ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DEL SOFTWARE

PLAGUE: UN EDITOR VISUAL PARA PDDL
PLAGUE: A VISUAL EDITOR FOR PDDL

Realizado por
José Luis Usero Vílchez
Tutorizado por
José Antonio Montenegro Montes
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, NOVIEMBRE 2015

Fecha defensa:
El Secretario del Tribunal

RESUMEN: Plague es un editor de archivos escritos en lenguajes de planificación como STRIPS y PDDL, que permite lanzar el algoritmo GrahPlan a partir de los archivos de dominio y problema editados y encontrar una solución al problema planteado. El objetivo del editor es eminentemente pedagógico: su uso es muy simple y viene con variados ejemplos de ambos lenguajes de planificación, de modo que el usuario pueda aprenderlos de forma paulatina. Además, la salida de la ejecución permite ir viendo paso a paso el desarrollo del algoritmo GraphPlan: los operadores que se van ejecutando, los no-ops que se han seguido, los mutex que se han aplicado en cada nivel y el tiempo empleado, además de la solución final al problema si se alcanza. El programa hace uso de dos utilidades que permiten compilar el código STRIPS o PDDL que son JavaGP y PDDL4J. Una vez ejecutado el problema de planificación, se obtiene la salida en pantalla y también se puede imprimir el problema completo incluida la solución. El objetivo ha sido crear un programa que permita al usuario editar rápidamente archivos STRIPS y PDDL, los pueda compilar velozmente y obtener el resultado en un solo sitio, con una salida mucho más clara, organizada y entendible y se evite el problema de tener que usar editores externos y una ventana de línea de comando para ejecutar GraphPlan.
PALABRAS CLAVES:
PDDL, STRIPS, GraphPlan, Planificación, Inteligencia Artificial.

ABSTRACT: Plague is a text editor for files written in action languages, such as STRIPS and PDDL, which allows running the GraphPlan algorithm from the domain archives and edited problems, and finding a solution to the proposed problem. The goal of the editor is primarily for pedagogical purposes: it is simple to use and comes equipped with a variety of examples in both action languages, so that the user can gradually learn. In addition, as the editor runs it allows the user to observe the step by step development of the GraphPlan algorithm: the operators being executed, the no-ops that have been followed, the mutex applied at each level and the time spent, as well as the final answer to the problem, if reached. The program uses two utilities allowing the STRIPS or PDDL code to be compiled: JavaGP and PDDL4J. Once the planning problem has been executed, the result is shown on screen and the complete problem can also be printed, including the solution. The objective has been to create a program that allows the user to quickly edit STRIPS and PDDL archives, to compile them swiftly and obtain the solution in a single place, with a result that is clear, organised and understandable, thus avoiding the problem of having to use external editors and command prompts to execute GraphPlan.
KEYWORDS: PDDL, STRIPS. GraphPlan, Planning, Artificial Intelligence

ÍNDICE

1. INTRODUCCIÓN

1.1. MOTIVACIÓN

La planificación es, según el diccionario de la Real Academia:

1. f. Acción y efecto de planificar.

2. f. Plan general, metódicamente organizado y frecuentemente de gran amplitud, para obtener un objetivo determinado, tal como el desarrollo armónico de una ciudad, el desarrollo económico, la investigación científica, el funcionamiento de una industria, etc.

En el campo de la Inteligencia Artificial, la planificación se puede definir como (1) un proceso de deliberación abstracto y explícito que elige y organiza las acciones anticipando sus resultados esperados.

Entre los algoritmos de planificación, el GraphPlan es uno de los más usados y podemos encontrar muchas implementaciones del mismo. Por ejemplo, javagp. Pero la salida es la siguiente:

```
INFO: Expanding graph
INFO: Goals not possible with 1 steps
INFO: Expanding graph
INFO: Goals not possible with 2 steps
…
INFO: Plan found
ir(m,a,c)
empujar(m,c,b,ca)
subirse(m,ca,b)
agarrar(m,p,b)
INFO: Planning took 0s
```

Las implementaciones no dan ninguna información sobre los pasos que va dando el algoritmo, los nodos que abre y porqué encuentra o no encuentra una solución de planificación válida al problema. Si hay un pequeño error en cualquier regla, resulta muy difícil depurarlo y, para “mundos” con muchos tipos de objetos y acciones, la tarea se complica.

Hemos pensado que un editor de “mundos” que acceda a un parser de PDDL y vaya dando retroalimentación de los pasos que realiza el algoritmo GraphPlan sería muy útil para cualquier persona que esté empezando a aprender PDDL.

La motivación de este proyecto es eminentemente pedagógica: crear una herramienta que permita al alumno o alumna aprender STRIPS, PDDL y el funcionamiento de GraphPlan. Y, por otro lado, mostrar a los alumnos y alumnas a los que doy clase que cuando nos planificamos podemos ser mucho más efectivos al estudiar, sacar buenas notas y tener tiempo para salir con los amigos, hacer deporte y todas esas cosas que hace la gente que no estudia una ingeniería.

1.2. OBJETIVOS

El objetivo de este proyecto es crear un editor de PDDL que ayude al desarrollador indicando la salida del parser de PDDL para cada paso de la aplicación del algoritmo GraphPlan al problema de planificación planteado.

Los requisitos que nos planteamos que cumpla el editor PLAGUE (Planning LAnGUage Editor) son los siguientes:

- El editor ha de tener todas las características básicas que un desarrollador desea de modo que se facilite su trabajo: copiar y pegar, pestañas, búsqueda, teclas de acceso rápido, etc.
- El editor ha de ser multidocumento, de modo que el desarrollador o desarrolladora pueda tener varios “mundos” abiertos a la vez. En la parte izquierda tendrá el archivo donde se define el dominio y a la derecha la definición del problema, de modo que tenga acceso visual a ambas a la vez.
- En la parte inferior se mostrará una ventana de salida, donde se irán mostrando las salidas del parser conforme se vayan produciendo.
- Se ha de permitir la ejecución paso a paso, indicando en la ventana de salida las acciones que se llevan a cabo y los estados a los que llevan estas acciones para cada paso del algoritmo GraphPlan.
- La salida también ha de poder soportar un modo mínimo. En este modo, la salida será similar a la mostrada en la página anterior, con solo el número de niveles y la solución.
- La salida ha de mostrar el tiempo que se tarda en encontrarla y el tiempo que se tarda en completar cada nivel hasta llegar a ella, de modo que permita al alumno o alumna comparar el comportamiento del algoritmo para problemas de planificación de distinta complejidad.
- Si se encuentra una solución al problema de planificación, el programa ha de permitir obtener una traza de los nodos que se han ido abriendo para llegar de la definición del problema a esta solución directamente.
- El editor ha de permitir la impresión de la salida en un formato agradable al usuario donde se puedan observar las acciones y los estados que el algoritmo va generando.
- Dado el enfoque eminentemente pedagógico del proyecto, el editor vendrá con algunos mundos precargados para mostrar el funcionamiento del algoritmo GraphPlan para que los alumnos y alumnas puedan comprobar su funcionamiento. Algunos mundos clásicos de planificación que se agregarán al sistema son el mundo de los bloques, el puzzle-8, el laberinto y el mono-plátano-silla.
- Teniendo en cuenta que el objetivo principal es pedagógico, se incluirá un tutorial interactivo para, usando problemas simples de planificación como el del pastel, ir llevando paso a paso al alumno o alumna por la escritura de sus primeros problemas de planificación en PDDL, aprovechando la salida para glosar el funcionamiento del algoritmo GraphPlan.

1.3. TECNOLOGÍAS

1.3.1. PDDL

PDDL es el lenguaje estándar de facto para resolver problemas de planificación en Inteligencia Artificial. Actualmente se encuentra en la versión 3.1.

- Los componentes básicos de una tarea de planificación en PDDL son:
- Objetos: seres o cosas que forman parte del "mundo".
- Predicados: propiedades de los objetos, pueden ser verdaderas o falsas.
- Estado inicial: propiedades iniciales de los objetos de nuestro "mundo".
- Estado objetivo: propiedades finales de nuestros objetos.
- Acciones: que se pueden llevar sobre los objetos y que cambian su estado.

Un problema de planificación en PDDL consta de dos archivos:

- Un archivo de dominio, donde se definen las acciones que pueden tener lugar, con los predicados que se tienen que cumplir para que se pueda realizar esa acción o precondiciones y los predicados que son verdad después de llevar a cabo esa acción o postcondiciones.
- Un archivo de problema, donde se define el estado inicial, esto es, los predicados que son ciertos al principio, y un estado final, los predicados que deben ser ciertos al final tras aplicar las acciones definidas en el dominio.

1.3.2. PDDL4J o JAVAGP

PLAGUE se va a construir sobre un parser de PDDL de código abierto que implementa GraphPlan como PDDL4J. PDDL4J está escrito en Java y contiene un parser de PDDL 3.0 y las clases Domain.java y Problem.java, así como otras clases auxiliares para gestión de errores.

La forma en la que PLAGUE opera es:

- El desarrollador escribe sus archivos de dominio y problema.
- Al *ejecutar* el problema de planificación, PLAGUE envía esos archivos al parser.
- Conforme el parser va ejecutando el algoritmo GraphPlan sobre los archivos, el código de PLAGUE inspecciona las acciones que se van llevando a cabo y los predicados que se van generando, los formatea y los va mostrando por la ventana de salida.
- Una vez terminado el algoritmo de planificación, PLAGUE guarda toda la información en un archivo para que pueda ser analizada.

1.3.3. Entorno de desarrollo

Como entorno de desarrollo se va a usar NetBeans y como lenguaje de programación Java por su facilidad de manejo, cantidad de material didáctico y amplio uso en la comunidad.

1.3.4. Modelado

Para las tareas de modelado del sistema se usará UML (Unified Modeling Language):

- Para el análisis de requisitos se realizarán casos de uso.
- Diseño: realizaremos diagramas de clase para estudiar las nuevas clases que servirán de interfaz entre la salida del parser y PLAGUE.

Dado el carácter eminentemente pedagógico de este proyecto, se va a hacer especial hincapié en diseñar una interfaz usable y una salida fácilmente entendible, usando los principios de diseño de la interfaz de Shneiderman.

2. MARCO TEÓRICO

2.1. UNA VUELTA POR STRIPS: EL MUNDO DE LOS AVIONES

Como hemos dicho, la planificación es (1) un proceso de deliberación abstracto y explícito que elige y organiza las acciones anticipando sus resultados esperados.

Vamos a explicar qué es planificar a partir de un ejemplo simple, el mundo *Aviones*. En principio, podemos resolver un problema de planificación con una implementación del algoritmo *GraphPlan* (2) (que estudiaremos en detalle más adelante) como *emplan* (3) y un par de archivos *dominioavion.txt* y *problemaavion.txt.* Vamos a ver que contienen estos archivos:

dominioavion.txt

```
operator cargar(P,A,AERO)
pre: paquete(P), avion(A), aeropuerto(AERO), en(P,AERO), en(A,AERO)
post: ~en(P,AERO), dentro(P,A)

operator descargar(P,A,AERO)
pre: paquete(P), avion(A), aeropuerto(AERO),dentro(P,A), en(A,AERO)
post: en(P,AERO), ~dentro(P,A)

operator volar(A,DESDE,HASTA)
pre: avion(A), aeropuerto(DESDE), aeropuerto(HASTA), en(A,DESDE)
post: ~en(A,DESDE), en(A,HASTA)
```

problemaavion.txt

```
start (
avion(a1),
avion(a2),
paquete(p1),
paquete(p2),
aeropuerto(agp),
aeropuerto(mad),
en(a1, agp),
en(p1, agp),
en(a2, mad),
en(p2, mad))

goal(
en(p1, mad),
en(p2, agp)
)
```

En el archivo *dominioavion.txt*, la palabra *operator* es bastante significativa. Seguramente esos van a ser operadores. Pero, ¿sobre qué operan? Y *pre* y *post* suenan bastante a precondiciones y postcondiciones… ¿de qué?

Y en el archivo *problemaavion.txt* nos encontramos con las palabras *start*, comienzo y *goal, objetivo.* Tenemos un comienzo y queremos llegar a un objetivo.

En un problema de búsqueda existen cuatro elementos fundamentales: un estado inicial, una serie de acciones, los resultados que tiene la aplicación de cada acción y un objetivo.

Pues bien, STRIPS (Stanford Research Institute Problem Solver) se refiere a un lenguaje que se creó para introducir planes en el generador de planes del mismo nombre. Este código está escrito en STRIPS. Vamos a desmenuzar el código anterior y, pensando que se trata de un mundo de aviones, vamos a interpretar el significado de cada línea:

```
start (
```

esto tiene que ser el estado inicial

```
avion(a1),
```

tenemos un objeto avión que se llama a1

```
avion(a2),
```

otro avión que se llama a2

```
paquete(p1),
paquete(p2),
```

los paquetes, que supongo que quiero transportar, p1 y p2

```
aeropuerto(agp),
aeropuerto(mad),
```

dos aeropuertos, el de Málaga y el de Madrid

Parece claro que en STRIPS en el estado inicial definimos los **objetos** que forman nuestro mundo. En este caso tenemos aviones, paquetes y aeropuertos.

```
en(a1, agp),
```

¿Qué puede significar esto? Podemos realizar la asunción de que tenemos el a1, el avión número 1, en el Aeropuerto de Málaga.

```
en(p1, agp),
```

Y aquí tenemos el paquete 1 en el Aeropuerto de Málaga también.

```
en(a2, mad),
en(p2, mad))
```

Y en el Aeropuerto de Madrid tenemos un avión y un paquete.

También parece claro que los objetos que hemos definido tienen **estados**. En este caso, encontrarse en un lugar.

```
goal(
```

Y goal no puede ser más que el objetivo.

```
en(p1, mad),
```

```
en(p2, agp)
```
Y mi objetivo no es otro que llevar el paquete 1, que está en Málaga, a Madrid y el paquete 2, que está en Madrid, a Málaga.
```
)
```

Pues ya nos hemos hecho una idea del contenido del archivo *problemaavion.txt*. Parece ser que se requiere un fichero donde se definan unas condiciones iniciales, con los objetos que tiene nuestro *mundo*, el estado de cada uno de ellos y los objetivos que queremos conseguir. Pero, ¿cómo llegamos del estado inicial al estado objetivo? Ha de haber alguna manera de cambiar los estados iniciales para llegar al objetivo. Vamos a ver el archivo *dominioavion.txt:*

```
operator cargar(P,A,AERO)
```
Un operator es un predicado o una acción que cambia el estado de uno o varios objetos. En este caso el operador cargar indica que necesita un paquete P, un avión A y un aeropuerto AERO.
```
pre: paquete(P), avion(A), aeropuerto(AERO), en(P,AERO),
en(A,AERO)
```
Tenemos unas precondiciones que definen los objetos que tienen que existir y los estados en los que se tienen que encontrar los objetos para que se pueda aplicar el predicado en cuestión.

Es de notar como hay que especificar que los objetos tienen que existir en la precondición, es decir, debe haber un objeto paquete P, un objeto avión A y un objeto aeropuerto AERO. Luego tenemos los estados en los que tienen que estar los objetos de nuestro *mundo*: el paquete P debe estar en el aeropuerto AERO y el avión A también debe estar en ese mismo aeropuerto. Si el paquete y el avión no están en el mismo aeropuerto, no podemos cargar el paquete en el avión.
```
post: ~en(P,AERO), dentro(P,A)
```
Y unas postcondiciones que especifican los estados que cambian. Hay estados que desaparecen, como en este caso el negado y estados a los que llegan los objetos como consecuencia de las acciones que se llevan a cabo.

Una vez que metemos el paquete en el avión, el paquete ya no está en el aeropuerto (que sí, que sigue estando en el aeropuerto pero vamos a pensar que tenemos una nave para los paquetes y que, una vez que los paquetes salen de esa nave y son introducidos en el avión, ya no rezan como que están en el aeropuerto, sino dentro del avión). Además, el paquete P está dentro del avión A.
```
operator descargar(P,A,AERO)
```
La operación descargar es simétrica a cargar.
```
pre: paquete(P), avion(A), aeropuerto(AERO),dentro(P,A),
en(A,AERO)
```
El paquete está dentro del avión y el avión está en el aeropuerto.
```
post: en(P,AERO), ~dentro(P,A)
```
Una vez sacado del avión, el paquete ya no está dentro del avión pero sí que está en el aeropuerto.
```
operator volar(A,DESDE,HASTA)
```
Los aviones pueden volar desde un origen DESDE hasta un destino HASTA.

```
pre: avion(A), aeropuerto(DESDE), aeropuerto(HASTA),
en(A,DESDE)
```

Ha de existir un avión que esté en un aeropuerto DESDE y ha de existir otro aeropuerto HASTA a donde quiera llegar.

```
post: ~en(A,DESDE), en(A,HASTA)
```

El avión ya no está en el aeropuerto de origen DESDE y sí que está en el aeropuerto de destino HASTA.

El archivo de dominio *dominioavion.txt* contiene los predicados, las acciones que los objetos pueden llevar a cabo y que cambian sus estados. Una vez escritos los dos ficheros y enviados al planificador, éste nos responde con una serie de acciones que consiguen, mediante una serie de predicados, ir desde el estado inicial al estado objetivo. Veamos el plan que nos ha devuelto javagp, una implementación de GRAPHPLAN:

```
cargar(p2,a2,mad)
cargar(p1,a1,agp)
volar(a1,agp,mad)
volar(a2,mad,agp)
descargar(p1,a1,mad)
descargar(p2,a2,agp)
```

Para ser un trozo de metal y plástico, el ordenador ha dado un plan bastante bueno. ¿Qué nos propone que hagamos?

1. Carga el paquete 2, que está en el aeropuerto de Madrid, en el avión 2, que está también en el aeropuerto de Madrid.
2. Carga el paquete 1, que está en el aeropuerto de Málaga, en el avión 1, que también está en el aeropuerto de Málaga.
3. Que el avión 1, que lleva el paquete 1, vaya de Málaga a Madrid.
4. Que el avión 2, que lleva el paquete 2, vaya de Madrid a Málaga.
5. Descarga el paquete 1 que está en el avión 1 en Madrid. OBJETIVO CUMPLIDO. Ya tenemos el paquete 1 en Madrid.
6. Descarga el paquete 2 que está en el avión 2 en Málaga. OBJETIVO CUMPLIDO. Ya tenemos el paquete 2 en Málaga.

2.2. Los grafos de planificación y GraphPlan

¿Quién hace la magia de, partiendo de unos objetos con sus propiedades y unas acciones, ir de un estado inicial a uno final? ¿Cómo lo hace? Vamos a ver como un algoritmo relativamente sencillo es capaz de resolver problemas de planificación de forma automática.

GraphPlan trabaja sobre **grafos de planificación**, que guardan información sobre los posibles planes y las restricciones sobre los objetos existentes. Un grafo de planificación es dirigido y está dividido en niveles. Tiene dos tipos de nodos: literales y acciones y cada nivel es de un tipo que se va alternando. Así, en el nivel 0 tenemos literales, en el nivel 1 acciones y así sucesivamente. Estos literales y acciones están unidos por líneas que indican, para cada acción, los literales que son precondiciones de la misma en el nivel anterior y los literales que son postcondiciones de la misma en el nivel posterior. Como una imagen vale más que mil palabras, vamos a mostrar un grafo de planificación:

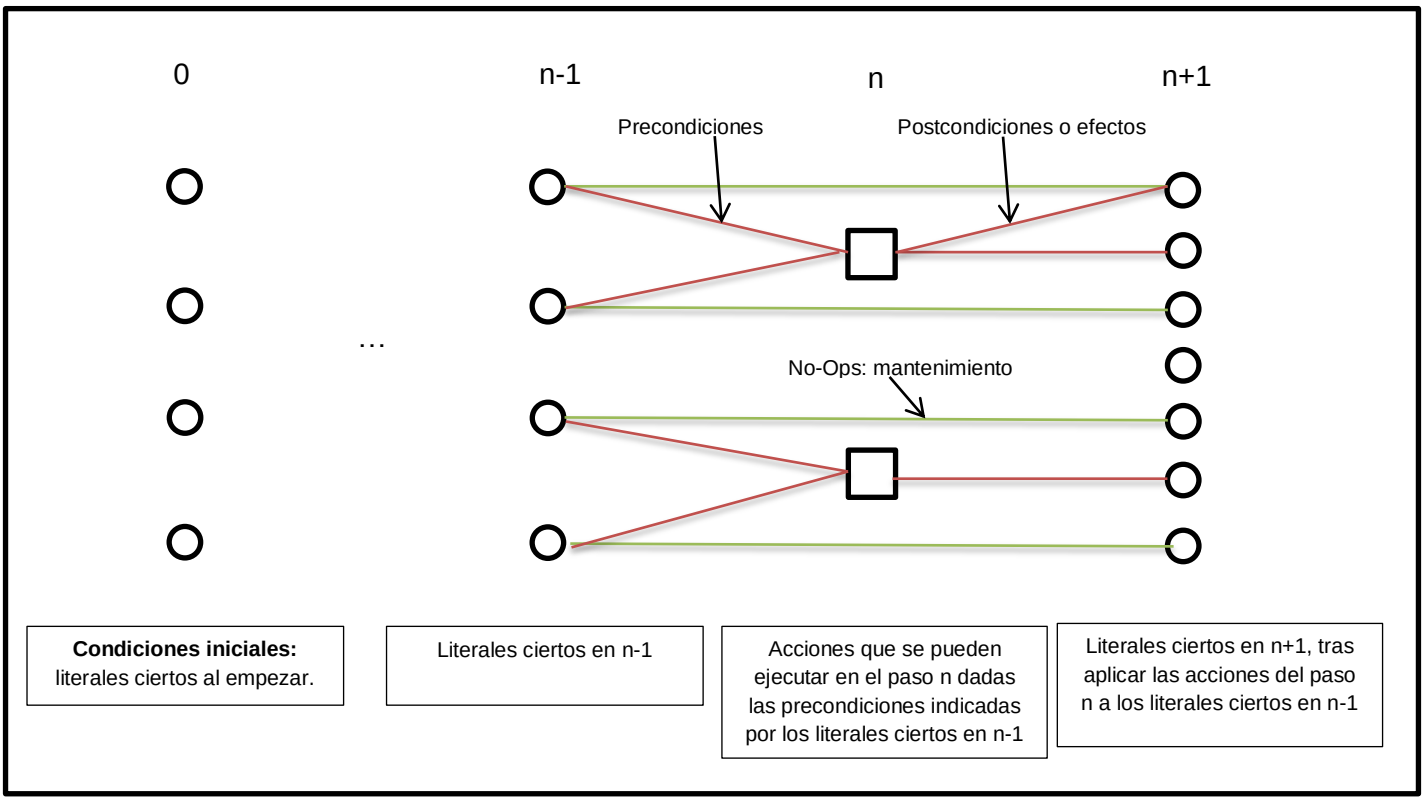


Figura 2.1. Grafo de planificación. Fuente (4)

En este grafo de planificación, tenemos en el nivel n-1 una serie de estados o literales ciertos. Por ejemplo, en el caso del mundo de los aviones visto anteriormente, en el nivel 0 tenemos los literales

```
avion(a1) paquete(p1) aeropuerto(agp) en(a1,agp) en(p1,agp)
```

En el siguiente nivel n tenemos las acciones que pueden ver sus condiciones satisfechas, esto es, solo se ponen las acciones cuyas precondiciones aparecen en el paso n-1. Así, en este ejemplo, en el nivel 1, nivel de acciones, podemos aplicar el operador

```
cargar(p1,a1,agp)
```

```
post: ~en(P,AERO), dentro(P,A)
```

El nivel n+1 es el conjunto de estados o literales consecuencia de aplicar las acciones del nivel n a los literales del nivel n-1. El resultado de esto se va a mostrar en el nivel 2, de estados, como la aparición del estado ~en(p1,agp), que niega la precondición en(p1,agp) del paso 0, así como la aparición de dentro(p1,a1).

.

