(Camino,Arbol,Umbral,Arbol1,Solucionado,Solucion)
%%%   Camino es el recorrido entre Inicio y el nodo en Arbol
%%%   Arbol1 es Arbol expandido bajo el Umbral
%%%   Si la meta se encuentra, Solucion guarda el camino solución
%%%   y  Solucionado = si

% Caso 1: la hoja con Nodo es una meta, construye una solución

expandir(Camino,hoja(Nodo,_),_,_,si,[Nodo|Camino]) :-
  meta(Nodo).

% Caso 2: una hoja con f-valor menor o igual al Umbral
% Generar succesores de Nodo y expandirlos bajo el Umbral

expandir(Camino,hoja(Nodo,F/G),Umbral,Arbol1,Solucionado,Sol) :-
  F  =<  Umbral,
(  bagof( M/C,(s(Nodo,M,C),not(member(M,Camino))),Succ),
   !,     % Nodo tiene sucesores
   listaSuccs(G,Succ,As), % Encontras subárboles As
   mejorF(As,F1),    % f-value of best successor
   expandir(Camino,arbol(Nodo,F1/G,As),Umbral,Arbol1,
      Solucionado,Sol)
  ;
   Solucionado = nunca  % Nodo no tiene sucesores
)  .

% Caso 3: Nodo interno con f-valor menor al Umbral
% Expandir el subárbol más promisorio con cuyo
% resultado, continuar/7 decidirá como proceder

expandir(Camino,arbol(Nodo,F/G,[A|As]),Umbral,Arbol1,
   Solucionado,Sol)  :-
  F  =<  Umbral,
  mejorF(As,MejorF), min(Umbral,MejorF,Umbral1),
  expandir([Nodo|Camino],A,Umbral1,A1,Solucionado1,Sol),
  continuar(Camino,arbol(Nodo,F/G,[A1|As]),Umbral,Arbol1,
      Solucionado1,Solucionado,Sol).

% Caso 4: Nodo interno con subárboles vacío
% Punto muerto, el problema nunca ser
á resuelto

expandir(_,arbol(_,_,[]),_,_,nunca,_) :-  !.

% Caso 5:  f-valor mayor que el Umbral
% Arbol no debe crecer

expandir(_,Arbol,Umbral,Arbol,no,_)  :-
  f(Arbol,F), F > Umbral.

%%% continuar(Camino,Arbol,Umbral,NuevoArbol,SubarbolSolucionado,
```

```
%%%          ArbolSolucionado,Solucion)

% Caso 1: el subartol y el arbol están solucionados
% la solución está en Sol

continuar(_,_,_,_,si,si,Sol).

continuar(Camino,arbol(Nodo,F/G,[A1|As]),Umbral,Arbol1,no,
    Solucionado,Sol)  :-
  insertarArbol(A1,As,NAs),
  mejorF(NAs,F1),
  expandir(Camino,arbol(Nodo,F1/G,NAs),Umbral,Arbol1,
     Solucionado,Sol).

continuar(Camino,arbol(Nodo,F/G,[_|As]),Umbral,Arbol1,nunca,
    Solucionado,Sol)  :-
  mejorF(As,F1),
  expandir(Camino,arbol(Nodo,F1/G,As),Umbral,Arbol1,
     Solucionado,Sol).

%%%listaSuccs(G0,[Nodo1/Costo1, ...], [hoja(MejorNodo,MejorF/G), ...])
%%%  hace una lista de árboles sucesores ordendados por F-valor

listaSuccs(_,[],[]).

listaSuccs(G0,[Nodo/C|NCs],As) :-
  G is G0 + C,
  h(Nodo,H),     % Heuristic term h(N)
  F is G + H,
  listaSuccs(G0,NCs,As1),
  insertarArbol(hoja(Nodo,F/G),As1,As).