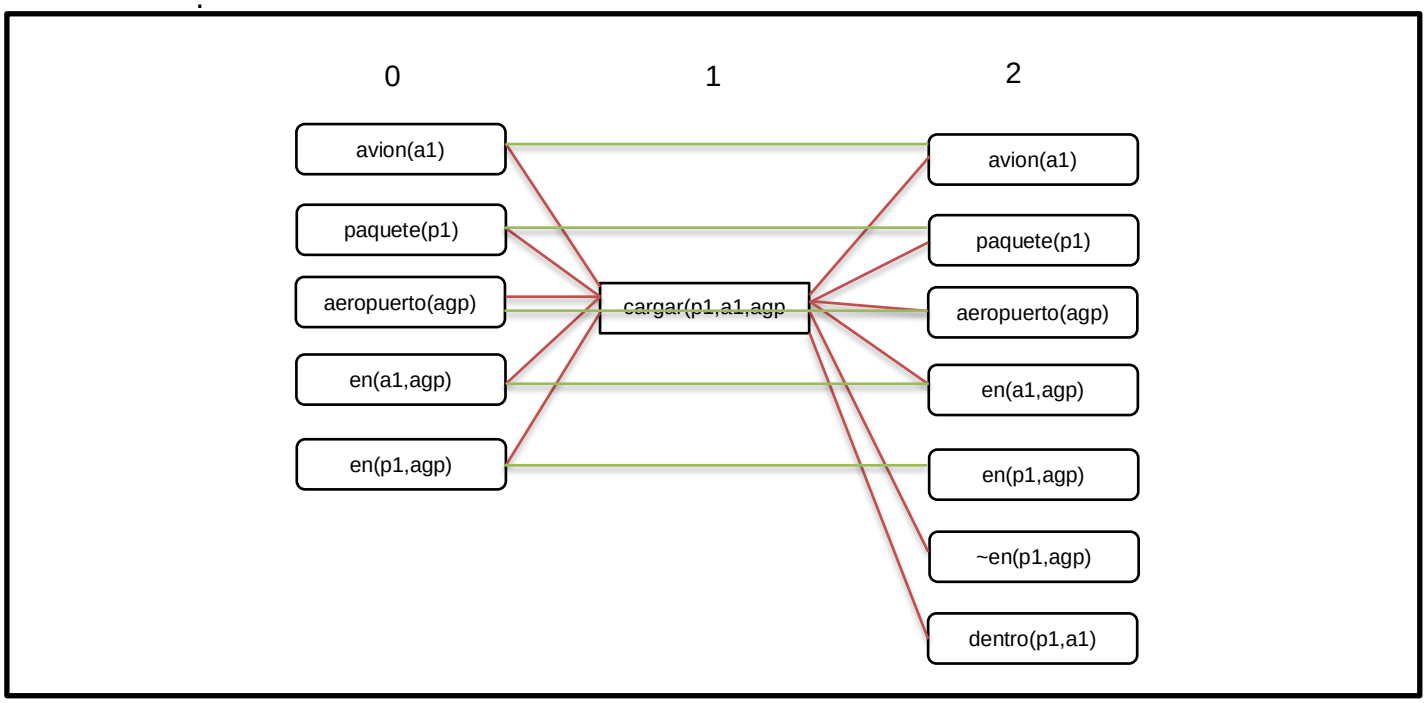


Figura 2.2. Acción cargar: subconjunto del grafo de planificación

En el primer nivel o nivel 0 del grafo de planificación vamos a tener un nodo por cada literal del estado inicial del problema. Luego en el nivel 1, el primer nivel de acciones, tendremos todas las acciones cuyas precondiciones sean cubiertas por los literales del nivel 0. Una vez llevadas a cabo estas acciones, el nivel 2 contendrá nodos para las postcondiciones de la realización de estas acciones, así como nodos para cada literal del nivel 0, que se van a repetir en este nivel. Es decir, los literales del nivel 0 van a seguir siendo verdaderos si ninguna acción lo niega. Pero, en nuestro ejemplo, sí que tenemos una acción que niega un literal del nivel 0. ¿Qué hacemos con eso? Lo veremos en el próximo capítulo.

¿Y hay que hacer esto para cada acción posible de cada nivel? Pues sí. ¿Y arrastrar todos los estados de un nivel a otro? Pues sí. ¿Y de qué sirve todo esto? Pues observa la figura 2.2. ¿No parece extraño que aparezca un estado en(p1,agp) y su negado ~en(p1,agp)? Pues esto nos va a servir para luego ir recortando el grafo y viendo qué acciones nos sirven y cuáles no.

Vamos a ver un ejemplo completo de grafo de planificación, vamos con el ejemplo clásico del pastel. En el mundo pastel tenemos, eso, un pastel, y un par de acciones que son:

```
operator comer(Pastel)
pre:tener(Pastel)
post:~tener(Pastel),comido(Pastel)

operator cocinar(Pastel)
pre:~tener(Pastel)
post:tener(Pastel)
```

Y los estados inicial y final son:

```
start (
tener(pastel),
~comido(Pastel)
)

goal(
comido(pastel),
tener(pastel)
)
```

Tenemos un pastel y queremos comernos el pastel y tener otro. ¿Qué hacemos? Pues nos comemos el que tenemos y cocinamos otro. Este ejemplo tan simple nos va a ayudar a ver el grafo de planificación completo.

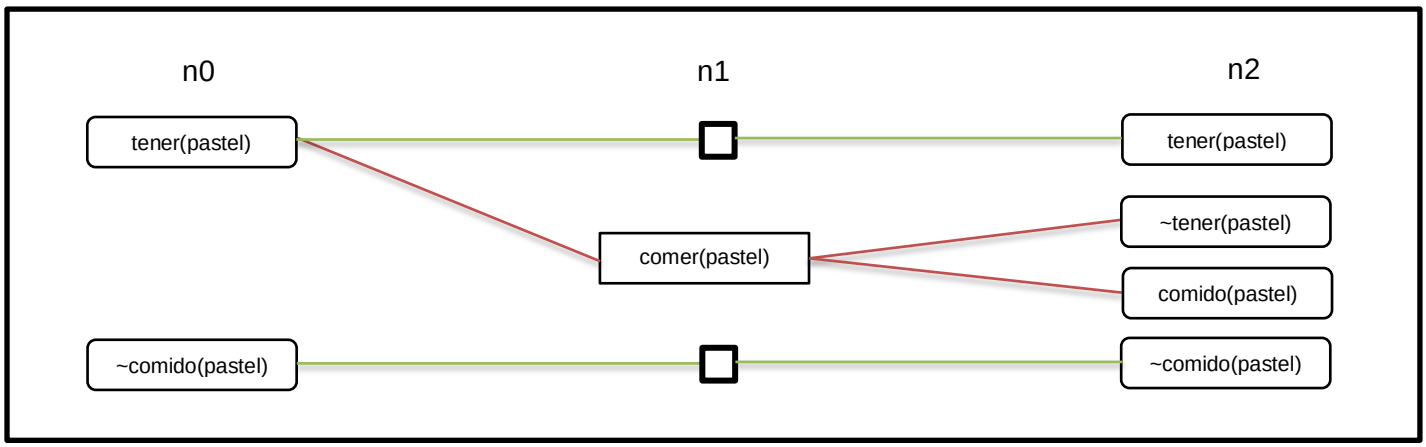


Figura 2.3. Primeros niveles del Mundo Pastel: noops y acción comer

La primera acción que se puede realizar es comer(pastel) ya que tenemos su precondición en el estado inicial. Al realizar esta acción, tenemos como postcondiciones ~tener(pastel) y comido(pastel), justo los estados contrarios a los estados iniciales, que se propagan al siguiente nivel de estados. ¿Qué podemos hacer a partir de aquí? Pues seguir propagando acciones a partir de las precondiciones que tenemos en el nivel n2. Vamos a ver cómo nos queda.

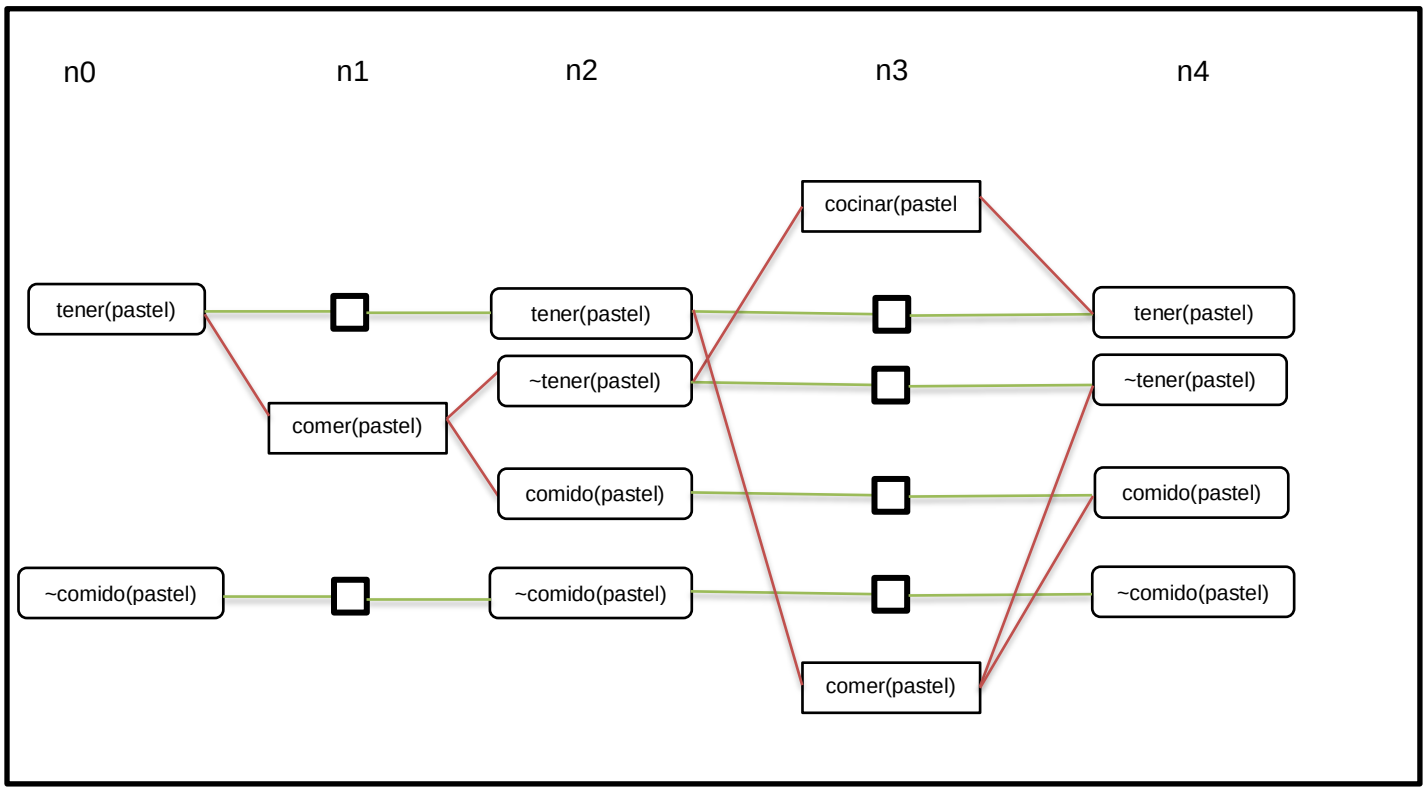


Figura 2.4. Grafo del Mundo Pastel completo

Con los cuatro estados del nivel 2, podemos ejecutar las dos acciones. Por un lado, como no tenemos un pastel, podemos ejecutar la acción cocinar(pastel). Por otro lado, como tenemos un pastel, podemos ejecutar la acción comer(pastel). Esto nos lleva a los mismos estados en el nivel 4 que tenía en el nivel 2, así que hemos alcanzado la condición de parada, que es que **dos niveles consecutivos de estados sean idénticos**. Esto significa que las acciones que hemos llevado a cabo no pueden cambiar ya el resultado. Con esto, ya hemos desarrollado por completo el grafo. Ahora sí, vamos a ver qué hacer con el grafo para llegar a una solución.

2.3. No se puede estar en el caldo y en las *tajás*: los mutex

En el grafo de planificación anterior, tenemos los estados y sus opuestos y parece obvio que no se pueden cumplir los dos a la vez. Ha de haber alguna manera de saber qué acciones nos llevan a esos estados inconsistentes para no ejecutarlas. De este modo, llegaremos a un conjunto de acciones que nos lleven por conjuntos de estados consistentes, y ésa será la solución del problema.

Una vez completado el grafo, hemos de crear enlaces de exclusión mutua. Vamos a ver en qué casos se da exclusión mutua. Hay tres tipos de mutex entre acciones del mismo nivel:

Mutex Tipo 1: Efectos inconsistentes: una acción niega los efectos de otra. En nuestro ejemplo del pastel, la acción comer(pastel) lleva a ~tener(pastel). Pero el noOp de tener(pastel) nos lleva a tener en el mismo nivel tener(pastel), justo el estado contrario. Y no es posible tener y no tener un pastel a la vez.

En la figura, así como en las siguientes los rectángulos con esquinas redondeadas son estados, los rectángulos con esquinas son acciones, las líneas rojas son precondiciones y postcondiciones, los cuadrados negros son acciones noops o persistencias que van unidas por líneas verdes y las curvas azules son enlaces mutex.

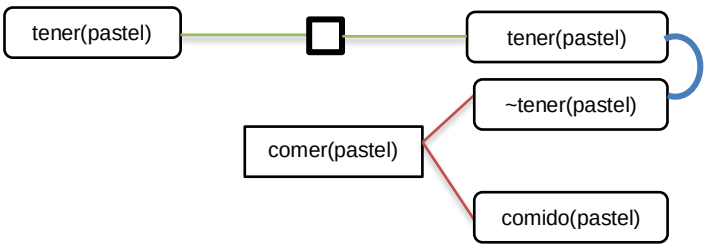


Figura 2.5. Mutex 1: efectos inconsistentes

Mutex Tipo 2: Interferencia: uno de los efectos de una acción niega la precondición de otra. En el mundo pastel no ocurre este mutex.

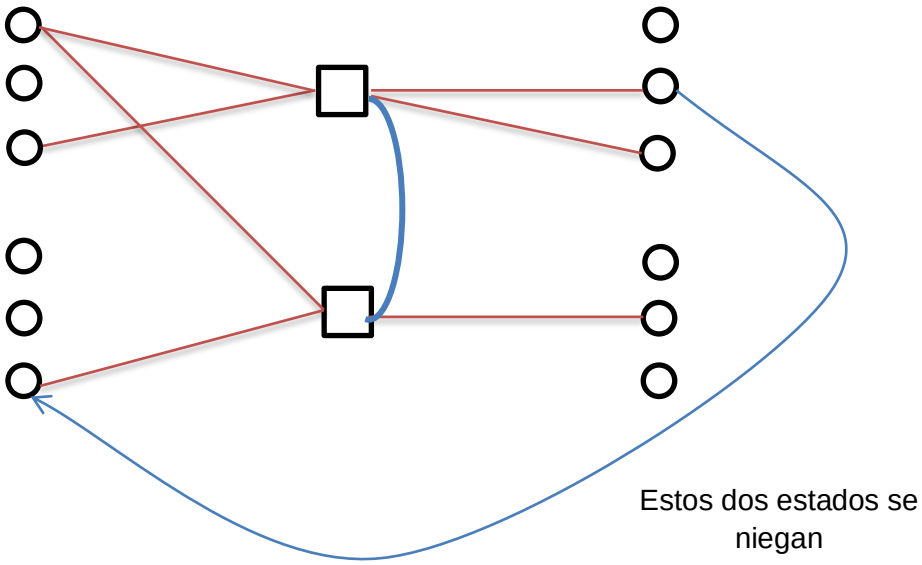


Figura 2.6. Mutex 2: interferencia

Mutex Tipo 3: Necesidades competitivas: la precondición de una acción se excluye mutuamente con la precondición de otra, es decir, son opuestas. En nuestro caso, comer(pastel) y cocinar(pastel) requieren de estados como ~tener(pastel) y tener(pastel), lo que evita que estas dos acciones se puedan dar en el mismo paso.

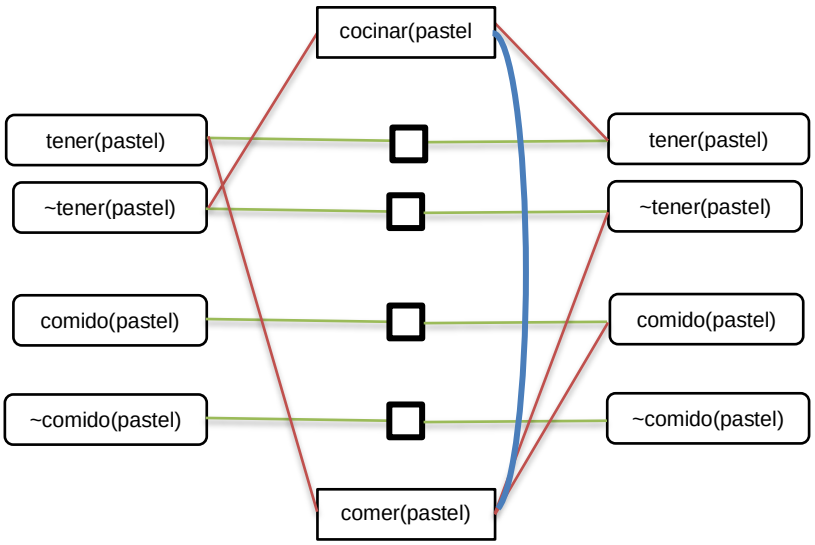


Figura 2.7. Mutex 3: necesidades competitivas

Y un mutex entre literales de un mismo nivel:

Mutex tipo 4: Soporte inconsistente: se da cuando aparecen juntos dos literales contrarios en el mismo nivel o cuando dos acciones se excluyen por necesitar una un literal y otra su negado.

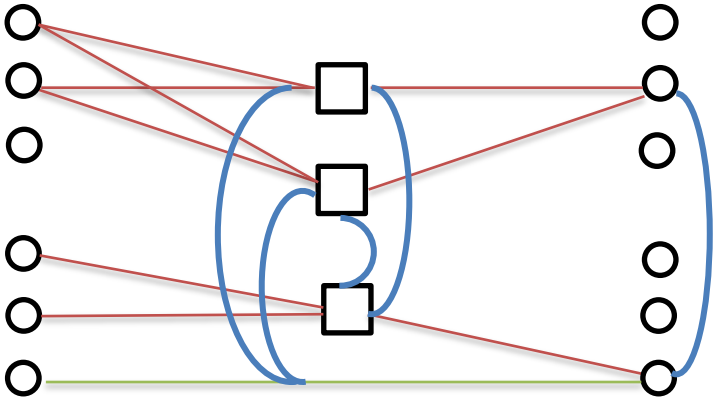


Figura 2.8. Mutex 4: soporte inconsistente

En nuestro ejemplo tener(pastel) y comido(pastel) se excluyen noOp con la acción comer(pastel) porque si me como el pastel ya está comido, lo que lleva a una inconsistencia ya que tengo el estado ~comido(pastel).

Si aplicamos estas cuatro reglas a nuestro grafo, nos queda una figura como la siguiente:

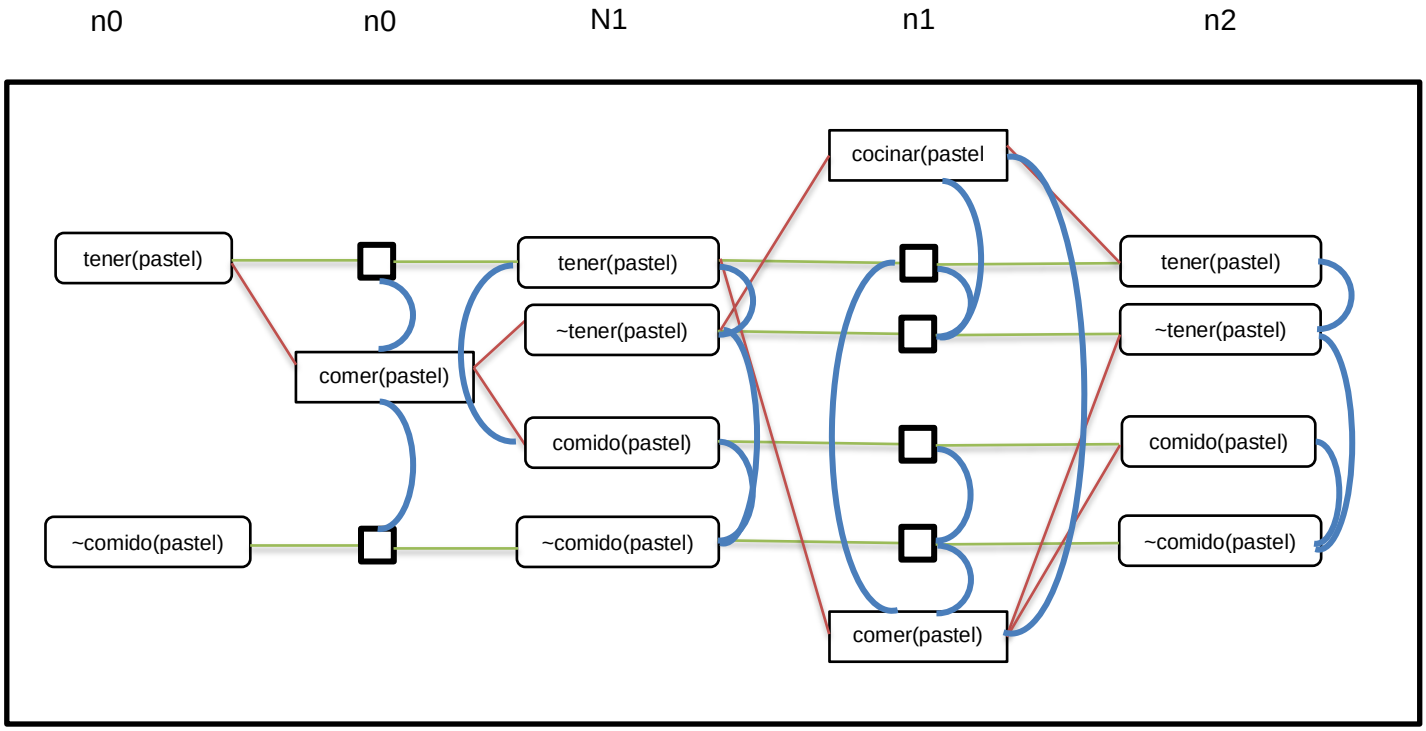


Figura 2.9. Grafo de planificación de Mundo Pastel con mutexs aplicados (5)

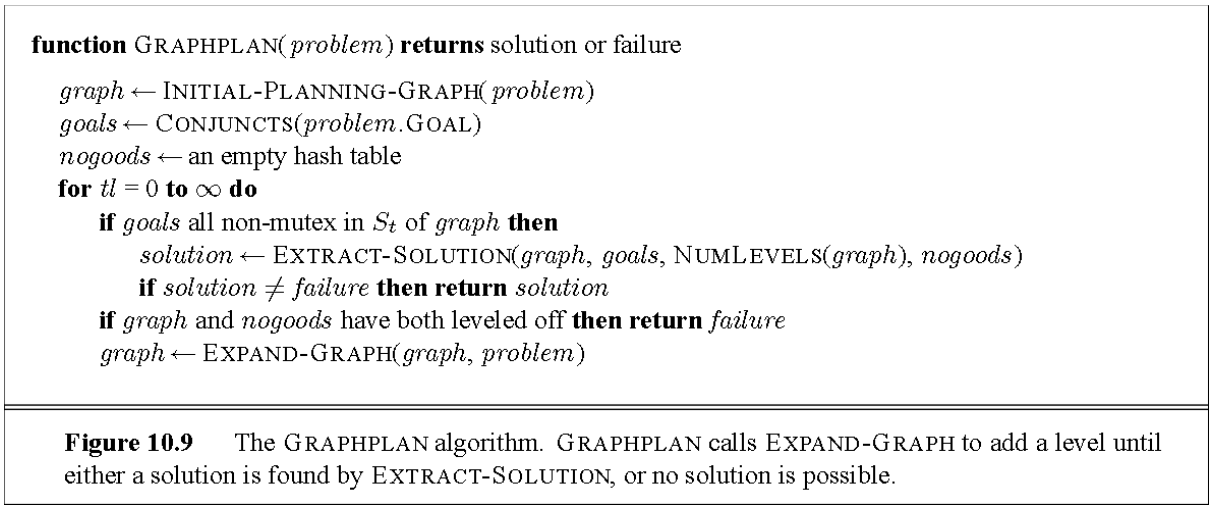
Aquí podemos ver el grafo de planificación con algunos mutex aplicados:

- comer(pastel) del primer nivel de acciones se bloquea con los noOp de tener(pastel) y ~comido(pastel) ya que la propia acción de comer(pastel) lleva a no tener el pastel y que ya haya sido comido.
- En el segundo nivel de estados las acciones tener(pastel) se excluye con ~tener(pastel) y comido(pastel) se excluye con ~comido(pastel).
- Además, no se puede tener(pastel) si se ha comido(pastel) y no se puede ~tener(pastel) si no se ha comido pastel ~comido(pastel).
- En el quinto nivel se repiten las exclusiones de estados del tercer nivel, descritas en los dos últimos puntos.
- Además, se excluyen las acciones cocinar(pastel) y comer(pastel) en el cuarto nivel porque llevan a tener(pastel) y ~tener(pastel), que son estados mutuamente excluyentes.

Pues ya lo tenemos el grafo preparado para el último paso: ¿habrá un conjunto de acciones que solucione el problema? Veamos.

2.4. "Tenemos un ejército" "Nosotros tenemos un… algoritmo"

Ahora vamos al meollo. Aquí está el GraphPlan tal y como aparece en el libro Artificial Intelligence: A Modern Approach, de S. Russell y P. Norvig.



function GRAPHPLAN(*problem*) **returns** solution or failure

graph ← INITIAL-PLANNING-GRAPH(*problem*)
goals ← CONJUNCTS(*problem*.GOAL)
nogoods ← an empty hash table
for *tl* = 0 **to** ∞ **do**
  if *goals* all non-mutex in S_t of *graph* **then**
    solution ← EXTRACT-SOLUTION(*graph*, *goals*, NUMLEVELS(*graph*), *nogoods*)
    if *solution* ≠ *failure* **then return** *solution*
  if *graph* and *nogoods* have both leveled off **then return** *failure*
  graph ← EXPAND-GRAPH(*graph*, *problem*)

Figure 10.9 The GRAPHPLAN algorithm. GRAPHPLAN calls EXPAND-GRAPH to add a level until either a solution is found by EXTRACT-SOLUTION, or no solution is possible.

Figura 2.10. El algoritmo GRAPHPLAN (5)

`function GRAPHPLAN(problem) returns solution or failure`

La función GRAPHPLAN recibe un problema y devuelve una solución o error.

`Graph <- INITIAL-PLANNING-GRAPH(problem)`

Se crea un grafo de planificación para el problema.

`goals<-CONJUNCTS(problem.GOAL)`

Se crea un objetivo, que es la conjunción de todos los estados del objetivo en el problema que se le pasa a GRAPHPLAN. Por ejemplo, en el problema anterior.

```
goal(
comido(pastel),
tener(pastel)
)
```

`nogoods <- an empty hash table`

Se crea una tabla hash vacía que va a contener conjuntos de objetivos que no se pueden conseguir.

`for tl=0 to ∞ do`

Aunque este bucle se puede repetir infinitas veces, ahora veremos las condiciones que hacen que, de una u otra forma, encontremos una solución o encontremos que no existe tal.

`if goals all non-mutex in St of graph then`

Si todos los objetivos no están en exclusión mutua en el paso St del grafo de planificación entonces: si he llegado a un paso de estados en el cual todos los estados del objetivo aparecen y no hay exclusión mutua entre ellos, eso quiere decir que hay algún subconjunto de acciones que hace que se llegue al objetivo. ¿Cuál es ese conjunto de acciones?

`solution<-EXTRACT-SOLUTION(graph,goals,NUMLEVELS(graph),nogoods)`

Se llama a la función EXTRAER-SOLUCIÓN, que a partir del grafo de planificación, los objetivos, el número de niveles y el conjunto de objetivos que no se pueden conseguir, devuelve el conjunto de acciones a realizar.

`if solution≠failure then return solution`

Si el conjunto de acciones es consistente, es decir, no da error, se devuelve este conjunto como solución.

`if graph and nogoods are both levelled off then return failure`

Si el grafo de planificación ha llegado a tener varios niveles de estados iguales y la tabla de conjuntos de estados erróneos también, entonces no se va a encontrar una solución así que se devuelve error.

`graph<-EXPAND-GRAPH(graph,problem)`

Si llegamos aquí y no se ha obtenido una solución, expandimos el grafo con un nivel más y seguimos probando.

¿Cómo extraemos una solución del grafo de planificación anterior? Los estados tener(pastel) y comido(pastel), no están en mutex, así que debe existir un conjunto de acciones que nos permitan alcanzarlos. Sin embargo, las acciones comer(pastel) y cocinar(pastel) no podemos realizarlas a la vez, ya que están en mutex. Así que vamos a elegir la acción comer(pastel). Pero no podemos realizarla, porque nos lleva a ~tener(pastel), que entra en exclusión con nuestro objetivo tener(pastel). Por tanto en ese nivel solo podemos elegir la acción cocinar(pastel). Ahora seguimos hacía atrás. ¿Podemos realizar la acción comer(pastel) en el primer nivel de acciones? Pues sí, porque nos lleva al estado ~tener(pastel), que es necesario para disparar la acción cocinar(pastel) y al estado comido(pastel), uno de nuestros objetivos. Así que la búsqueda hacia atrás de una solución nos devuelve un conjunto de acciones que no están en mutex y llegan a los objetivos. ¡Perfecto!

En este caso hemos tenido suerte y hemos encontrado una solución. Pero, ¿y si no la hay? ¿Se sigue ejecutando hasta el infinito? el algoritmo es muy listo y sabe cuándo no tiene que seguir buscando. ¿Cómo lo hace? Bueno, cuando ve que hay dos niveles iguales, con los mismos estados y los mismos mutex, se da cuenta de que ninguna acción va a cambiar la situación, así que, ¿para qué seguir? (En esto muestra una gran inteligencia porque algunos seres humanos nos empecinamos en algo y no veas…) Una vez llegados a este *punto fijo*, si alguno de los estados objetivo no se cumple, o dos estados objetivo están en mutex, es que ningún conjunto de acciones va a conseguir encontrar una solución, así que se devuelve *error*.

2.5. Una cita para cenar

2.5.1. La cita resuelta

Ahora que tenemos una idea bastante clara de lo que hace GRAPHPLAN, vamos a aplicarlo a un ejemplo un poco más complejo, la cita para la cena (6), un modelo clásico de planificación.

El problema consiste en organizar una cena romántica para nuestra novia (o novio), que está durmiendo plácidamente. El objetivo es múltiple, ya que tenemos que sacar la basura, hacer la cena y envolver un regalo que hemos preparado para que la cita sea perfecta. Para darle un poco más de emoción, le voy a pedir matrimonio así que el regalo es un anillo de compromiso. Podemos realizar cuatro acciones: cocinar, envolver el regalo, sacar la basura y sacar la basura con un carrito, que es más cómodo.

Pero cada acción tiene sus precondiciones y postcondiciones. Para empezar a cocinar tenemos que tenerlas manos limpias y al final tenemos una apetitosa cena preparada. Para envolver el regalo tenemos que estar en silencio para no despertar a nuestra soñadora pareja, y, tras envolverlo, tenemos el regalo perfectamente envuelto y preparado. Sacarbasura elimina la basura, pero llevarla en las manos durante todo el camino hace que al final no tengamos las manos limpias. Sacarbasuraconcarrito no nos llena las manos, pero el carrito hace mucho ruido así que no estamos en silencio.

En el estado inicial, tenemos las manos limpias, la casa está en silencio y tenemos una bolsa de basura. Al final queremos tener la cena preparada, el regalo envuelto y la basura fuera en el contenedor. ¿Cómo podemos conseguirlo?

Aquí tenemos el archivo domaincena.txt escrito en pddl para aclarar las acciones que podemos realizar:

```
operator cocinar(X)
pre: manoslimpias(X)
post: cena(X)

operator envolver(X)
pre: silencio(X)
post: regalo(X)

operator sacarbasura(X)
pre: true
post:~basura(X),~manoslimpias(X)

operator sacarbasuraconcarrito(X)
pre: true
post:~basura(X),~silencio(X)
```

Y el archivo *problemcena.txt* con los estados inicial y final:

```
start(
basura(x), //aunque no se coloque como condición inicial, la
solución es la misma, ya que exigimos en el objetivo que no
halla basura
manoslimpias(x),
silencio(x)
)

goal(
cena(x),
regalo(x),
~basura(x)
)
```

Y la solución que nos devuelve GRAPHPLAN:

```
cocinar(x)
envolver(x)
sacarbasura(x)
```

2.5.2. Un cambio de notación

En este ejemplo, para ser consistentes con la bibliografía, vamos a modificar ligeramente los gráficos que hemos ido mostrando. Hasta ahora, hemos seguido la notación del libro *Artificial Intelligence: A Modern Approach de S. Russell y P. Norvig*, en el que, de un nivel de estados a otro, se mantienen los literales negativos. A partir de ahora, vamos a usar la notación de la publicación original de *A. Blum y M. Furst, Fast Planning Through Planning Graph Analysis* (7), donde solo se pasan de un estado a otro los estados positivos. Por ejemplo, la acción *sacarbasura* se va a representar del siguiente modo:

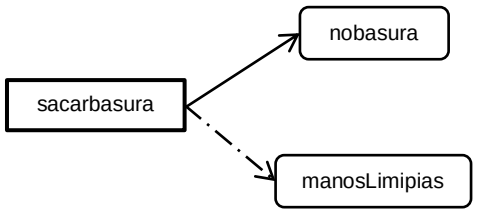


Figura 2.11. Notación de Blum y Furst para acciones y estados

La acción *sacarbasura* activa el estado *noBasura* y niega el estado *manosLimpias.* Los estados positivos van a seguir a una flecha continua y los estados negativos van a seguir a una flecha discontinua.

Mantenemos la notación de los noOp, ya que el algoritmo puede decidir no ejecutar cierta acción de un paso a otro y este *operador* nos permite mantener un estado de un paso a otro del algoritmo.

2.5.3. Un plan consistente

Un plan es una solución si es:

- Completo, es decir, todos los estados objetivo se cumplen en el último paso y las precondiciones de todas las acciones en el paso i son satisfechas por estados del paso i.
- Consistente: las acciones en el paso i pueden ser ejecutadas en cualquier orden, sin que una de esas acciones deshaga las precondiciones de otra acción en el paso i ni deshaga los efectos de otra acción en el paso i.

Para el problema que nos ocupa, vamos a ver una solución completa y consistente:

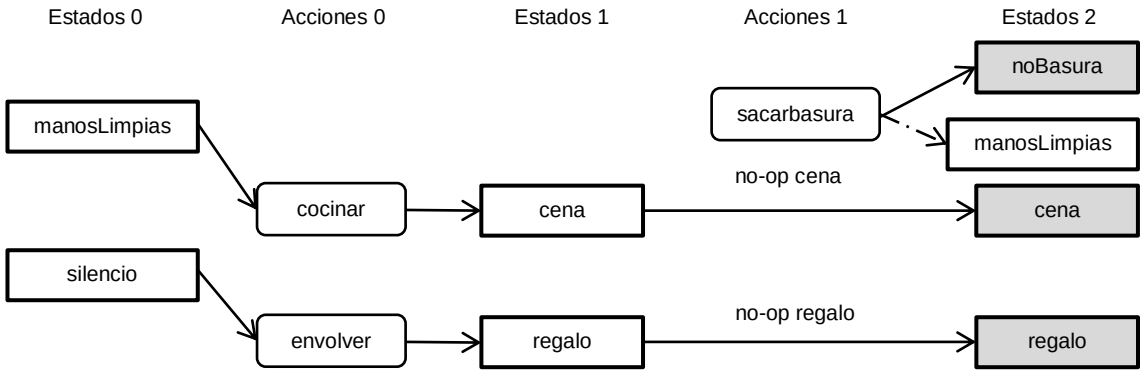


Figura 2.12. Plan completo y consistente de la cita para cenar (8)

Las acciones serán rectángulos redondeados y los estados rectángulos con pico. Los estados sombreados en gris son los estados objetivos.

Este plan es completo porque en el estado final se alcanzan todos los estados objetivo y es consistente porque las acciones no entran en exclusión mutua: podemos cocinar porque tenemos las manos limpias y podemos envolver el regalo porque estamos en silencio, después, podemos sacar la basura. Aunque terminemos con las manos sucias, eso no supone un problema ya que alcanzamos los estados finales y éste no aparece de ninguna manera en ellos.

2.5.4. Un grafo de planificación para la cita para cenar

Un grafo de planificación contiene un conjunto de planes que son completos y consistentes y además nos permite *podar* los estados y las acciones que no se pueden alcanzar desde el estado inicial, los pares de estados y acciones que no son consistentes en un paso determinado y los planes que no alcanzan los estados objetivo. Si del grafo de planificación quitamos todos los planes señalados, los que nos quedan son planes que contienen estados y acciones consistentes que se

pueden alcanzar desde el estado inicial y nos llevan al estado final, es decir, tenemos soluciones al problema.

Aunque tienen muchas ventajas, por ejemplo que se pueden construir en un tiempo polinomial y su tamaño también lo es, lo que ahorra mucho tiempo y espacio en memoria comparado con otros enfoques donde el conjunto de estados a estudiar crece exponencialmente, todavía quedan planes no factibles que hay que eliminar haciendo una búsqueda enfocada en el objetivo.

Vamos a ir siguiente paso a paso como construimos el grafo de planificación para este problema:

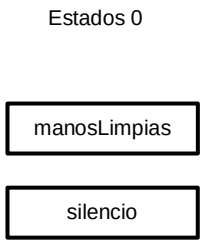


Figura 2.13. Estado inicial

Una vez establecidos los estados iniciales, se genera la **capa de acciones 0**, que contendrá todas las acciones posibles que se pueden realizar tomando como precondiciones los estados iniciales.

Luego se genera la **capa de estados 1** que contiene el resultado de aplicar las acciones a los estados iniciales, es decir, las postcondiciones de cada acción ejecutado en la capa de acciones 0. En nuestro caso:

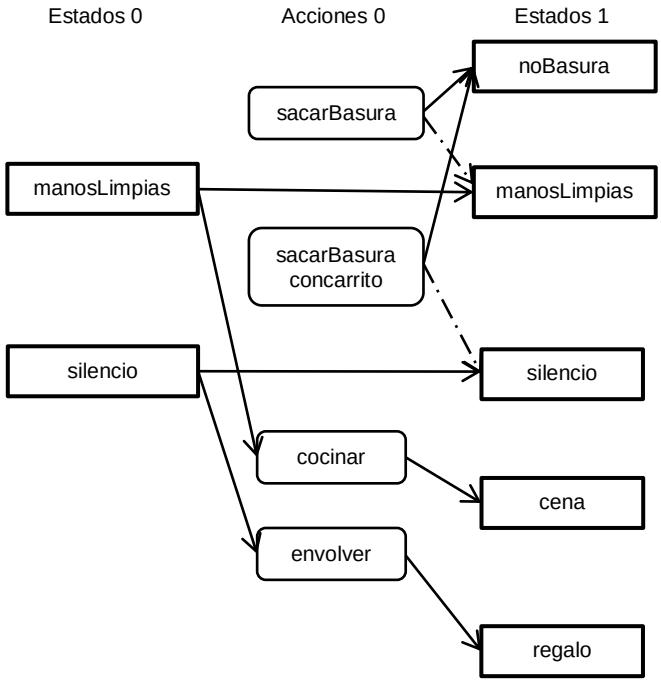


Figura 2.14. Acciones en el nivel 0 y estados en el nivel 1

Las acciones *sacarBasura* y *sacarBasuraconcarrito* me llevan al estado *noBasura,* pero tienen como contrapartida que con una me ensucio las manos y con otra ya no estoy en silencio. Además, puedo ejecutar la acción *cocinar* porque tengo las manos limpias inicialmente y puedo envolver el regalo porque estoy en silencio.

Ahora me pregunto: **¿aparecen todos los estados objetivo?** Pues sí, tenemos *noBasura, cena* y *regalo*. Pero he de cumplir una condición más y es que tienen que aparecer sin excluirse mutuamente. Así que ahora tengo que buscar las exclusiones mutuas, primero entre acciones y luego entre estados.

Recordemos que dos acciones son mutuamente exclusivas en un paso i si ningún plan puede contener ambas acciones a la vez. Hay **tres tipos de exclusiones mutuas entre acciones**:

- Efectos inconsistentes: la acción A borra los efectos de la acción B.
- Los efectos interfieren con las precondiciones: A elimina las precondiciones de B o viceversa.
- Compiten por necesidades: A y B tienen precondiciones inconsistentes, es decir, opuestas.

Vamos a ver las exclusiones mutuas entre acciones una por una:

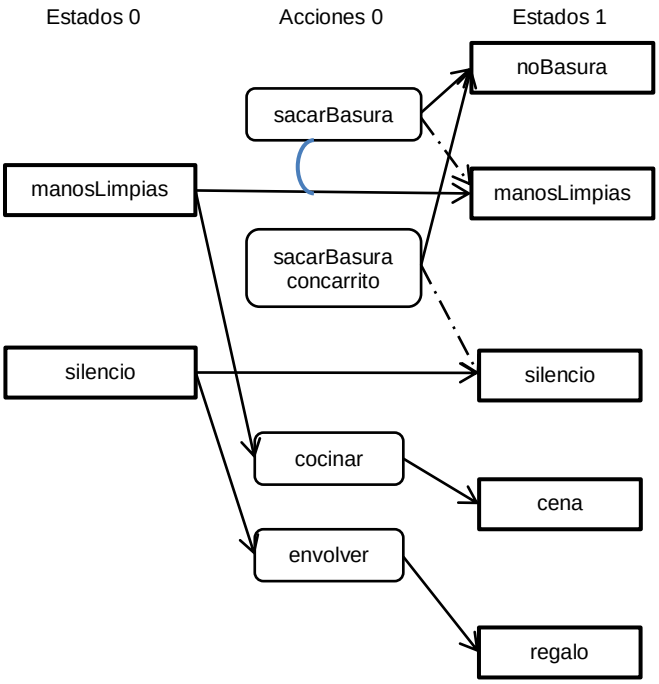


Figura 2.15. Mutex entre sacarBasura y manosLimpias

La acción *sacarbasuraconcarrito* lleva negar el estado *manosLimpias* que aparece entre los estados iniciales, es decir, llegamos al paso 1 con un estado y su negado y eso es inconsistente, no puedo tener las manos limpias y sucias a la vez.

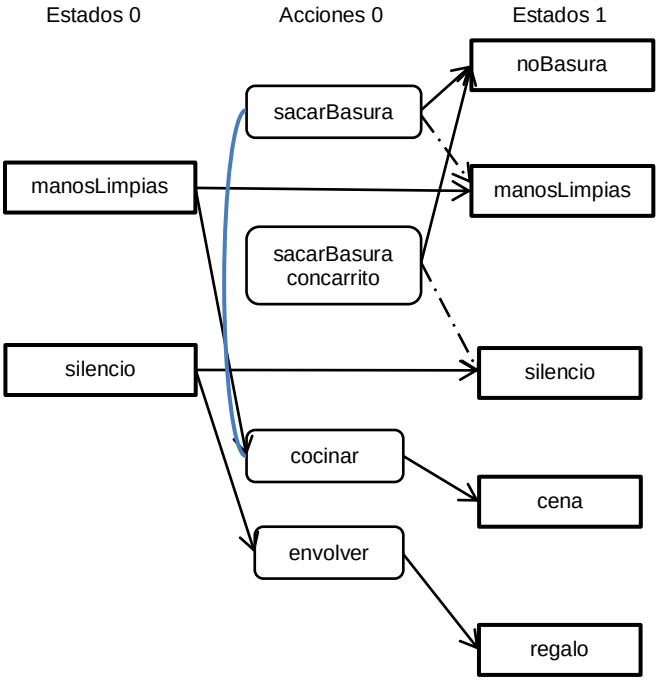


Figura 2.16. Mutex entre sacarBasura y cocinar

sacarBasura te ensucia las manos, que es precondición de *cocinar*, con lo que los efectos de una acción interfieren con las precondiciones de otra. No voy a poder cocinar y sacar la basura en el mismo paso, porque uno me va a ensuciar las manos, que necesito limpias para cocinar. Es de notar que, si en un paso se pueden realizar dos acciones a la vez, éstas deberán poder ejecutarse en cualquier orden. En este caso, si saco la basura luego no puedo cocinar, así que estas dos acciones son mutuamente excluyentes.

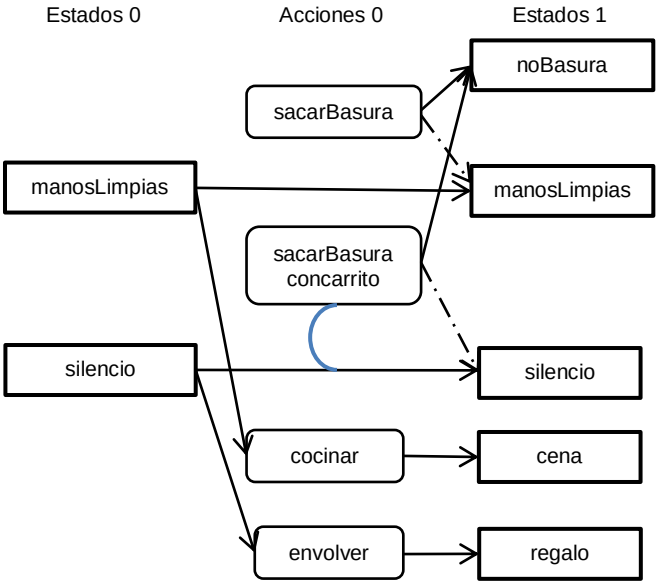


Figura 2.17. Mutex entre sacarBasuraconcarrito y Silencio

Si saco la basura con el carrito, hago ruido, con lo que niego el estado *silencio*, que era un estado inicial. Otra exclusión mutua.

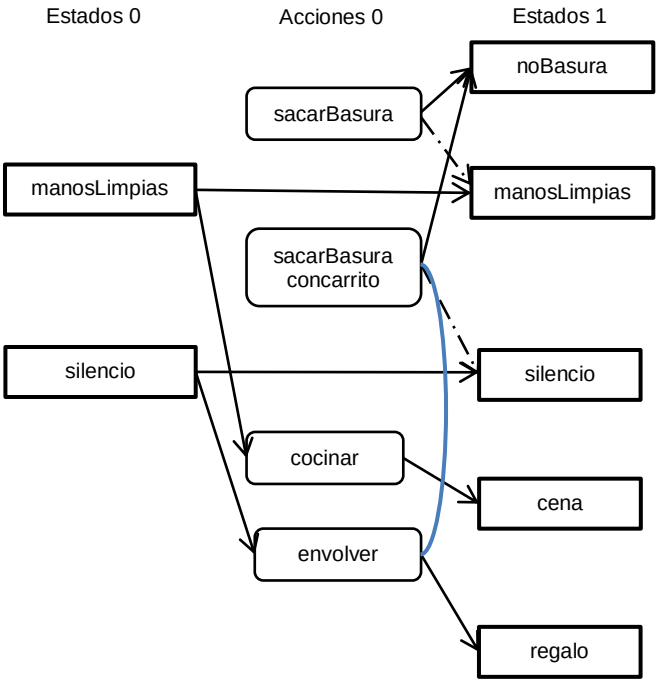


Figura 2.18. Mutex entre sacarBasuraconcarrito y envolver

Si saco la basura con el carrito, ya no estoy en silencio, así que no podría envolver el regalo porque para ello necesito silencio.

Resumiendo, tengo todas estos mutexs de acciones:

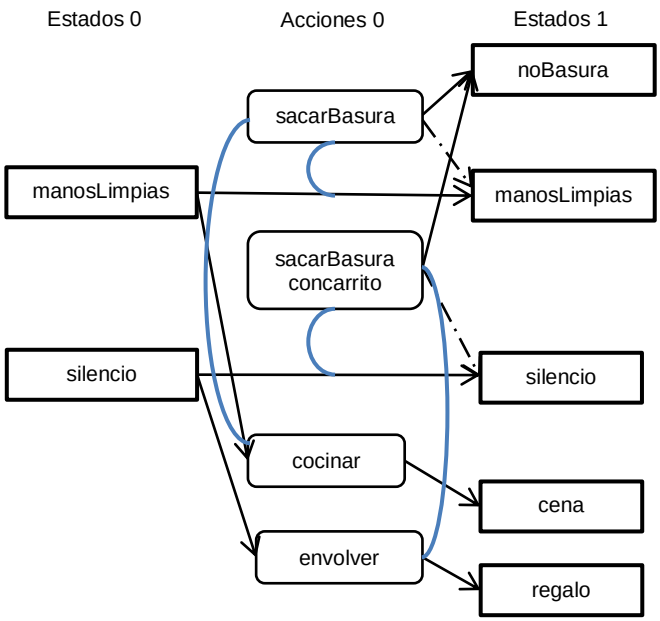


Figura 2.19. Todos los mutex detectados en el nivel 0

También hay **exclusiones mutuas entre estados**. Veamos cómo se reconocen. Dos estados A y B son inconsistentes en un paso i si ningún plan válido puede contener ambos estados en ese paso, o, también, si todas las maneras de alcanzar el estado A excluye todas las maneras de alcanzar B.