%%%Inserta A en una lista de arboles As preservando el orden por f-valor

insertarArbol(A,As,[A|As]) :-
  f(A,F), mejorF(As,F1),
  F =< F1, !.

insertarArbol(A,[A1|As], [A1|As1]) :-
  insertarArbol(A,As,As1).


%%%Extraer f-valores

f(hoja(_,F/_),F).    % f-valor de una hoja
f(arbol(_,F/_,_),F).     % f-valor de un árbol

mejorF([A|_],F) :-  f( A, F).
mejorF([], 9999).

min(X,Y,X) :-  X  =<  Y, !.
min(_,Y,Y).
```

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

```
?- primeroMejor([en(a,b), en(b,c)] -> stop, Plan).
Plan = [[despejado(2), en(c, a), despejado(c), en(b, 3),
         despejado(b), en(a, 1)]->mover(c, a, 2),
        [despejado(c), en(b, 3), despejado(a), despejado(b),
         en(a, 1)]->mover(b, 3, c),
        [despejado(a), despejado(b), en(a, 1), en(b, c)]
         ->mover(a, 1, b),
        [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.

8.7 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.

8.7.1 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 *bloque*(*Bloque*) entre otras. Este tipo de meta hace que *Bloque* siempre está 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 *libre*(*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 instanciaciones 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         [libre(Bloque),libre(A),en(Bloque,De)]).
```

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

```
1 [libre(Bloque),libre(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* está en el bloque *c*! Esto podría arreglarse si especificáramos que *De* y *A* deben ser diferentes:

```
1 precond(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 *metas_logradas*/2 la siguiente cláusula:

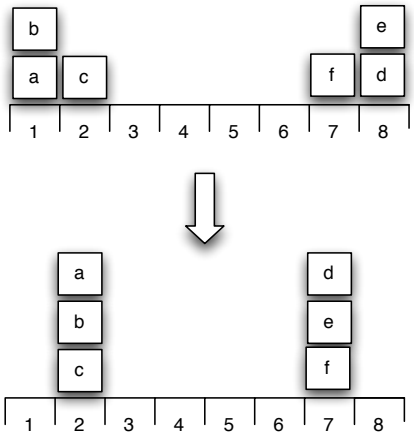

Figura 8.6: Una tarea de planeación con dos planes independientes

```
metas_logradas(Estado,[Meta|Metas]) :-
   satisface(Meta),
   metas_logradas(Estado,Metas).
```

De forma que debemos definir también:

```
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.

8.7.2 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 8.6, 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:

```
Plan1 = [mover(b,a,c), mover(a,1,b)]
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:

```
[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 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 8.6. 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:

```
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 $libre(a)$, 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.

8.8 LECTURAS Y EJERCICIOS SUGERIDOS

La planeación es un problema muy estudiado en Inteligencia Artificial. Los planificadores aquí construídos fueron introducidos por Bratko [18] en el capítulo dedicado al tema. Una revisión exhaustiva y reciente de este problema puede encontrarse en el libro de Ghallab, Nau y Traverso [55]. Una lectura más tradicional se puede encontrar en el compendio de artículos revisado por Allen, Hendler y Tate [1]. La descripción de las acciones en estos planificadores es muy cercana a la propuesta de STRIPS, originalmente presentada por Fikes y Nilsson [46]. El mundo de los bloques ha sido ampliamente discutido en este contexto, un interesante estudio sobre su complejidad y la fuente de esta fue realizado por Ghallab, Nau y Traverso [55].

Ejercicios

Ejercicio 8.1. *Pruebe estos planificadores en otros dominios de aplicación, por ejemplo, el proceso de inscripción a la universidad. Describe las acciones en los términos adecuados y las metas del proceso.*

Ejercicio 8.2. *¿Cual de los planificadores resulta de interés para su tema de investigación? Justifique la respuesta.*

Ejercicio 8.3. *Lea el artículo de Ghallab, Nau y Traverso [55]. Explique el concepto de* deadlock *y analice si alguno de los planificadores presentados pueden contender con él.*