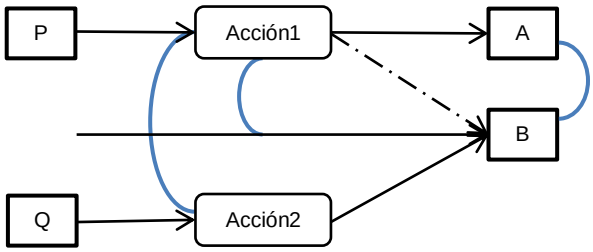


Figura 2.20. Exclusiones mutuas entre estados

En nuestro ejemplo no se produce este tipo de mutex. Bueno, ya tenemos todos los mutex aplicados. ¿Qué hacemos ahora?

Ahora hay que ver si todos los estados objetivo aparecen sin exclusiones mutuas entre ellos. Veamos:

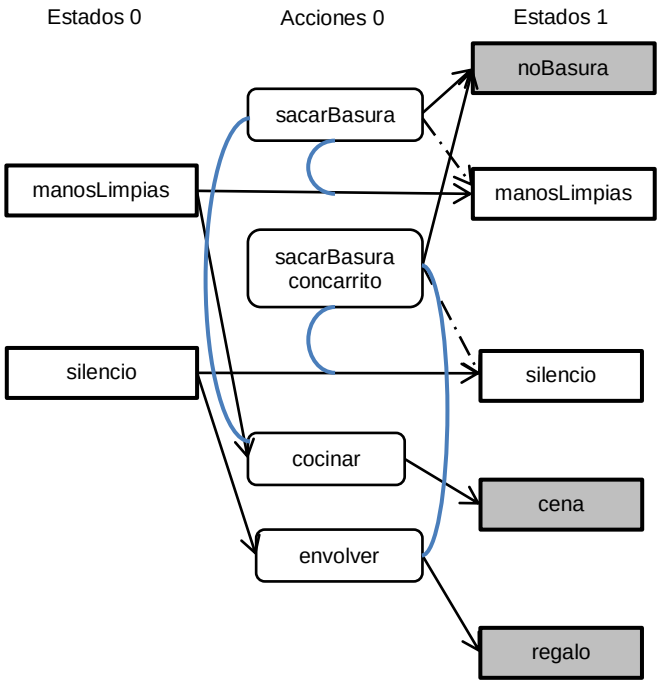


Figura 2.21. ¿Aparecen los estados objetivo sin exclusiones mutuas?

Si saco la basura no puedo cocinar porque me lleno las manos y si saco la basura con el carrito hago ruido y entonces no puedo envolver el regalo, así que no hay un conjunto de acciones que me lleve a todos los estados objetivo. Así que voy al siguiente paso, que es **generar el siguiente paso.** Aplicando lo que ya sabemos resulta el siguiente grafo de planificación.

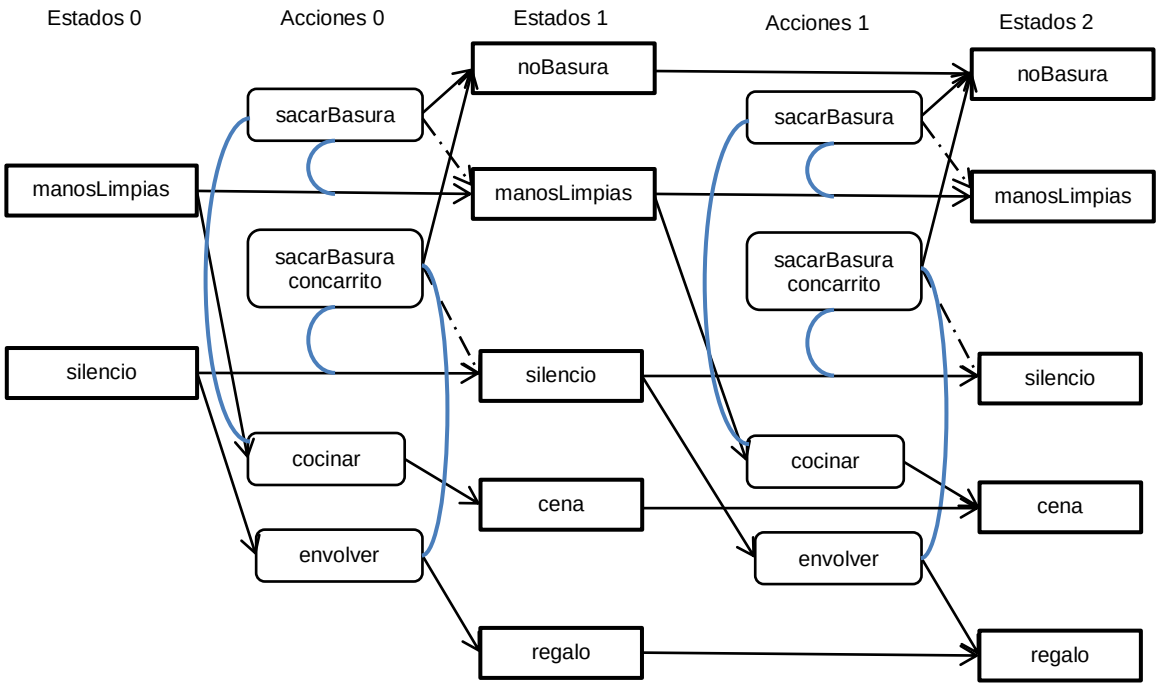


Figura 2.22. Grafo generado en el paso 1

¿Qué hemos hecho hasta ahora?

1. Colocar los estados iniciales en el paso 0.
2. Colocar las acciones posibles en el paso 0.
3. Ejecutarlas para obtener los estados del paso 1.
4. Comprobar los mutexs en el paso 1.
5. Si todos los estados objetivo están dentro de los estados del paso 1 sin mutex, retroceder en el grafo buscando una solución. Si se encuentra la solución, volver y finalizar.
6. Como no hemos encontrado la solución, extendemos el grafo un nivel más.

En Graphplan se hace un camino hacia adelante y otro hacia atrás: hacia adelante buscando los estados objetivo y hacia atrás comprobando si se han alcanzado esos estados sin inconsistencias. Ahora vamos a ver como se busca una solución. La idea es encontrar acciones consistentes en el paso n, luego en el nivel n-1, hasta encontrar un conjunto de acciones en cada paso que cumplan todos los estados objetivo.

Una descripción del algoritmo de búsqueda de la solución es ir recursivamente encontrando acciones que obtengan los métodos en el paso n, en el paso n-1 y así hasta conseguirlo:

Encontrar acciones que consigan cada objetivo O en el paso n.
Para cada acción A que haga que el objetivo G se consiga en el paso n.
Si A no está en exclusión mutua con otra acción en el paso n
Seleccionar esa acción
Finalmente
Si ninguna acción consigue el objetivo O
dar un paso hacia atrás buscando el objetivo O
Finalmente
Si se ha encontrado una acción para cumplir cada objetivo en el paso n
Entonces ir hacia atrás en las precondiciones de las acciones seleccionadas a n-1
Si no, buscar la siguiente solución en el paso n+1

La idea es seleccionar en el paso n-1 las acciones que nos permitan alcanzar todos los estados objetivo en el paso n, resolviendo un problema de satisfacción de restricciones (CSP). Las variables de este CSP son cada uno de los estados objetivo, el dominio el conjunto de acciones del paso n-1 que añaden un estado objetivo y las restricciones son los mutex del paso n-1. Si se encuentra la solución al CSP, se va hacia atrás para comprobar las precondiciones del paso n-1, si no se consigue, se genera el siguiente paso n+1.

En caso de tener que elegir entre realizar una acción y seguir un nodo noOp, siempre se eligen los noops. Esto evita que el plan contenga pasos redundantes.

Bueno, para aclarar lo que hemos descrito, vamos a seguir los pasos a ver hasta donde llegamos.

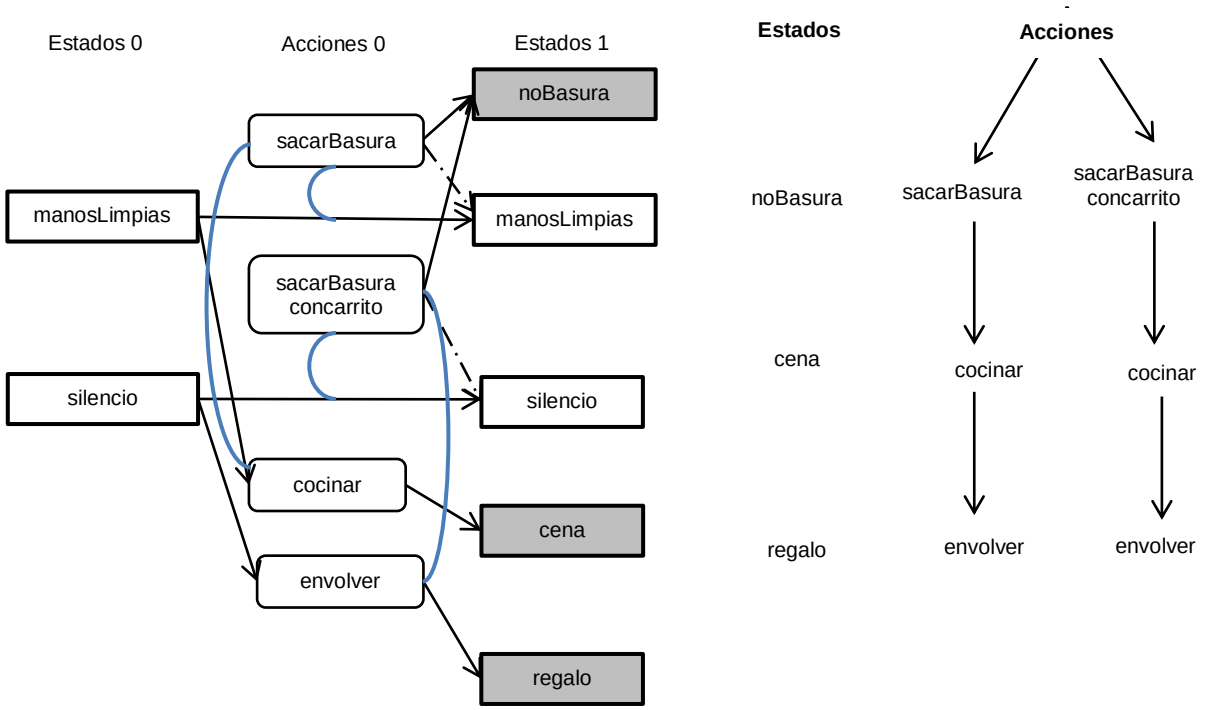


Figura 2.23. Grafo de planificación y árbol de acciones Inicio

A la izquierda tenemos nuestro conocido grafo de planificación. A la derecha, tenemos un árbol de acciones: los estados objetivo y, para cada uno, como podemos conseguirlo. ¿Cómo podemos conseguir que no tengamos basura? Pues tenemos dos opciones, o *sacarBasura* o *sacarBasuraconcarrito*. Ejecutemos la acción que ejecutemos, para conseguir los otros dos objetivos tenemos que ejecutar *cocinar* y luego *envolver.*

Una vez que tengo este árbol de acciones, lo podo usando los mutex. En este caso, ninguna de las dos ramas me sirve. En la primera, las acciones *sacarBasura* y *cocinar* están en mutex por que nos llenamos las manos. En la segunda, las acciones *sacarBasuraconcarrito* y *envolver* están en mutex porque al sacar el carrito pierdo el *silencio*.

Por tanto, tengo que generar el siguiente paso en el algoritmo, aplicando acciones y obteniendo estados. Una vez obtenido el siguiente estado, construyo ell árbol y empezamos a podar.

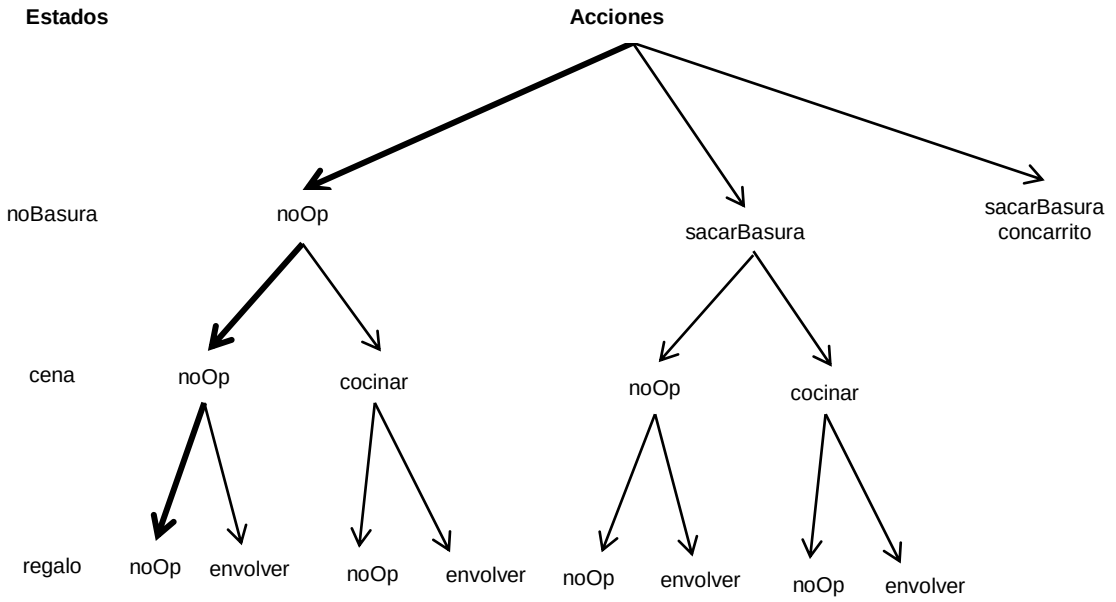


Figura 2.24. Árbol de acciones completamente desarrollado – siguiendo acciones noOp en el primer recorrido

La idea es ir tomando todas las combinaciones posibles de acciones e ir probándolas hasta dar con una correcta. La primera que nos encontramos en el árbol es noOp, noOp, noOp. Podemos imaginar que si en el paso 0 no hemos conseguido una solución, si no hacemos nada en el paso 1 no la vamos a conseguir. Lo que hago es volver al nivel 0 siguiendo las líneas noOp. Así, tengo:

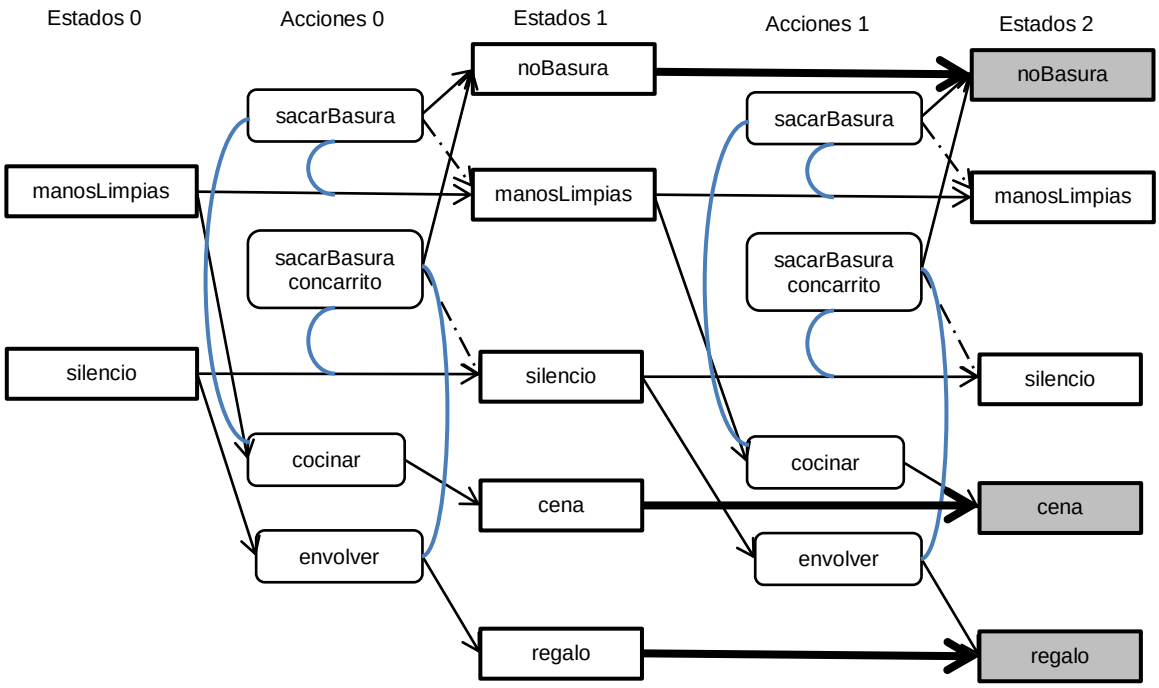


Figura 2.25. Grafo de planificación tras aplicar noOps

Éste es el camino hacia atrás o backtrack. Una vez que hemos vuelto hacia atrás tenemos que volver a mirar el árbol de nivel 0 del siguiente modo.

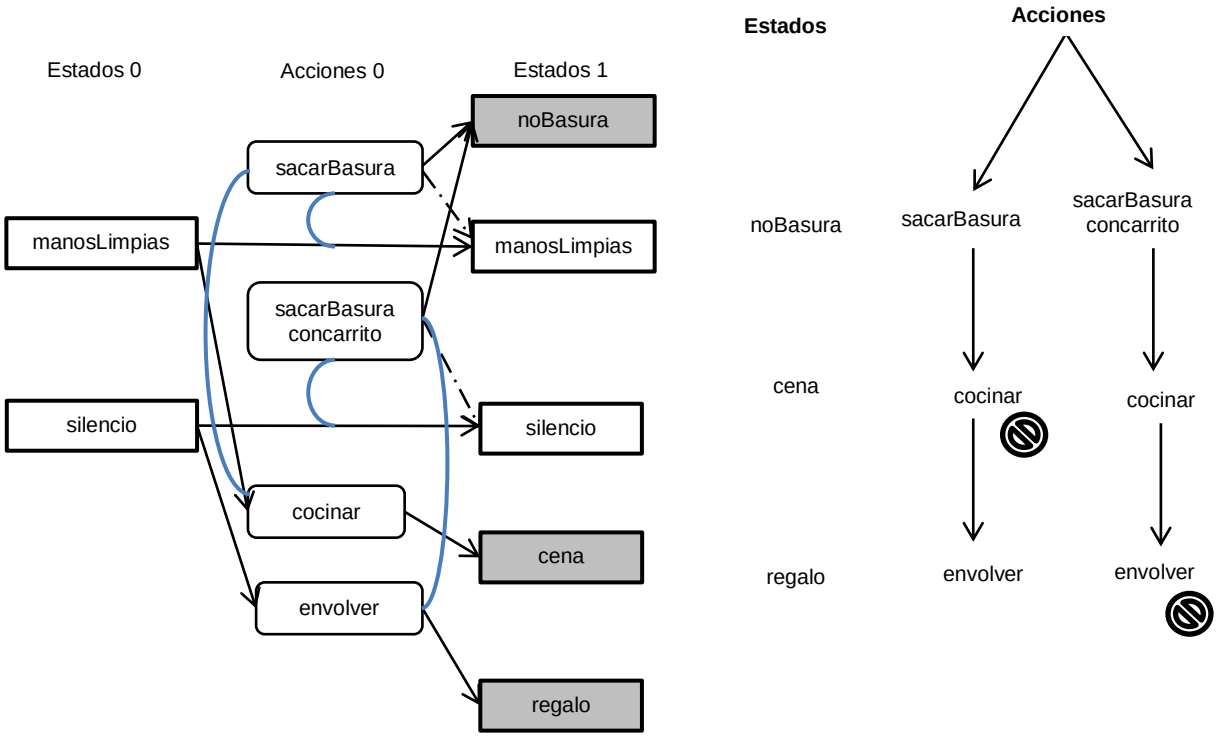


Figura 2.26. Backtrack: comprobar los mutex y detectar inconsistencias

Vuelvo a comprobar los mutex y veo que no puedo cocinar y sacar la basura en el mismo paso porque al sacar la basura me ensucio las manos para cocinar. Y tampoco puede envolver el regalo si saco la basura con el carrito porque hago ruido.

Ahora tengo que volver a buscar en el árbol de acciones y probar la siguiente combinación.

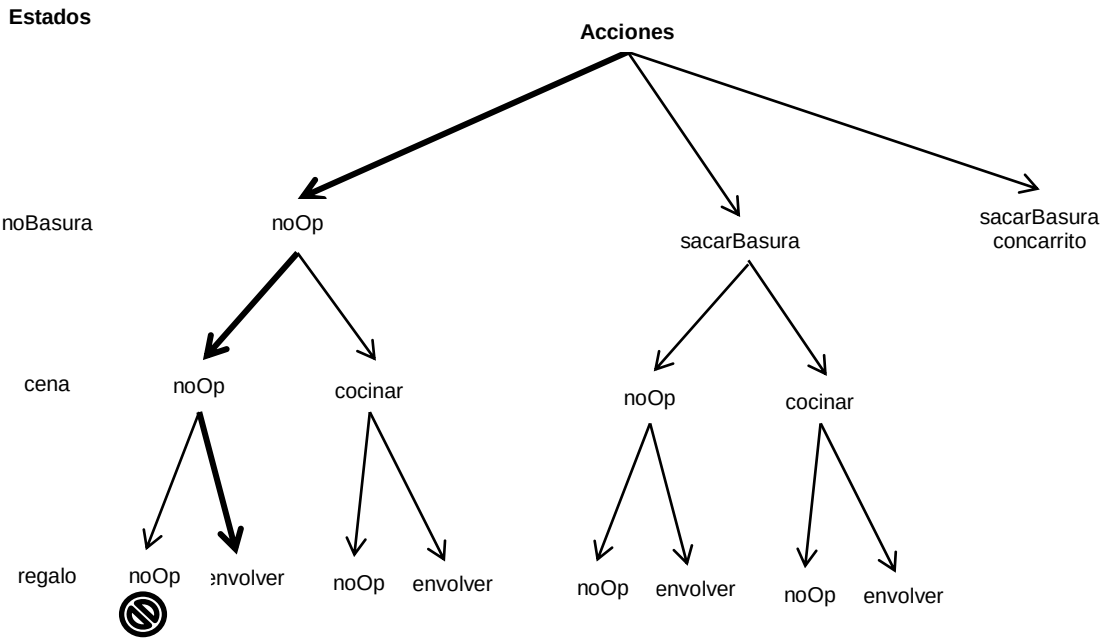


Figura 2.27. Árbol de acciones: segundo recorrido posible del mismo

Ahora ejecuto noOp, noOp y envolver. Veamos a donde llegamos en el nivel 2.

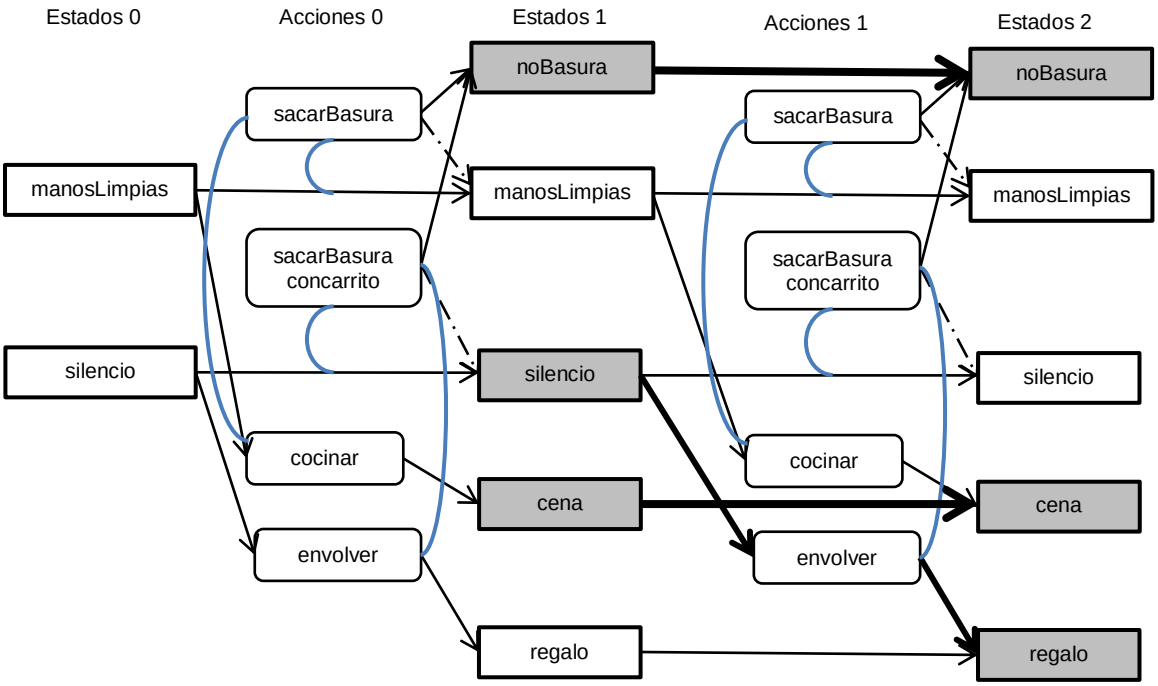


Figura 2.28. Grafo de planificación del segundo recorrido del árbol

Ahora al volver de ejecutar la acción *envolver* tenemos en los estados del nivel 1 *noBasura, silencio* y *cena.* Así es como funciona el algoritmo, va buscando conjuntos de estados de niveles anteriores que se puedan conseguir con acciones de niveles anteriores hasta dar con la correcta. Como vemos, ha cambiado un estado, ¿podemos conseguir estar en silencio ejecutando un conjunto de acciones de nivel 0? Veamos.

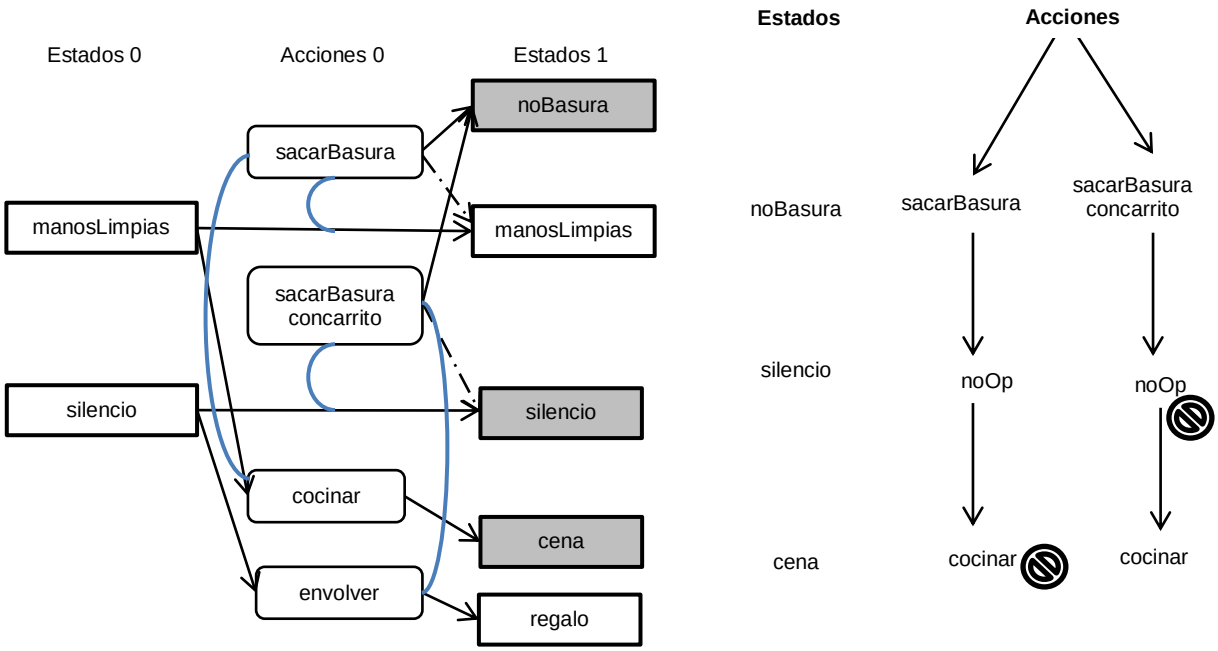


Figura 2.29. Backtrack del segundo recorrido del árbol

Observemos en el árbol como para conseguir el estado *silencio* no necesita realizar ninguna operación en el estado 0. Veamos si algún conjunto de acciones me permite llevar a ese estado. Si saco la basura no puedo cocinar porque tengo las manos sucias y si saco la basura con el carrito no estoy en silencio, que es un estado objetivo en este nivel. Así que no hay ningún conjunto de estados que me permita llegar a los estados objetivo. Paso a la siguiente rama del árbol.

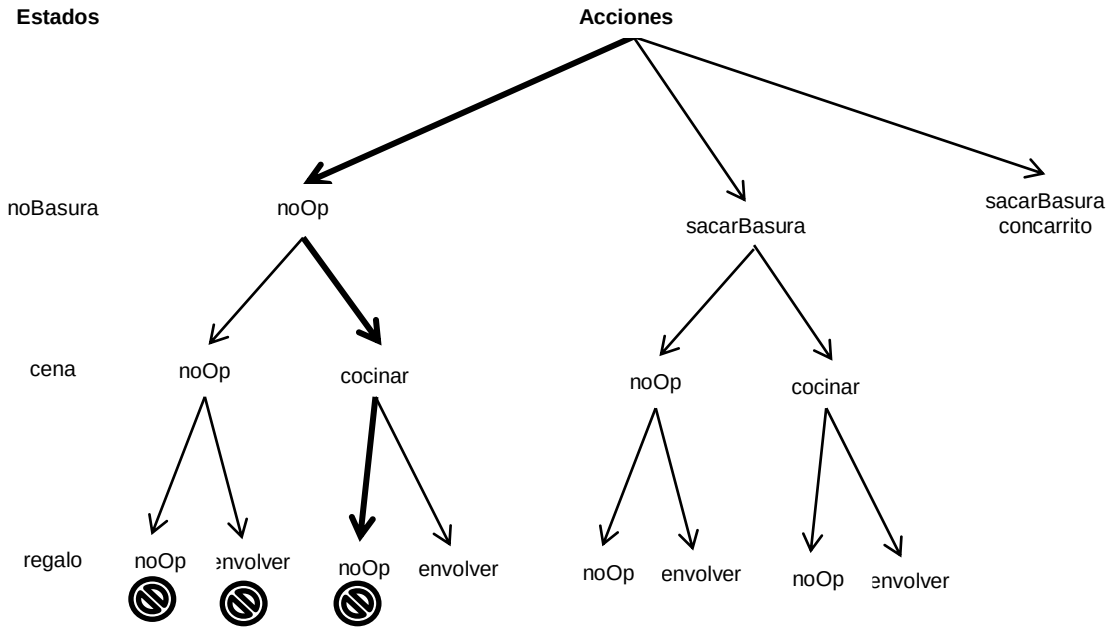


Figura 2.30. Tercer camino recorrido en el árbol

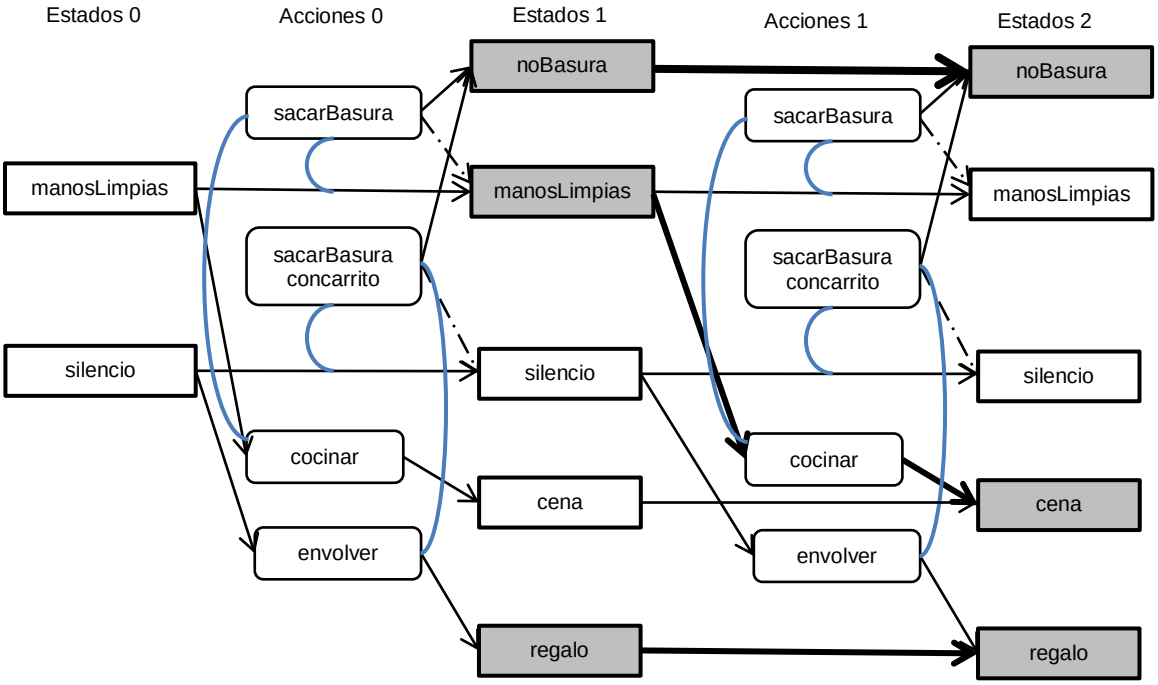


Figura 2.31. Grafo de planificación para el tercer recorrido del árbol

Ahora tenemos otro grupo de estados a conseguir ejecutando acciones en el nivel 0. Si queremos cocinar en el nivel 1, tenemos que llegar con las manos limpias, así en al terminar el nivel 0 tenemos que no tener basura, tener las manos limpias y el regalo preparado. ¿Podemos conseguirlo con un grupo de acciones del nivel 0?

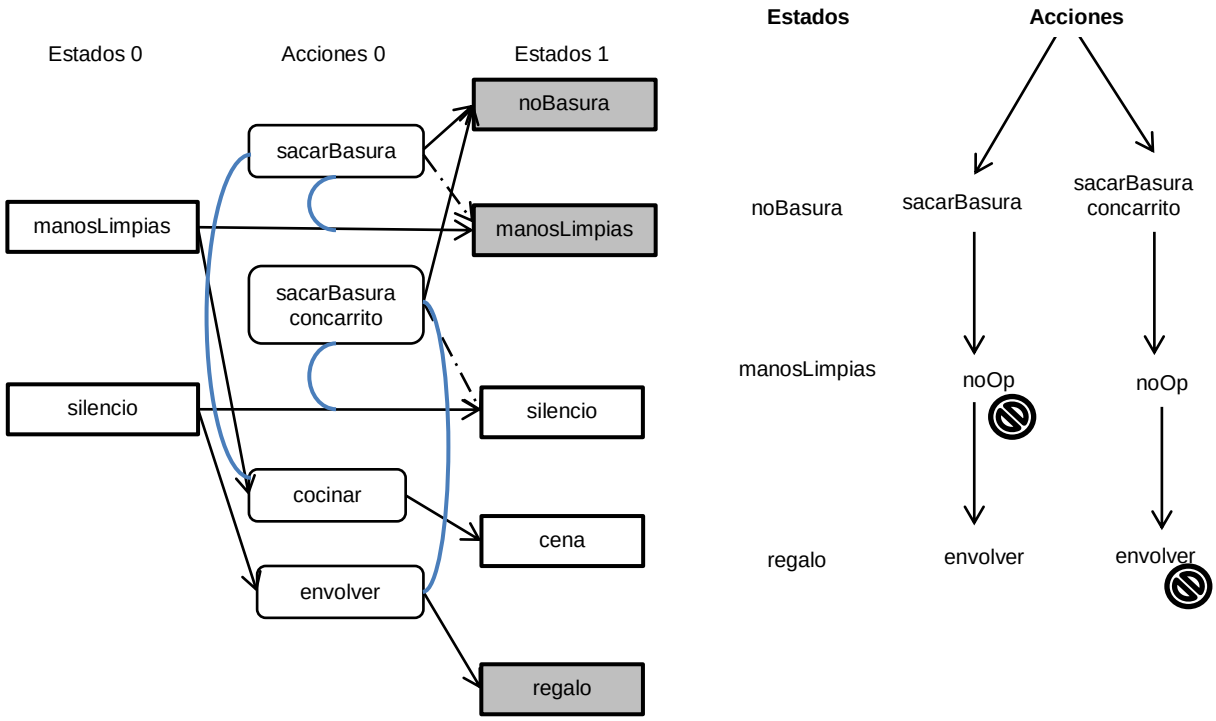


Figura 2.32. Backtrack del tercer recorrido del árbol

Veamos. Si saco la basura no tengo las manos limpias, que es un estado que necesito alcanzar así que sacar la basura no es una opción. Y si saco la basura con el carrito, hago ruido, rompo el silencio y ya no puedo envolver el regalo, para lo que necesitaba el silencio. Otra camino que no nos lleva al objetivo. Vamos a por otro recorrido en el árbol.

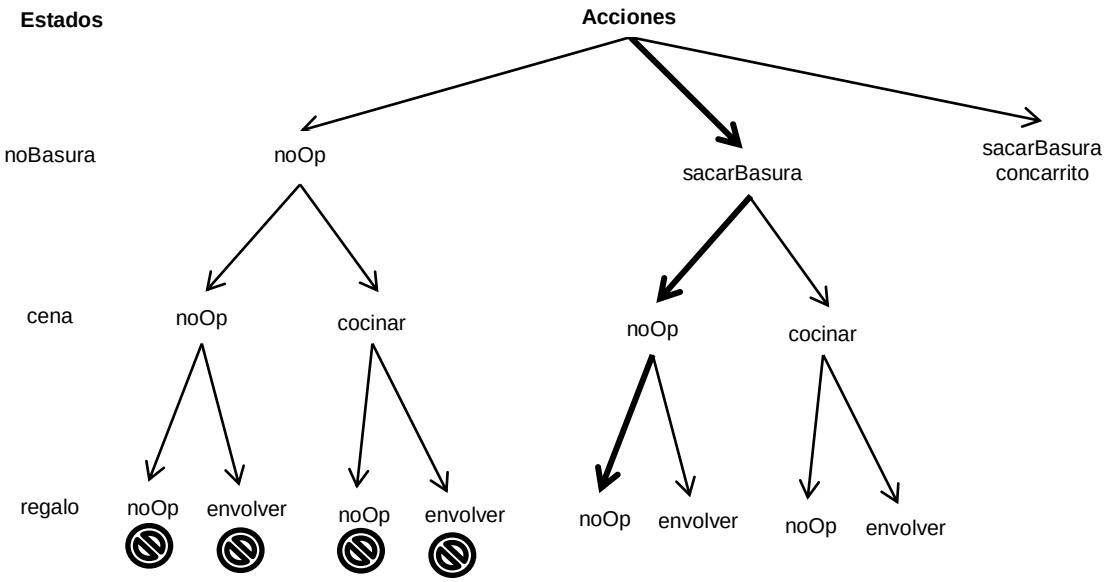


Figura 2.33. Quinto recorrido en el árbol

Vamos a ver si podemos conseguir nuestro objetivo sacando la basura en el nivel 1 de acciones. Si volvemos hacia atrás en el primer en el nivel tenemos:

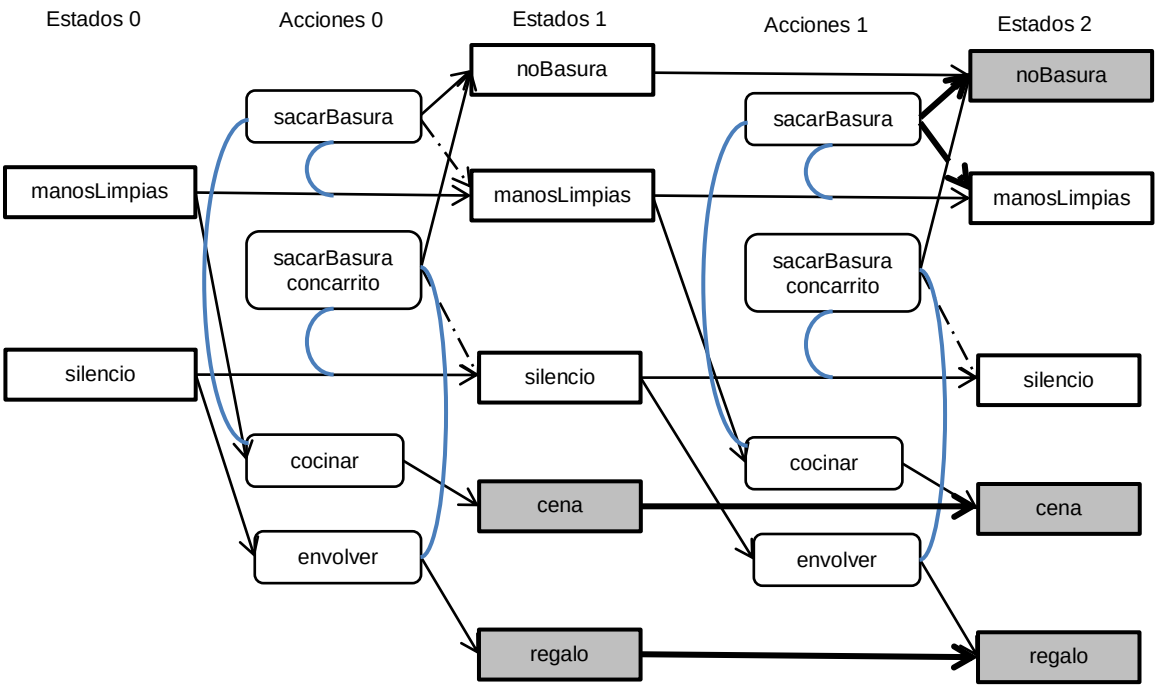


Figura 2.34. Grafo de planificación para quinto recorrido del árbol

En el nivel 1 saco la basura así que solo tengo que conseguir ejecutando acciones del nivel 0 la cena y el regalo envuelto. ¿Será posible?

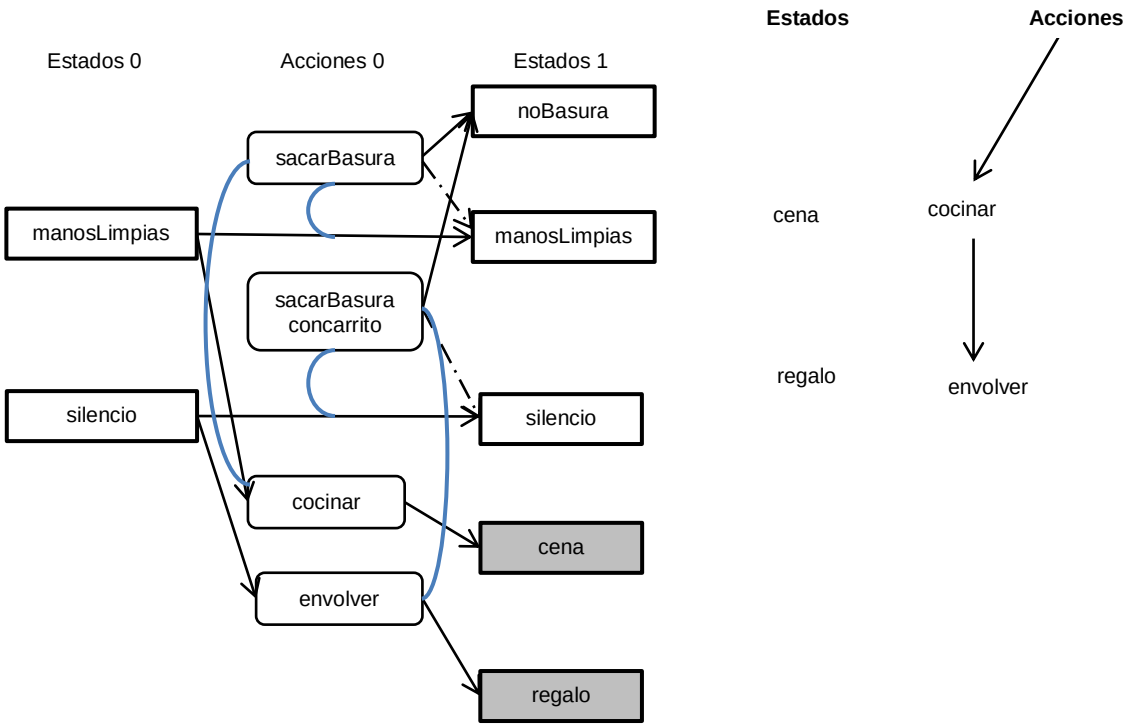


Figura 2.35. Backtrack para el quinto recorrido del árbol

Para conseguir la cena lo único que podemos hacer en el nivel 0 es cocinar y para conseguir el regalo lo único que podemos hacer es envolver. Y, afortunadamente, no hay mutex entre estas dos acciones: si cocinamos no hacemos ruido y si envolvemos el regalo no nos llenamos las manos, así que podemos realizar estas dos acciones a la vez en el nivel 0. Estamos ante un estado consistente. Por tanto la solución es cocinar y envolver y, por último, sacar la basura a mano (aunque vamos a llegar a casa con las manos sucias… qué poco romántico). Aquí tenemos el grafo de planificación con la solución.

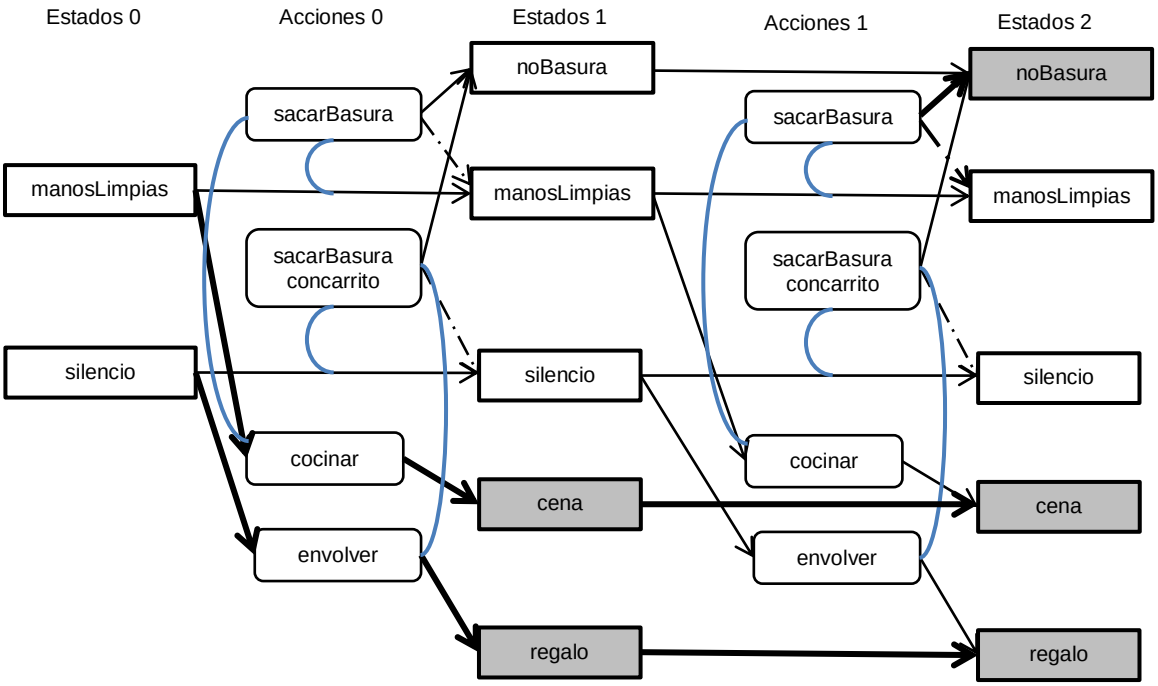


Figura 2.36. Grafo de planificación completo para la solución

Hemos visto que, conforme vamos hacia atrás, comprobamos si los conjuntos de estados en cada nivel satisfacen la solución al problema. Para acelerar un poco el algoritmo, podemos guardar en cada nivel los conjuntos de estados que, en ese nivel, no satisfacen el problema. Así, si un recorrido del árbol nos lleva a los mismos estados en cierto nivel que otro que no tenía solución, no tenemos que seguir hacia atrás para saber que no la tiene, lo podemos descartar inmediatamente. A eso le vamos a llamar **lista de subobjetivos no consistentes.** Un subobjetivo es un conjunto de estados al que tenemos que llegar en el nivel n para que podamos, desde ahí y ejecutando una serie de acciones, llegar a los objetivos del último nivel. La técnica para crear estas listas se puede describir así:

- Si un conjunto de estados objetivo en la capa n no puede ser alcanzada, apuntarla en la lista de subobjetivos no consistentes para esa capa.
- Para cada nuevo subobjetivo que alcanzamos en una capa n, ver si está en la lista de subobjetivos no consistentes para ese nivel.
    - Si está, devolver error para ese recorrido.
    - Si no está, comprobar si es una solución

Vamos a seguir el algoritmo añadiendo la lista de subobjetivos no consistentes. Hemos visto que al hacer el primer recorrido del árbol no podemos llegar en el nivel 1 al conjunto de objetivos *noBasura, cena y regalo* con ningún conjunto de acciones del nivel 0, así que añado a la lista de subojetivos no consistentes de nivel 0 el conjunto:

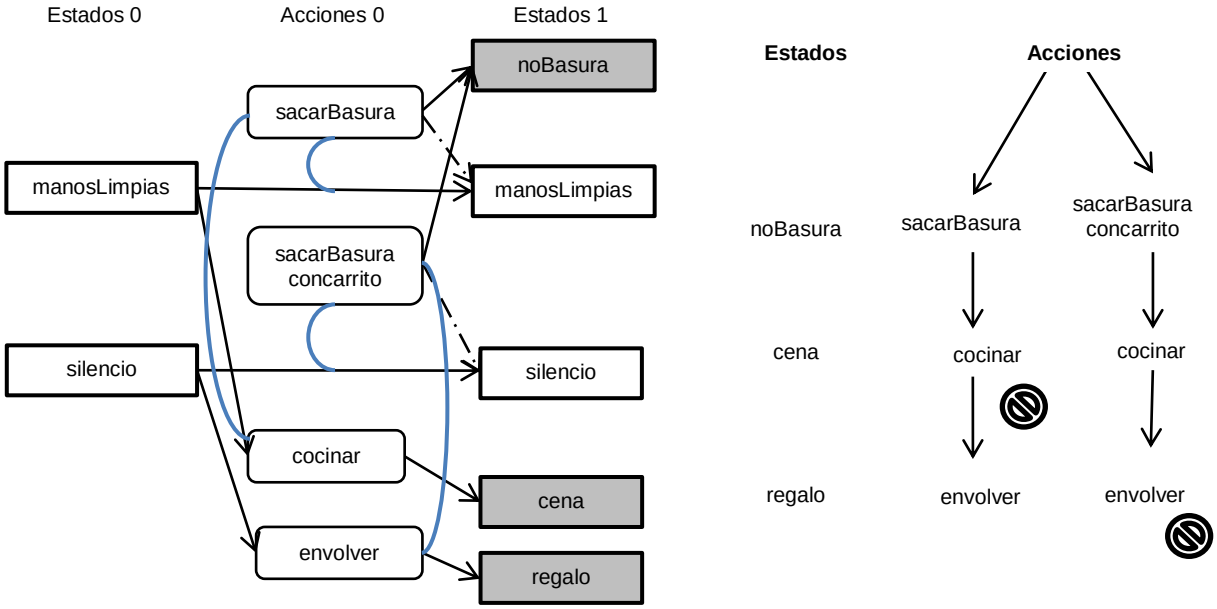


Figura 2.37. Generando lista de subobjetivos no consistentes

LISTA NIVEL 0 {
{noBasura,cena,regalo} }

En el siguiente recorrido del árbol hemos averiguado que no podemos tener en el estado 1 el subconjunto *noBasura, silencio* y *cena* con acciones del nivel 0, así que lo añadimos a la lista de nivel 0:

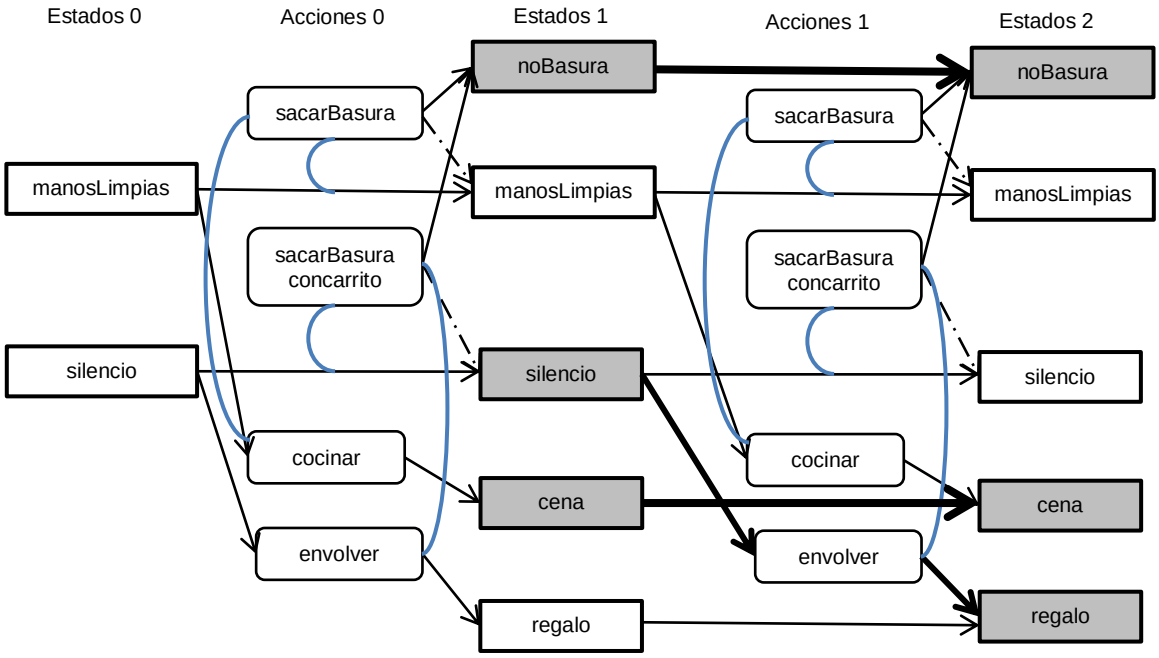


Figura 2.38. Grafo para generar lista de subojetivos no consistentes

LISTA NIVEL 0 {
{noBasura, cena, regalo}
{noBasura, silencio, cena}
}

Y en el siguiente recorrido añado a la lista el conjunto de estados {noBasura, manosLimpias, regalo}

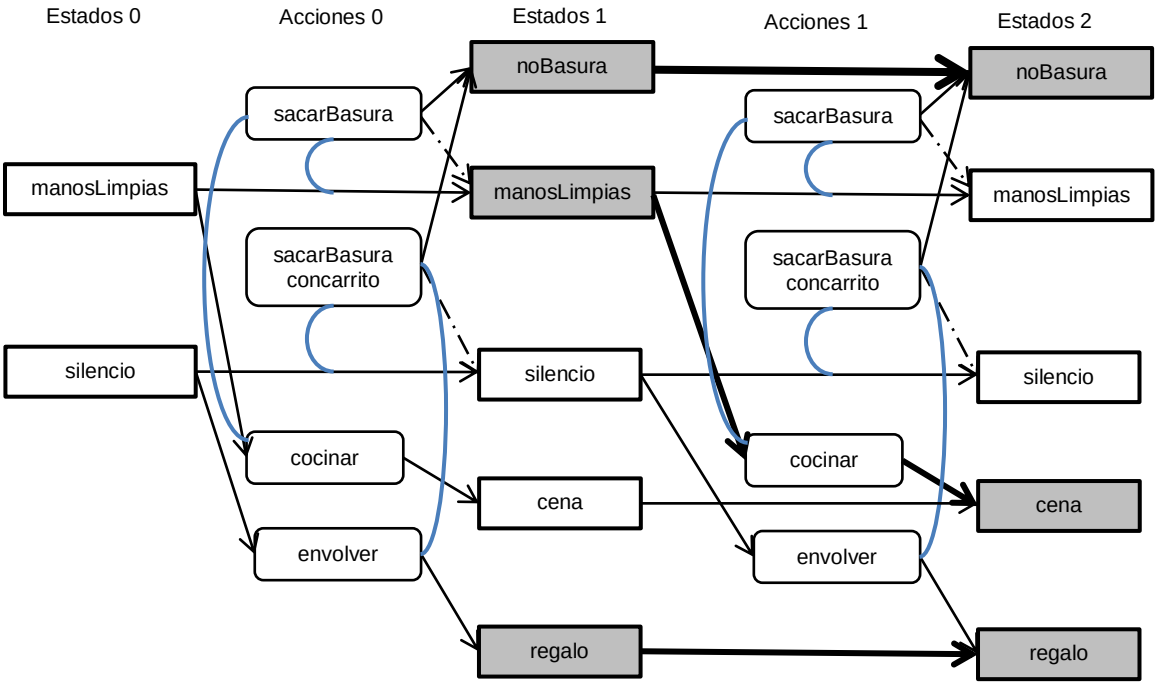


Figura 2.39. Grafo para generar lista de subobjetivos no consistentes (Paso 2)

LISTA NIVEL 0 {
{noBasura, cena, regalo}
{noBasura, silencio, cena}
{noBasura, manosLimpias, regalo} }

Cuando llego a la solución, antes de darla como buena, comparo su contenido en el nivel 0 con los conjuntos de estados del nivel 0. El conjunto de estados {cena,regalo} no aparece, así que puedo dar la solución como buena.

Algunas propiedades del algoritmo GraphPlan son:

- El número de estados va aumentando porque conforme un estado se añade a un nivel, no se vuelve a eliminar en ningún nivel posterior.
- El número de mutex se decrementa: una vez que un mutex desaparece no vuelve a aparecer.
- El algoritmo garantiza que las acciones que se puedan realizar en paralelo lo hagan de modo que los planes generados tardarán lo menos posible.
- Pero el algoritmo no garantiza planes óptimos secuencialmente, es decir, se pueden conseguir los objetivos en niveles posteriores usando menos acciones.

- En el árbol siempre se toman primero los caminos que tienen la mayor cantidad de noOps, esto es, siempre se favorece no realizar ninguna acción. Así, los planes generados no contienen pasos redundantes.
- El grafo de planificación, en algún momento, alcanza un punto fijo. Hay un punto a partir del cual los estados y los mutex no cambian.

En el ejemplo que nos ocupa hemos llegado a una solución bastante pronto y hemos terminado. Pero, ¿qué pasa si no hay solución? Graphplan tiene la propiedad de que solo devuelve error si no existe ningún plan que satisfaga los estados objetivo. Una forma simple de terminar el algoritmo sería ver si se ha alcanzado ese punto fijo, este nivel a partir del cual nada cambia y, si en ese punto fijo uno de los estados objetivo no se ha alcanzado o dos estados objetivo están en mutex, se devuelve error. Pero también tenemos que mirar esta lista de subobjetivos no consistentes que hemos ido creando porque puede mostrarnos que una supuesta solución que hemos encontrado en un nivel, cuando vayamos hacia atrás, veremos que no lo es.

Precisamente por estas listas es por lo que tenemos que seguir generando niveles más allá del punto fijo. Porque aunque los estados y los mutex no cambian, estas listas de subobjetivos no consistentes sí que cambian. Así, al añadir niveles estamos dando tiempo que permita a las acciones que se repartan en el tiempo para que no se ejecuten a la vez y así los conjuntos de estados que eran inalcanzables antes con más niveles no lo son. Además, esta lista decrece conforme vamos añadiendo niveles, de modo que, llegado un nivel determinado, el conjunto de estados que nos bloqueaba un plan desaparece. Entonces hemos encontrado una solución. Sin embargo, si llegamos a un punto fijo en el que estas listas también alcanzan un punto fijo y no cambian, podemos colegir que el problema no tiene solución.

2.6. PDDL: otro lenguaje para definir problemas de planificación

2.6.1. PDDL: introducción al lenguaje

PDDL son las siglas de Planning Domain Definition Language, Lenguaje de Definición de Dominios de Planificación. Es un lenguaje para la definición de problemas de planificación creado para la competición de planificación AIPS-98 que ha ido evolucionando en las sucesivas ediciones dela misma. Se basa en el lenguaje que hemos descrito anteriormente, STRIPS, y otros lenguajes de planificación como ADL (Action Description Language).Es el estándar de facto y lo soportan casi todos los planificadores. Para introducir la información en el planificador, el mundo a tratar se separa en dos archivos, un archivo de dominio donde están los tipos, los predicados y las acciones o esquemas de acción, y un archivo de problema con los objetos, el estado inicial de los mismos y el objetivo.

Un fichero de dominio de PDDL tiene la siguiente estructura:

El nombre del dominio.

```
(define (domain <nombre>)
```

Una línea de requerimientos:

```
(:requirements :adl :typing)
```

Una definición de tipos en el dominio:

```
(:types –bloque)
```

Los predicados posibles en el dominio:

```
(sobre ?a –bloque ?b –bloque)
```

Los esquemas de acción del dominio están formados por:

El nombre de la acción:

```
(:action <nombre>
```

La lista de parámetros:

```
:parameters (?a - tipo1 ?b –tipo2)
```

La precondición, que es un conjunto de fórmulas con and, or, not, forall y exists:

```
:precondition
(and <formula>… <formula>)
(or <formula> … <formula>)
(not <formula>)
(forall (?a –tipo1 … ?b –tipo2) <formula>)
(exists (?a –tipo1 … ?b –tipo2) <formula>)
```

Y los efectos, un conjunto de literales, efectos condicionales y cuantificadores:

```
:effect
(not <predicate>)
(and <efecto1> … <efecto2>)
(when <formula> <efecto>)
(forall (?a –tipo1 … ?b –tipo2 <efecto>)
```

Un fichero de problema de PDDL tiene la siguiente estructura:

```
(define (problema <nombre>)
(:domain <nombredominio>)
```

Definición de los objetos del problema con su correspondiente tipo.

```
(:objects a – tipo1 b – tipo2)
```

Estado inicial: conjunto de predicados verdaderos al principio.

```
(:init (limpio a) (sobre a b))
```

Objetivo: fórmulas que indican los predicados finales.

```
(:goal (and (formula1) (formula2))
```

Vamos a ver un ejemplo que acompaña a PDDL4J, el intérprete de GraphPlan que usaremos. Un caso de planificación está definido mediante dos archivos, en este ejemplo, *gripper.pddl* y *pbb1.pddl*. *gripper.pddl* es el archivo de dominio, que contiene los predicados y las acciones. *pbb1.pddl* es el archivo problema, que tiene los objetos, el estado inicial y el estado objetivo. Veamos como es el archivo *gripper.pddl*.

Un archivo de dominio siempre empieza con la definición del nombre del dominio, en este caso, gripper.

(define (domain gripper)

Los requerimientos indican que parte del lenguaje vamos a usar. PDDL es muy amplio y la mayoría de los planificadores solo soportan un subconjunto del mismo, así que aquí se señala que parte se va a usar. Los requerimientos más usados son:

- strips: el subconjunto más básico de PDDL, que usa solo strips.
- equality: se usa el símbolo = como de igualdad.
- typing: se usan tipos de datos en el dominio.
- ADL: se usa el lenguaje ADL.

```
(:requirements :strips)
```

Los predicados son los objetos y estados de estos objetos en el dominio que estamos definiendo. Así, en este mundo podemos tener 3 objetos: habitaciones, pelotas y el robot con su brazo o brazos robóticos. Hay 4 estados: si el robot está en una habitación, si una pelota está en una habitación, si es brazo robótico está libre y si el brazo está agarrando un objeto.

```
(:predicates (room ?r)      r es una habitación
      (ball ?b)             b es una pelota
      (gripper ?g)          g es un brazo robótico
      (at-robby ?r)         el robot está en la habitación r
      (at ?b ?r)            la pelota b está en la habitación r
      (free ?g)             el brazo robótico está libre
      (carry ?o ?g))        el brazo robótico g está cogiendo el objeto o
```

Las acciones tienen unos parámetros, es decir, unos objetos sobre los que actúan, unas precondiciones, estados que tienen que ser verdaderos para que se pueda ejecutar la acción, y efectos o postcondiciones, estados que son verdaderos después de la ejecución de la acción y estados que dejan de ser verdaderos después de la ejecución de la acción.

La acción mover tiene dos parámetros, un lugar de salida y un lugar de llegada. Los lugares de salida y llegada deben ser habitaciones y el robot tiene que estar en la habitación de salida. El efecto es que el robot acaba en la habitación de llegada y ya no está en el de salida.

```
(:action move
 :parameters  (?from ?to)
 :precondition (and  (room ?from) (room ?to) (at-robby ?from))
 :effect (and  (at-robby ?to) (not (at-robby ?from))))
```

La acción coger requiere un objeto, una habitación y un robot. El objeto tiene que ser una pelota, la pelota tiene que estar en la habitación y el robot también. Además, el robot tiene que estar libre.

```
(:action pick
 :parameters (?obj ?room ?gripper)
 :precondition  (and  (ball ?obj) (room ?room) (gripper ?gripper)
 (at ?obj ?room) (at-robby ?room) (free ?gripper))
 :effect (and (carry ?obj ?gripper) (not (at ?obj ?room))
             (not (free ?gripper))))
```

La acción soltar require un objeto, una habitación y el robot. El objeto es una pelota, el robot tiene la pelota cogida y está en la habitación. El robot suelta la pelota y la pelota queda en la habitación, el robot queda libre y ya no está cogiendo la pelota.

```
 (:action drop
   :parameters  (?obj  ?room ?gripper)
   :precondition  (and  (ball ?obj) (room ?room) (gripper ?gripper)
                     (carry ?obj ?gripper) (at-robby ?room))
   :effect (and (at ?obj ?room) (free ?gripper) (not (carry ?obj
?gripper))))
)
```

Ahora vemos como es el archivo *pbb1.pddl.*

```
(define (problem pb1)
```

Se define el nombre del problema.

```
(:domain gripper)
```

Se indica el dominio al que pertenece el problema.

```
(:requirements :strips)
```

Se toma el subconjunto del lenguaje en strips.

```
 (:objects roomA roomB Ball1 Ball2 left right)
```

Estos son los objetos del problema: tenemos dos habitaciones, dos pelotas, y dos brazos robóticos.

```
 (:init
      (room roomA)
      (room roomB)
      (ball Ball1)
      (ball Ball2)
      (gripper left)
      (gripper right)
      (at-robby roomA)
      (free left)
      (free right)
      (at Ball1 roomA)
      (at Ball2 roomA))
```

En el estado inicial tenemos el robot en la habitación A con los brazos izquierdo y derecho libres y las dos pelotas en la habitación A.

```
 (:goal (and (at Ball1 roomB)
        (at Ball2 roomB)))
)
```

En el estado final, queremos tener las dos pelotas en la habitación B.

En (8) M. Veloso introduce algunas extensiones de PDDL como el tipado y el coste de las acciones para que el planificador pueda elegir entre varios planes correctos, el más efectivo. Veamos el archivo typed-gripper-domain.pddl (9), un dominio con tipos y costes:

```
(define (domain typed-gripper)
```

En los requerimientos se ha especificado tipado de objetos y especificación de coste de las acciones.

```
  (:requirements :typing :action-costs)
```

Defino tres tipos de objetos, habitaciones, pelotas y brazos robotizados.

```
  (:types room ball gripper)
```

Incluso puede definir constantes: defino dos constantes, left y right, que son del tipo brazo robotizado ya que son los únicos que voy a tener en el dominio.

```
  (:constants left right - gripper)
```

Y defino los predicados:

```
  (:predicates
```

La diferencia es que ahora los objetos están tipados, por ejemplo, en la siguiente línea defino r del tipo room, es decir, tiene que ser una habitación.

```
    (at-robby ?r - room)
    (at ?b - ball ?r - room)
    (free ?g - gripper)
    (carry ?o - ball ?g - gripper)  )
```

Además, se define *total-cost* que permitirá calcular el coste de la solución que el planificador devuelva.

```
  (:functions
    (total-cost)
  )
```

Las acciones producen modificaciones en el valor de la función total-cost.

```
;;un robot puede mover de un sitio ?from a un sitio ¿to
  (:action move
    :parameters (?from ?to - room)
    :precondition (at-robby ?from)
    :effect (and (at-robby ?to)
                 (not (at-robby ?from))
```

La acción mover hace que el coste total del plan que usa esta acción aumente en 10.

```
    (increase (total-cost) 10))
  )
```

;;el robot puede recoger objetos del suelo con el gripper

```
  (:action pick
    :parameters (?obj - ball ?room - room ?gripper - gripper)
    :precondition (and (at ?obj ?room) (at-robby ?room) (free ?gripper))
    :effect (and (carry ?obj ?gripper ) (not (at ?obj ?room))
            (not (free ?gripper))
```

Sin embargo, la acción de recoger un objeto solo tiene un coste de 1.

```
    (increase (total-cost) 1))
 )
```

;;El robot puede soltar un objeto en una habitación del gripper

```
  (:action drop
    :parameters (?obj - ball ?room - room ?gripper - gripper)
    :precondition (and (carry ?obj ?gripper) (at-robby ?room))
    :effect (and (at ?obj ?room) (free ?gripper)
    (not (carry ?obj ?gripper))
Y la acción de soltar tiene un coste de 1.
    (increase (total-cost) 1))
  )
)
```

Y en el archivo problema las modificaciones son:

```
(define (problem typed-gripper)

  (:domain typed-gripper)
  (:objects
```

Cada objeto es de un tipo.

```
    rooma roomb - room
    ball1 ball2 ball3 ball4 - ball
    left right - gripper)
```

Y la métrica, que indica el objetivo del problema, es que el coste total sea el menor posible (minimize). Esta información se pasa al planificador para que su elección de acciones sea la óptima para minimizar el coste.

```
  (:metric minimize (total-cost))

  (:init
```

El coste total al empezar el problema es 0.

```
   (= (total-cost) 0)
      (at-robby rooma)
      (free left)
      (free right)
      (at ball1 rooma)
      (at ball2 rooma)
      (at ball3 rooma)
      (at ball4 roomb))
```

```
  (:goal (and (at ball1 roomb)
              (at ball2 roomb)
              (at ball3 roomb)
              (at ball4 roomb)))
)
```

La sintaxis de la definición de un tipo es

```
(:types tipo1 tipo2 tipo3)
```

con todos los tipos de objetos que vamos a tener en el dominio.

En (10) se define una constante como un símbolo que tendrá el mismo significado para todos los problemas del dominio. En nuestro ejemplo, los brazos robóticos left y right van a ser constantes para todos los problemas porque tenemos un robot que tiene dos brazos robóticos. El número de habitaciones y pelotas puede cambiar. El número de brazos no lo va a hacer.

Total-cost es un *fluente*, un objeto que puede fluir libremente, o sea, cambiar. La única diferencia de total-cost con los estados, que son también fluentes, es que no tiene como valores verdadero o falso, sino un número entero positivo. Al definir el problema, lo primero que hacemos es inicializar el fluente total-cost a 0. Luego en las distintas acciones que llevamos a cabo podemos modificar el valor de ese coste: en nuestro caso, mover el robot de una habitación a otra incrementa el coste en 10, mientras que tomar o soltar un objeto solo cuesta 1. Finalmente, definimos con una métrica nuestro objetivo con este *fluente* que es que tenga el mínimo valor posible.

En (11) podemos ver un ejemplo del uso de funciones en PDDL en el siguiente ejemplo:

```
(define (domain test-domain)
(:requirements :typing :equality :conditional-effects
:fluents)
(:types car box)
(:constants goldie - car)
(:predicates (parked ?x - car) (holding ?x - box)
(in ?x - box ?y - car))
```

La función fuel-level se aplica sobre un objeto de tipo coche y nos da, eso, la cantidad de gasolina que tiene el coche.

```
(:functions (fuel-level ?x - car))
(:action load
:parameters (?x - box ?y - car)
:precondition (and (holding ?x) (parked ?y))
:effect (and (in ?x ?y)
```

En este ejemplo, además de decir que el objeto x está en el coche y, tenemos que añadir que el objeto y no está en ninguno de los otros coches que hayamos definido en el problema.

```
(forall (?z - car)
(when (not (= ?z ?y))
(not (in ?x ?z))))))
(:action refuel
:parameters (?x - car)
```

Si el nivel de gasolina del coche está por debajo de 10, y ésta es la utilidad de la función, se incrementa el nivel en 1.

```
:precondition (< (fuel-level ?x) 10)
:effect (increase (fuel-level ?x) 1)))
```

2.6.2. Un ejemplo de PDDL: los zumos

En (12) el Dr. Zeyn Saigol introduce una implementación de GraphPlan con un ejemplo bastante interesante con funciones. Analicemos el código. Como siempre tenemos un archivo de dominio llamado *jugs.pddl*.

```
(define (domain Jugs)
```

Este dominio está tipado, tiene funciones de coste y optimización y efectos condicionales.

```
(:requirements :typing :fluents :conditional-effects)
```

Hay dos tipos de objetos: lo que es una jarra y lo que no lo es.

```
(:types jug nonjug)
```

Tenemos dos funciones, una para obtener la capacidad de la jarra y otra para obtener la cantidad de líquido que contiene la misma.

```
(:functions (capacity ?j - jug)
         (contents ?j - jug)
            )
```

Y aquí definimos las acciones. La acción fill tiene como parámetro una jarra, y, para poder llenarla, el líquido contenido ha de ser menor que su capacidad porque si no la lleno más allá de su capacidad y derramo el líquido y un efecto, que es asignar al contenido la capacidad, de modo que queda llena.

```
(:action fill
 :parameters (?j - jug)
 :precondition (< (contents ?j) (capacity ?j))
 :effect (and (assign (contents ?j) (capacity ?j))))
```

La acción empty (vaciar) tiene como parámetro una jarra y como precondición que tenga algo de líquido. Se asigna el valor de 0 como efecto.

```
(:action empty
 :parameters (?j - jug)
 :precondition (> (contents ?j) 0)
 :effect (and (assign (contents ?j) 0)))
```

La acción pour (verter) toma una cantidad de líquido de una jarra y lo echa en otra. Tiene como parámetro las dos jarras y el contenido de la primera ha de ser mayor de 0, es decir, tiene que tener algo.

```
(:action pour
 :parameters (?j1 ?j2 - jug)
 :precondition (> (contents ?j1) 0)
 :effect (and
```

Y aquí vamos con los efectos condicionales. Si (when) la suma de los contenidos de las dos jarras no supera la capacidad de la segunda jarra, que es donde voy a echar la primera, pongo el contenido de la primera jarra a 0 e incremento el contenido de la jarra 2 con el líquido de la jarra 1.

```
(when (<= (+ (contents ?j1) (contents ?j2)) (capacity ?j2))
(and (assign (contents ?j1) 0)
(increase (contents ?j2) (contents ?j1))))
```

Si entre las dos hacen más de la capacidad de la segunda, se va a derramar, así que me quedo en el límite, pongo el contenido de la jarra 2 a su capacidad, es decir, la lleno, y a la jarra 1 le resto lo que he echado, que era la capacidad de la jarra 2 menos el contenido que tenía antes.

```
(when (> (+ (contents ?j1) (contents ?j2)) (capacity ?j2))
(and (assign (contents ?j2) (capacity ?j2))
(decrease (contents ?j1) (- (capacity ?j2) (contents
?j2)))))))

)
```

Y el archivo de problema *jug1.pddl*.

```
(define (problem jugs1)
    (:domain Jugs)
    (:objects jug1 jug2 - jug)
```

Tengo dos jarras vacías, una de 5 litros y otra de 3 litros.

```
    (:init (= (capacity jug1) 5)
           (= (capacity jug2) 3)
           (= (contents jug1) 0)
           (= (contents jug2) 0)
    )
```

El objetivo es que la primera jarra tenga 1 litro.

```
    (:goal (and (= (contents jug1) 1)))
```

El objetivo es minimizar el tiempo total.

```
    (:metric minimize total-time)
)
```

La solución que da este interpretador de pddl es bastante completa, indicando las acciones, los mutex de acciones, los mutex de literales y los literales totales. La solución que nos da es la siguiente:

```
fill(jug2)
```

```
pour(jug2,jug1)
fill(jug2)
pour(jug2,jug1)
empty(jug1)
pour(jug2,jug1)
```

2.6.3. Otro ejemplo de PDDL: el almacén

El almacén es un ejemplo muy interesante para ver como usar una variable para optimizar el problema. En este caso, se optimiza el gasto en gasolina.

El archivo de dominio es éste:

```
(define (domain Depot)
(:requirements :typing :fluents)
```

En este ejemplo se usan subtipos. Tenemos dos tipos principales, place y locatable, que podemos traducir como lugar y localizable. Los objetos depot (depósito) y distributor(distribuidor) son places, es decir, lugares. Los objetos truck (camión), hoist (montacargas) y surface (superficie) son locatables, esto es, objetos que pueden encontrarse en un lugar. Por último, los objetos pallet(palé) y crate(cajón) son del tipo surface (superficie).

```
(:types place locatable - object
          depot distributor - place
          truck hoist surface - locatable
          pallet crate - surface)
```

El predicado posn indica que el objeto localizable x está en el lugar y.

```
(:predicates (posn ?x - locatable ?y - place)
```

On indica que un cajón está sobre una superficie.

```
  (on ?x - crate ?y - surface)
```

In indica que un cajón tiene dentro un camión.

```
  (in ?x - crate ?y - truck)
```

Lifting indica que una caja tiene un montacargas encima.

```
  (lifting ?x - hoist ?y - crate)
```

Available indica si un montacargas está libre.

```
  (available ?x - hoist)
```

Clear indica si una superficie está libre.

```
  (clear ?x - surface)
```

)

Tenemos algunas variables:

```
(:functions
```

Cada camión tiene un límite de carga.

```
 (load_limit ?t - truck)
```

Current_load es la carga actual en kilos dentro del camión.

```
 (current_load ?t - truck)
```

El peso de cada cajón.

```
 (weight ?c - crate)
```

El coste de gasolina de una operación.

```
      (fuel-cost)
)
```

Conducir lleva un camión de un lugar a otro. Ya hemos visto que los lugares pueden ser de distintos tipos. El subtipado nos permite especificar movimientos entre distintos tipos de objetos sin tener que definir distintas operaciones. Así, un camión de puede mover entre depósitos y distribuidores. El camión tiene que estar en el primer lugar y los efectos son que el camión deja de estar en ese lugar, se mueve al otro lugar y el incremento de coste de gasolina, que va a ser el parámetro a minimizar, se incrementa en 10 unidades.

```
(:action Drive
:parameters (?x - truck ?y - place ?z - place)
:precondition (and (posn ?x ?y))
:effect (and (not (posn ?x ?y)) (posn ?x ?z)
          (increase (fuel-cost) 10)))
```

La acción levantar requiere un montacargas, un cajón, una superficie y un lugar. El montacargas x ha de estar en en el lugar p y tiene que estar libre. La caja ha de estar en la misma posición, el cajón tiene que estar en el montacargas y el cajón tiene que estar libre. El efecto es que el cajón deja de estar en la posición p, el cajón deja de estar libre, el montacargas deja de estar libre también y el montacargas está levantando el cajón, la superficie donde estaba el cajón queda libre y el cajón deja de estar sobre la superficie. Esta operación tiene un gasto de fuel de 1.

```
(:action Lift
:parameters (?x - hoist ?y - crate ?z - surface ?p - place)
:precondition (and (posn ?x ?p) (available ?x) (posn ?y ?p)
(on ?y ?z) (clear ?y))
:effect (and (not (posn ?y ?p)) (lifting ?x ?y) (not (clear
?y)) (not (available ?x)) (clear ?z) (not (on ?y ?z))
(increase (fuel-cost) 1)))
```

La acción soltar es justamente la opuesta a levantar y los efectos son justo los contrarios.

```
(:action Drop
:parameters (?x - hoist ?y - crate ?z - surface ?p - place)
:precondition (and (posn ?x ?p) (posn ?z ?p) (clear ?z) (lifting ?x ?y))
:effect (and (available ?x) (not (lifting ?x ?y)) (posn ?y ?p) (not (clear
?z)) (clear ?y)(on ?y ?z)))
```

La acción cargar requiere de un cajón montado en un montacargas y un camión en un lugar. El montacargas y el camión han de estar en el mismo lugar y el cajón ha de estar en el montacargas. Además, se impone la condición de que el peso del cajón más lo que hay ya en el camión no puede sobrepasar el límite de carga del camión. El efecto es que la caja deja el montacargas y se coloca en el camión, y se incrementa la carga del camión en el peso de la caja.

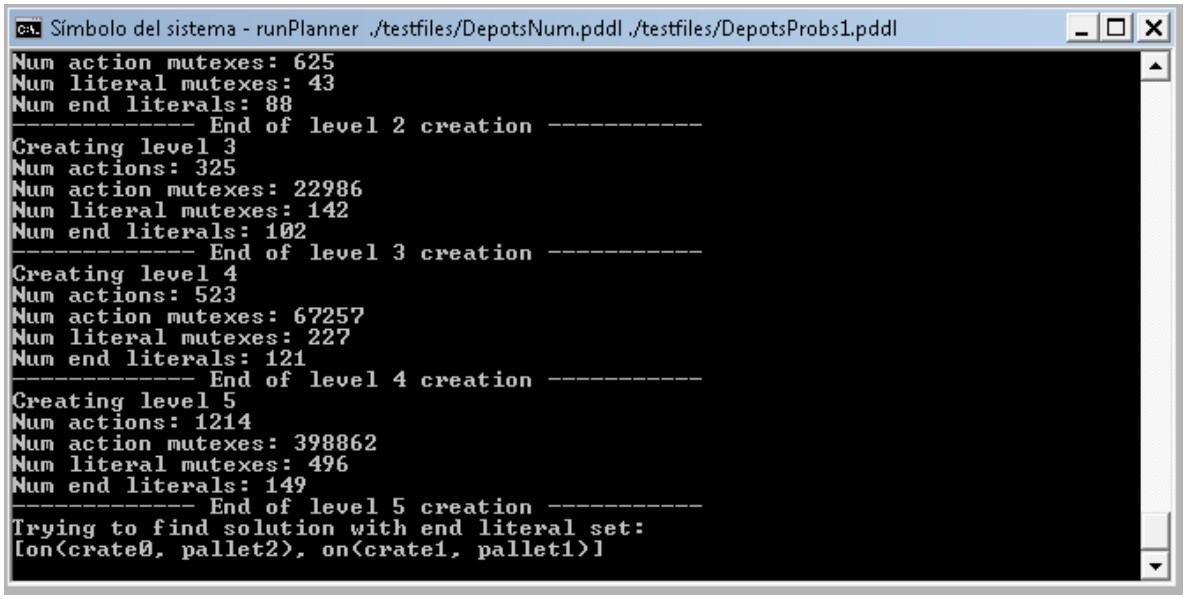
```
(:action Load
:parameters (?x - hoist ?y - crate ?z - truck ?p - place)
:precondition (and (posn ?x ?p) (posn ?z ?p) (lifting ?x ?y)
```

```
            (<= (+ (current_load ?z) (weight ?y)) (load_limit ?z)))
:effect (and (not (lifting ?x ?y)) (in ?y ?z) (available ?x)
            (increase (current_load ?z) (weight ?y))))
```

La operación descargar es justamente la inversa de la operación cargar. Se descarga la caja que está en el camión en algún lugar p donde hay un montacargas.

```
(:action Unload
:parameters (?x - hoist ?y - crate ?z - truck ?p - place)
:precondition (and (posn ?x ?p) (posn ?z ?p) (available ?x) (in ?y ?z))
:effect (and (not (in ?y ?z)) (not (available ?x)) (lifting ?x ?y)
            (decrease (current_load ?z) (weight ?y))))
)
```

No encuentra solución, pero podemos ver como la salida nos da muchos más detalles que otras implementaciones de GraphPlan.



2.6.4. Otro ejemplo clásico: mundo bloques

Introducimos el dominio de bloques por ser un ejemplo clásico. Tenemos una mesa con bloques y con un brazo robótico podemos mover los bloques colocándolos unos encima de otros o devolviéndolos a la mesa. El archivo de dominio es el siguiente:

```
(define (domain Blocks)
(:requirements :fluents)
```

El predicado *handempty* indica que el brazo robótico no está sosteniendo ningún bloque.

```
(:predicates (handempty)
```

El bloque b no tiene nada encima.

```
       (clear ?b)
```

El bloque b está encima del bloque a.

```
       (on ?a ?b)
```

El bloque b está sobre la mesa.

```
       (ontable ?b)
```

El brazo robótico está sosteniendo el bloque b.

```
            (holding ?b)
      )
```

La acción pickup consiste en que el brazo coge un bloque de la mesa. Requiere de un bloque, de que el brazo esté libre y de que el bloque b esté encima de la mesa y no tenga nada sobre él. El efecto es que el brazo queda sosteniendo el bloque, el brazo ya no está libre ni el bloque está sobre la mesa.

```
(:action pickup
 :parameters (?b)
 :precondition (and (handempty) (ontable ?b) (clear ?b))
 :effect (and (holding ?b)
              (not (handempty))
              (not (ontable ?b))
              (not (clear ?b))))
```

La acción *putdown* consiste en dejar un bloque sobre la mesa y es la operación inversa a pickup.

```
(:action putdown
 :parameters (?b)
 :precondition (holding ?b)
 :effect (and (handempty)
              (ontable ?b)
              (clear ?b)
              (not (holding ?b))))
```

La acción *stack* (apilar) consiste en colocar un bloque x que el brazo robótico está sosteniendo sobre un bloque y cuya parte superior está libre.

```
(:action stack
 :parameters (?x ?y)
 :precondition (and (holding ?x) (clear ?y))
 :effect (and (on ?x ?y)
              (handempty)
              (clear ?x)
              (not (holding ?x))
              (not (clear ?y))))
```

La acción desapilar es la simétrica a apilar.

```
(:action unstack
 :parameters (?x ?y)
 :precondition (and (clear ?x) (on ?x ?y) (handempty))
 :effect (and (clear ?y)
              (holding ?x)
              (not (clear ?x))
              (not (on ?x ?y))
              (not (handempty))))
```

```
)
```

Con estas simples operaciones, podemos resolver problemas como éste:

```
(define (problem blocks1)
    (:domain Blocks)
    (:objects a b c)
```

Tenemos un bloque a sobre la mesa, el bloque b sobre el a, y el bloque c sobre el b, y el brazo robótico libre.

```
    (:init (ontable a)
        (on b a)
           (on c b)
        (clear c)
        (handempty)
    )
```

El objetivo es tener el bloque b sobre la mesa, el bloque a sobre el b y bloque c sobre el a.

```
    (:goal (and (ontable b) (on a b) (on c a)))
```

La métrica a optimizar es que se minimice el tiempo.

```
    (:metric minimize total-time)
)
```

La solución que da GraphPlan es la siguiente:

```
unstack(c,b)
putdown(c)
unstack(b,a)
putdown(b)
pickup(a)
stack(a,b)
pickup(c)
stack(c,a)
```

Pone en la mesa el bloque c, pone en la mesa el bloque b, coge el bloque a, lo pone encima de a y luego pone el bloque c encima de a.

2.6.5. Un ejemplo con FORALL

En (8) se puede encontrar un ejemplo de FORALL. En un dominio de camiones tenemos la acción drive-truck(conducir camión) que nos permite llevar un camión de un sitio loc-from a otro loc-to dentro de la misma ciudad.

```
(:action drive-truck
:parameters (?truck - truck ?loc-from ?loc-to - location
?city - city)
:precondition
(and (at ?truck ?loc-from)
(in-city ?loc-from ?city)
(in-city ?loc-to ?city))
```

El efecto es que el camión deja de estar en el sitio de partida y pasa de estar en el sitio de llegada.

```
:effect (and (at ?truck ?loc-to)
(not (at ?truck ?loc-from))
```

Y aquí viene la parte interesante. Un camión va a estar lleno de objetos y todos esos objetos tienen que moverse con el camión. Así que para todos los objetos del dominio (aquí está la utilidad del forall) que cumplan la condición de que están dentro del camión, también hay que moverlos del sitio de salida al sitio de llegada.

```
(forall (?x - obj)
(when (and (in ?x ?truck))
(and (not (at ?x ?loc-from))
(at ?x ?loc-to))))))
```

En (13) se describe otro ejemplo clásico de uso de forall con el mismo objetivo que el anterior. En el mundo maletín tenemos un maletín que podemos mover de un lugar m a un lugar l. La constante B indica el maletín y tiene que estar en m y no debe tener que ser movido al mismo sitio, por eso el sitio de partida m no puede ser igual que el sitio de llegada l. El efecto lógico es que el objeto deje de estar en m y esté en el pero, como en el caso anterior, para todos los objetos z que estén dentro del maletín y no sean el propio maletín, los quitamos del lugar de partida m y los colocamos en el lugar de llegada l.

```
(:action mov-b
     :parameters (?m ?l –location)
     :precondition (and (at B ?m) (not = ?m ?l)))
     :effect (and (at b ?l) (not (at B ?m))
          forall (?z)
               (when (and (in ?z) (not (= ?z B)))
                    (and (at ?z ?l) (not (at ?z ?m))))))
```

2.6.6. Un ejemplo con EXISTS

El operador EXISTS indica si existe algún objeto que cumpla algún predicado. Por ejemplo, tenemos un tipo zapatos.

```
(:types Shoes)
```

Y un predicado que indica que una cosa es de tipo zapato y de color verde.

```
(:predicates(Green ?things – Shoes))
```

Pues bien, podemos especificar como condición para llevar a cabo una acción que exista un zapato y que, además, sea de color verde.

```
(exists(?things – Shoes)(Green Shoes ?things))
```

2.6.7. Un ejemplo con todo

En (14) se muestra un ejemplo con todo junto con un dominio de neveras.

```
(define (domain fridge-domain-rich)
  (:requirements :adl)
```

Tenemos 4 tipos de objetos: tornillos, la placa, el compresor y la nevera.

```
(:types screw backplane compressor fridge)
```
Screwed indica si un tornillo está perfectamente atornillado.
```
(:predicates (screwed ?s)
```
Holds indica si un tornillo está acoplado, es decir, está sosteniendo a la placa.
```
(holds ?s ?b)
```
In-place indica si la placa está en su sitio.
```
(in-place ?b)
```
Part-of indica si una placa es parte de una nevera.
```
(part-of ?b ?f)
```
Fridge-on indica si la nevera está encendida.
```
(fridge-on ?f)
```
Convers indica si un objeto b, que será la placa, cubre al compresor x.
```
(covers ?b – object ?x - compressor)
```
Attached indica si el compressor está puesto.
```
(attached ?x - compressor)
```
Ok indica si el objeto funciona.
```
(ok ?c))
```

La acción unfasten (desatornillar) quita un tornillo de una placa.
```
(:action unfasten
        :parameters (?x - screw ?y - backplane)
```
El tornillo x s ha de estar apretado y estar sobre la placa y.
```
        :precondition (and (screwed ?X) (holds ?x ?y) )
```
El efecto es que el tornillo ya no está apretado.
```
        :effect (not (screwed ?X)))
```

La acción fasten aprieta el tornillo.
```
(:action fasten
        :parameters (?x - screw ?y - backplane)
```
El tornillo debe estar puesto en la placa pero no apretado.
```
        :precondition (and (not (screwed ?X)) (holds ?x ?y))
```
Y el efecto es que el tornillo queda apretado.
```
        :effect (screwed ?X))
```
La acción remove-backplane quita la placa de la nevera.
```
(:action remove-backplane
        :parameters (?x - backplane ?f - fridge)
```
La placa debe estar colocada y ser una parte de la nevera.
```
        :precondition (and (in-place ?x) (part-of ?x ?f)
```
La nevera no debe estar encendida.
```
                    (not (fridge-on ?f))
```
Todos los tornillos
```
                    (forall (?s - screw)
```
que estén enchufados a la placa x y no estén atornillados. Aquí vemos la función del imply : fuerza una condición que tienen que cumplir los objetos del forall.

```
                    (imply (holds ?s ?x)
                        (not (screwed ?s)))))
```

El efecto es que la placa ya no está en su lugar, detrás de la nevera.

```
        :effect (not (in-place ?X)))
```

La acción attach-backplane fija la placa a la parte posterior de la nevera.

```
(:action attach-backplane
        :parameters (?x - backplane ?f - fridge)
```

La placa no debe estar colocada, debe ser una parte de la nevera, la nevera no debe estar encendida y cogemos todos los tornillos que pertenecen a la placa y no están atornillados.

```
        :precondition (and (not (in-place ?x))
                  (part-of ?x ?f) (not (fridge-on ?f))
                  (forall (?s - screw)
                     (imply (holds ?s ?x)
                         (not (screwed ?s)))))
```

El efecto es que la placa queda fijada a la nevera.

```
        :effect (in-place ?X))
```

La acción start-fridge enciende la nevera.

```
(:action start-fridge
        :parameters (?f - fridge)
```

Para que podamos encender la nevera, debe existir una placa que esté en su lugar y sea parte de la nevera. Además, todos los tornillos que pertenezcan a esta nevera han de estar apretados. Y la nevera, por supuesto, no debe estar encendida.

```
        :precondition (exists (?x - backplane)
                   (and (in-place ?x)
                     (part-of ?x ?f)
                     (forall (?s - screw)
                        (imply (holds ?s ?x)
                            (screwed ?s)))
                     (not (fridge-on ?f))))
```

Y el efecto es que la nevera se enciende.

```
        :effect (fridge-on ?f))
```

La acción stop-fridge para la nevera.

```
(:action stop-fridge
        :parameters (?f - fridge)
```

La nevera ha de estar encendida.

```
        :precondition (fridge-on ?f)
        :effect
```

Y el efecto es que dejará de estar encendida.

```
        (not (fridge-on ?f)))
```

La acción change.compressor cambia el compresor.

```
(:action change-compressor
```

El parámetro es el compresor nuevo a colocar.

```
        :parameters (?y - compressor)
```

vars indica variables auxiliares, aquí x es el compresor antiguo que vamos a cambiar.

```
:vars (?x - compressor)
```

El compresor x a cambiar ha de estar puesto en la nevera y el nuevo y que vamos a poner no ha de estar puesto.

```
:precondition (and (attached ?x) (not (attached ?y))
```

Para todas las placas que cubran al compresor que queremos cambiar y que no estén puestas.

```
(forall (?a - backplane)
   (imply (covers ?a ?x)
       (not (in-place ?a)))))
```

El efecto es que el compresor x ya no está en la nevera y el y sí.

```
:effect (and (not (attached ?X)) (attached ?y)
```

Para todas las placas

```
(forall (?a - backplane)
```

Si la placa a cubre al compresor x (con esto se selecciona la única placa que cubre al compresor x)

```
(when (covers ?a ?x)
```

La placa ya no cubre al compresor x, sino que cubre al compresor y

```
(and (not (covers ?a ?x))
  (covers ?a ?y)))))))
```

Y he aquí el problema:

```
(define (problem rich-fixb)
   (:domain fridge-domain-rich)
   (:situation fixb)
   (:goal (and (attached c2) (ok c2) (fridge-on f1))))
```

La situación fixb se refiere a un problema definido anteriormente del que tomamos la situación inicial.

```
(define (problem fixb-strips)
    (:domain fridge-domain-strips)
```

Tenemos cuatro tornillos, una placa, dos compresores y una nevera. El objetivo es quitar el compresor c1 y colocar el c2 en funcionamiento, dejando la nevera encendida al final.

```
  (:objects s1 s2 s3 s4 b1 c1 c2 f1)
  (:init (screw s1)
      (screw s2) (screw s3) (screw s4)
      (backplane b1)
      (compressor c1) (compressor c2) (fridge f1)
```

La placa b1 cubre al compresor c1, el compresor forma parte de la nevera, o sea, está dentro.

```
      (covers b1 c1) (part-of b1 f1)
```

Los cuatro tornillos están sujetando la placa.

```
(holds s1 b1)  (holds s2 b1)  (holds s3 b1)
(holds s4 b1)
```

Los compresores funcionan y la nevera está encendida.

```
(ok c1) (ok c2) (fridge-on f1)
```

Los cuatro tornillos están atornillados.

```
(screwed s1) (screwed s2) (screwed s3) (screwed s4)
```

La placa está en su lugar y el compresor también.

```
(in-place b1) (attached c1))
```

El objetivo es colocar el compresor 2 funcionando y la nevera encendida.

```
(:goal (AND (attached c2) (ok c2) (fridge-on f1)))
(:length (:serial 13 ) (:parallel 6))
  )
```

3. DESARROLLO DE LA APLICACIÓN

Una vez fijado el marco teórico formado por los lenguajes de planificación STRIPS y PDDL y tras habernos dado un buen chapuzón en el funcionamiento del algoritmo GraphPlan, que es el que nos va a permitir encontrar soluciones de planificación a los problemas planteados, podemos empezar a estudiar cómo vamos a realizar la aplicación.

Se decide usar un enfoque de modelado para el desarrollo rápido de aplicaciones (RAD) que nos va a permitir realizar el programa con velocidad, usando los componentes que ya tenemos, el compilador de PDDL y unas implementaciones del algoritmo Graphplan. El objetivo es eminentemente pedagógico: dotar al alumnado de una herramienta que les permita escribir dominio en PDDL y comprobar como el algortimo GraphPlan va dando pasos hasta encontrar una solución de planificación al problema planteado.

Además, usaremos UML el lenguaje de modelado de sistemas de software más extendido sobre la herramienta CASE Visual Paradigm para diseñar los diagramas necesarios.

3.1. RAD (Rapid Application Develompent)

En su libro Rapid Application Development **(15)**, James Martin describe que el desarrollo rápido de aplicaciones se basa en tres ideas principales:

- Desarrollo iterativo: se van creando versiones que se mejoran en cada paso.
- Construcción de prototipos: se empiezan creando prototipos sobre los que el usuario puede interactuar y dar ideas sobre el futuro funcionamiento de la aplicación.
- Uso intensivo de herramientas CASE (Computer Aided Software Engineering) que ayuden a acelerar el desarrollo.

Este método de desarrollo de aplicaciones tiene múltiples ventajas:

- Se hace un énfasis especial en la simplicidad y la usabilidad de la interfaz gráfica de usuario.
- Aumenta la velocidad de desarrollo a través del uso de métodos como el prototipado rápido, el uso de herramientas CASE, reusabilidad de código, etc…
- Se reduce la complejidad limitando la funcionalidad permitida al usuario final.

Y algunos inconvenientes:

- Las aplicaciones son menos escalables.
- Se pueden descartar características útiles hasta próximas versiones po las limitaciones de tiempo impuestas por el método.

RAD es una idea que defiende que las aplicaciones pueden ser desarrolladas más rápido y con más calidad a través de ciertas metodologías que incluyen:

- Análisis de requisitos mediante talleres y muestreos entre los posibles usuarios.
- Prototipado y pruebas de diseño repetidas desde el principio.

- Reuso de componentes de software.
- Una planificación rígida que deja las mejoras en el diseño para siguientes versiones.
- Formalidad menor en la comunicación entre personas y grupos.

Figura 3.1. Pasos en el desarrollo tradicional

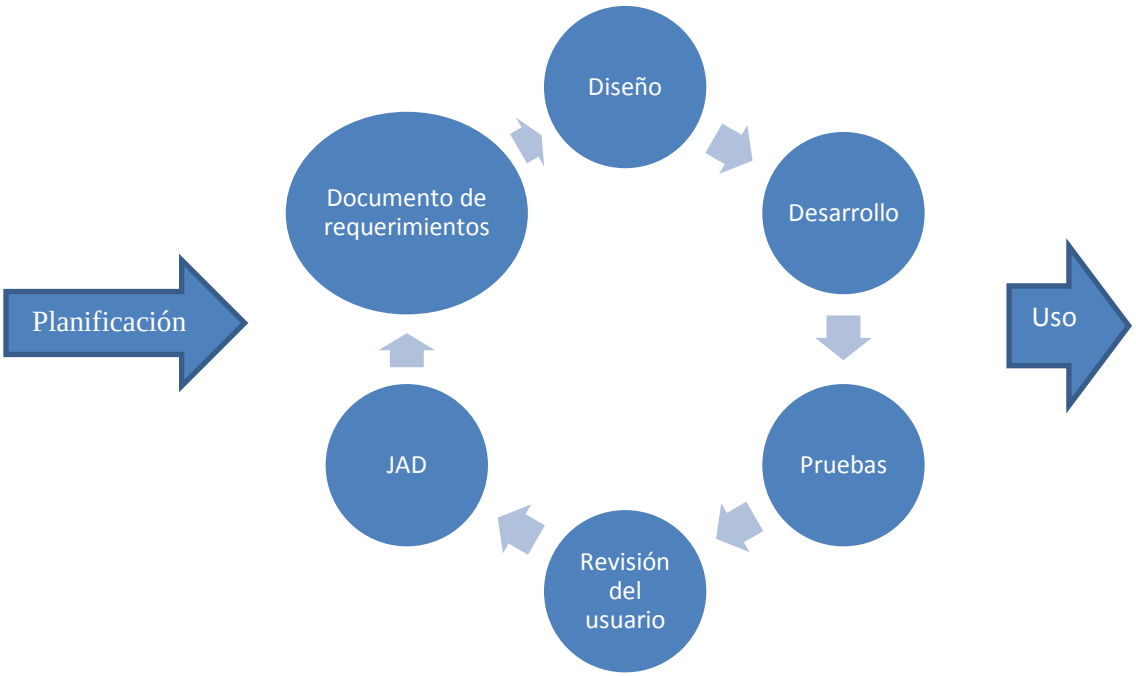


Figura 3.2. Pasos en el desarrollo RAD

JAD significa Joint Application Design (diseño de aplicaciones unido). Es una reunión donde se unen los futuros usuarios y los diseñadores y programadores en un taller centrado en la aplicación. Esto acelera el desarrollo porque una de las partes en las que se pierde más tiempo es en la comunicación entre usuarios y desarrolladores.

3.2. RAD y UML: fases

Vamos a seguir las siguientes fases en el desarrollo:

1. Recolección de requerimientos
   1.1. Proceso de negocio: UML Diagrama de actividades
   1.2. Análisis de dominio: UML diagrama de clases de alto nivel.
   1.3. Identificación cooperativa del sistema: éste depende del PDDL y de Graphplan.
   1.4. Requerimientos del sistema: lista de requerimientos y UML refinar el diagrama de clases, diagrama de paquetes.
2. Análisis
   2.1. Entendimiento del uso del sistema: UML Diagrama de Casos de Uso.
   2.2. Secuencia de los casos de uso: analizar su secuencia y producir una descripción textual.
   2.3. Refinar los diagramas de clases: asociaciones, describir clases, multiplicidades, generalizaciones, agregados -> Diagrama de clases refinado.
   2.4. Analizar cambios en el estado de los objetos: UML Diagrama de estado
   2.5. Definir las interacciones entre objetos: UML Diagramas de secuencia y colaboración
   2.6. Análisis de integración con sistemas de cooperación: NO
3. Diseño
   3.1. Desarrollar y refinar el diagrama de objetos: tomar el diagrama de clases y generar todos los diagramas de objetos examinando cada operación y desarrollando un diagrama de actividades correspondiente. UML Diagrama de actividades
   3.2. Diagramas de componentes: UML Diagrama de componentes
   3.3. Plan de liberación: diagrama de liberación
   3.4. Diseño y prototipo de interfaces de usuario: analizar los casos de uso de UML
   3.5. Documentación: diagramas UML del diseño
4. Desarrollo
   4.1. Construir el código
   4.2. Probar el código
   4.3. Construir interface de usuario, conectar al código, probar.
   4.4. Documentación del sistema.
5. Uso
   5.1. Instalación
   5.2. Pruebas

4. RECOLECCIÓN DE REQUERIMIENTOS

4.1. Proceso de negocio

En esta fase desgranamos algunas de las actividades más comunes usando diagramas de actividades de UML. Aquí glosamos algunas, dejando en el archivo de Visual Paradigm el grueso de los diagramas de actividades realizados.

Figura 4.1. Edición de documentos

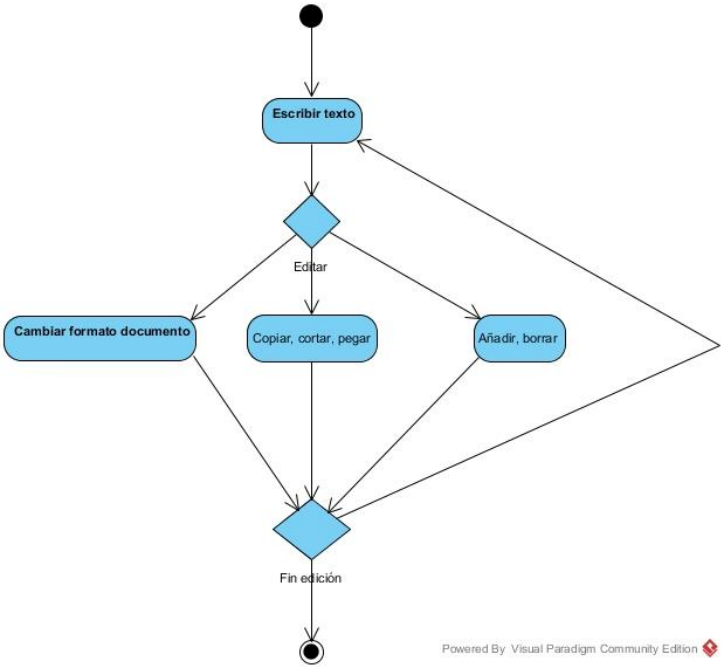


Figura 4.2. Editar archivo

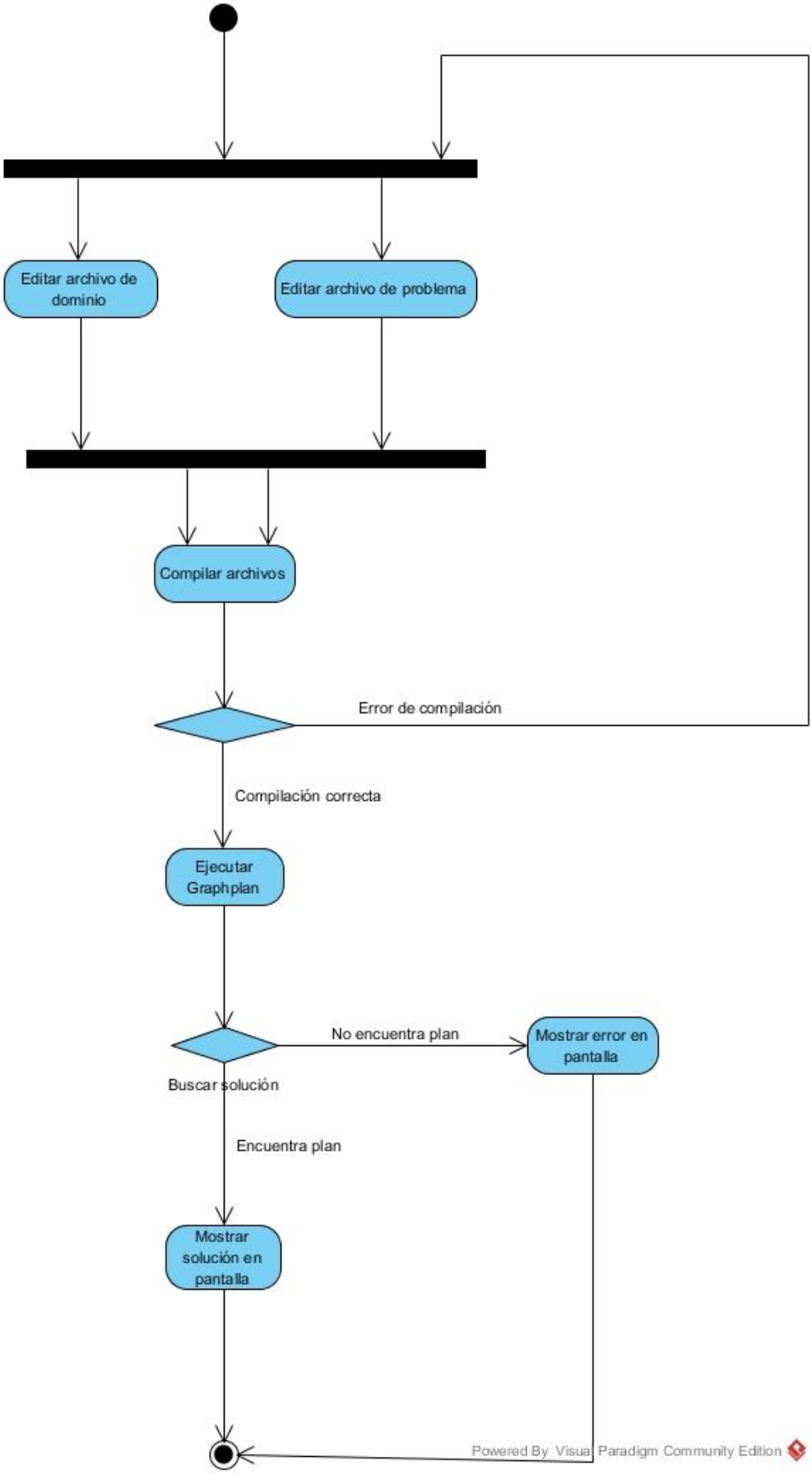


Figura 4.3. Ejecutar plan

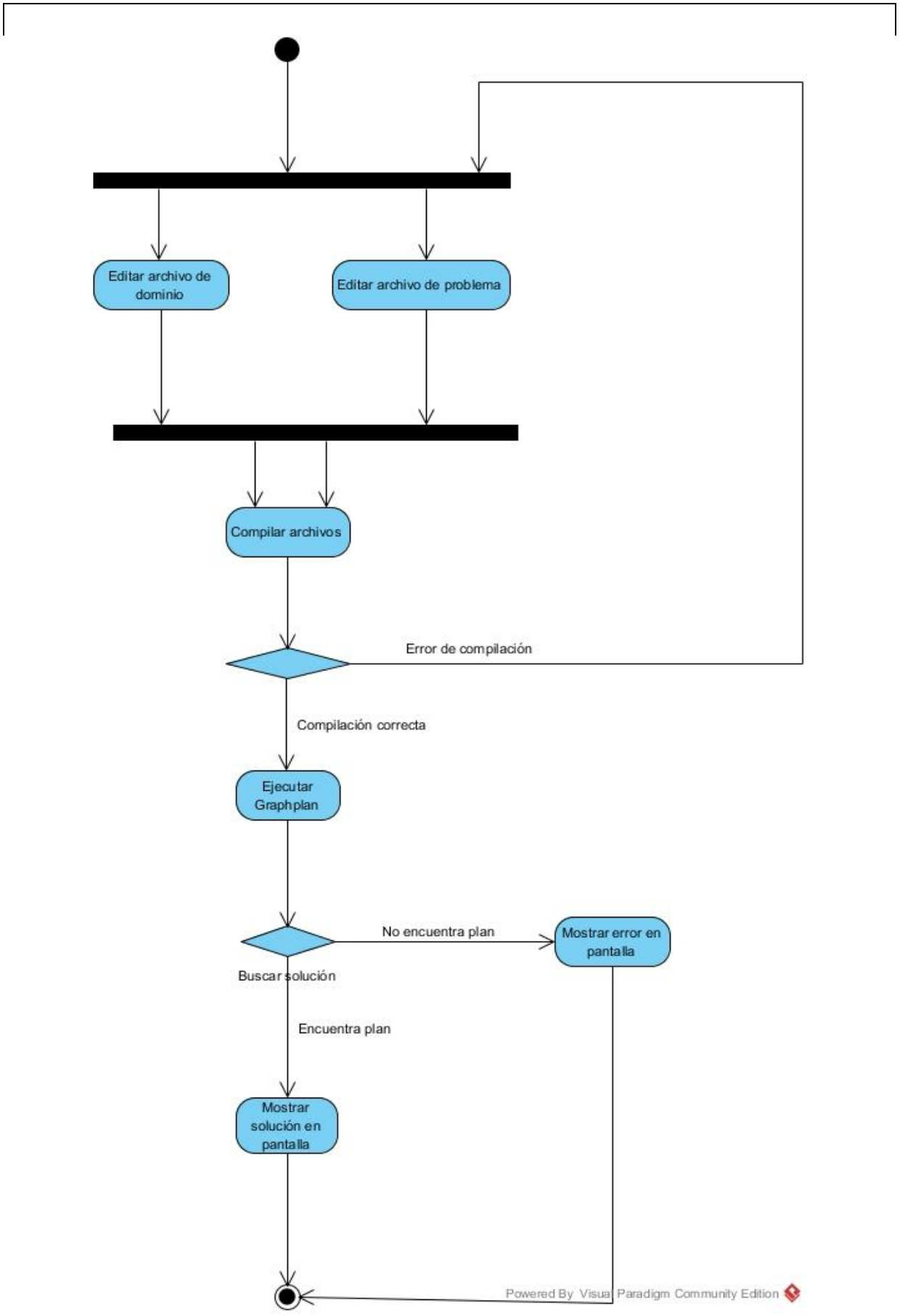


Figura 4.4. Ejecutar plan

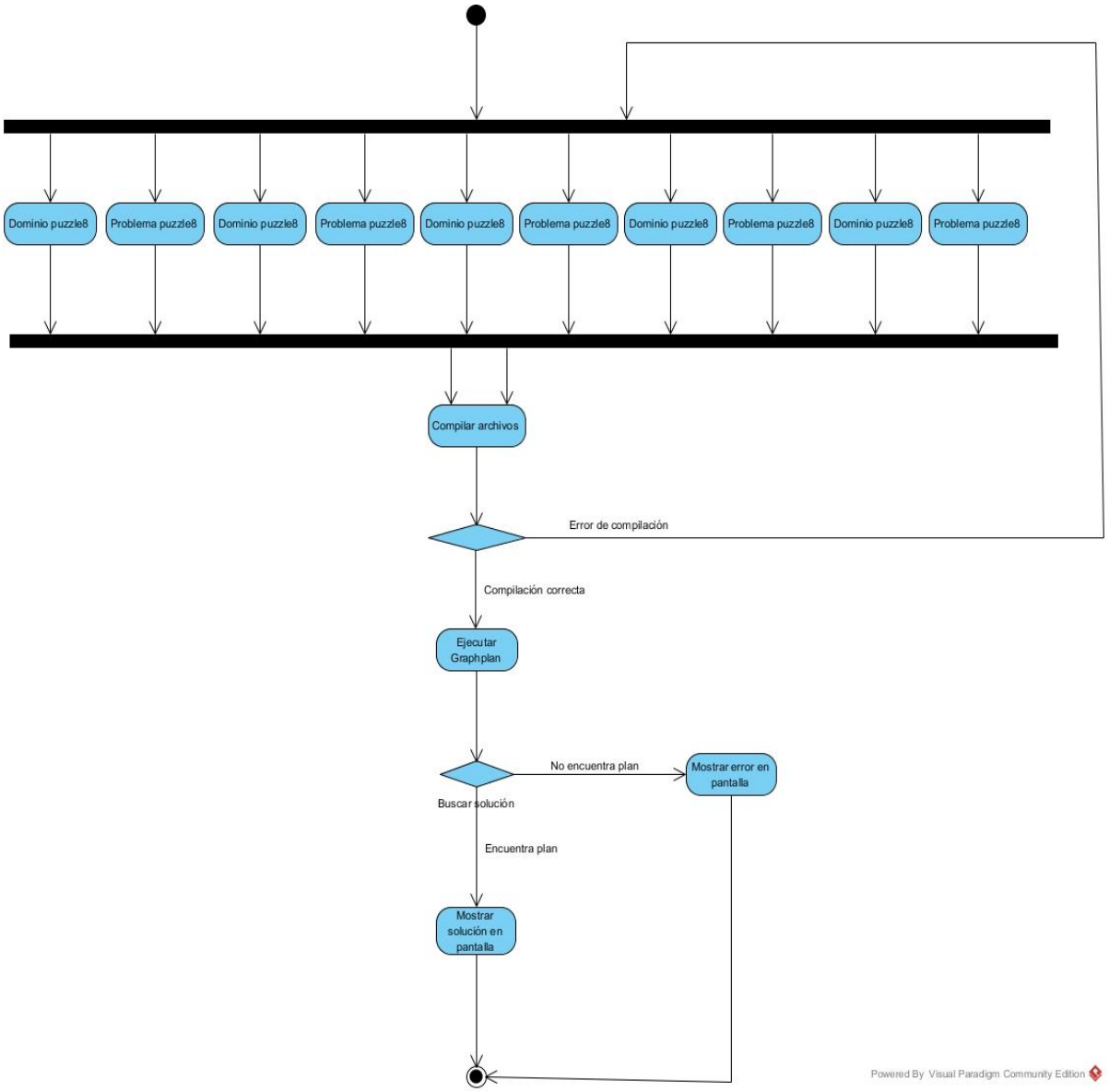


Figura 4.5. Diagrama de actividad: ejemplos

4.2. Análisis de dominio

El resultado del análisis de dominio es un diagrama de clases de alto nivel. A alto nivel, tenemos un mundo formado por dos clases, la clase dominio y la clase problema.

Un diagrama de clases nos va a dar una visión del sistema mediante la descripción de las clases que lo componen y las relaciones que hay entre las mismas. De este modo, sabemos con qué tipo de objetos estamos tratando, qué propiedades tienen y qué operaciones pueden realizar estos objetos, así como las relaciones que se pueden establecer entre ellos.

En un Mundo hemos de tener un dominio y un problema y la única operación pública que vamos a realizar sobre el mundo es GraphPlan, que nos da una solución de planificación al problema.

Un dominio y un problema tienen ambos un nombre y un texto que los describe. En ambos necesitamos operadores que los compilen para comprobar su corrección y se puedan pasar a GraphPlan.

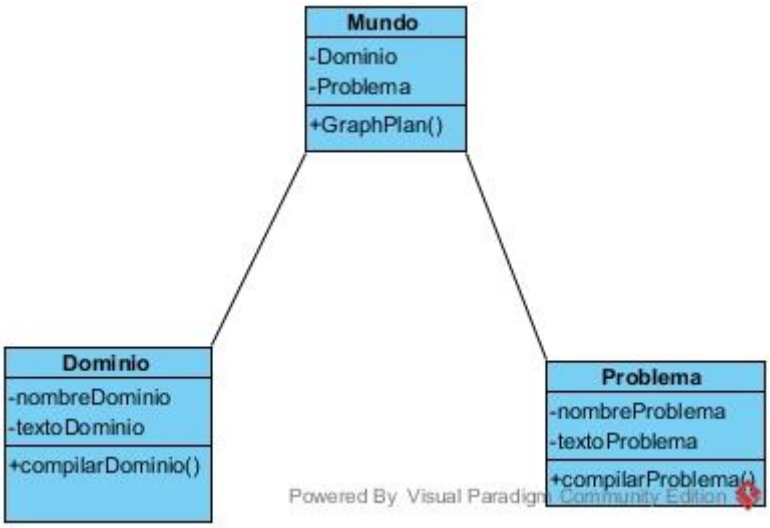


Figura 4.6. Diagrama de clases de alto nivel

4.3. Requerimientos del sistema

Los requerimientos coinciden con los objetivos que nos hemos planteado al principio, en resumen:

- Un editor de texto multidocumento.
- Ventana de salida donde se vea la salida del parser conforme se va produciendo.
- Ejecución paso a paso.
- Salida mínima con solo la solución de la planificación.
- Salida máxima con todos los pasos dados uno por uno y el tiempo que se tarda.
- Mundos precargados.
- Tutoriales en vídeo.

Por último, definimos el diagrama de paquetes. Un diagrama de paquetes muestra la organización de los distintos módulos y subsistemas que forman el sistema completo. Así, se muestra la estructura y las relaciones entre estos subsistemas o módulos. jPlague hace uso de JAVAGP y de PDDL4J, que, a su vez, contiene una implementación de GraphPlan.

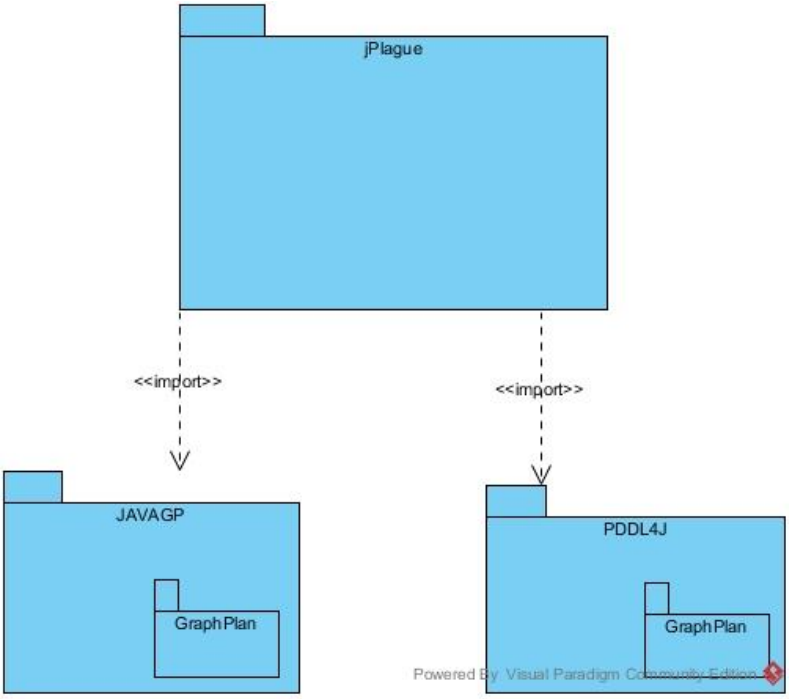


Figura 4.7. Diagrama de paquetes inicial

5. ANÁLISIS

5.1. Entendimiento del uso del sistema: Diagramas de Casos de Uso

Los diagramas de casos de uso (16) describen el comportamiento del sistema desde un punto de vista externo, es decir, se describe cómo un usuario, otro sistema, una parte del hardware o una señal de tiempo pone en marcha el sistema, la secuencia de pasos que realiza el sistema y el resultado del mismo. A las entidades que inician secuencias de pasos se les llama actores y cada secuencia de eventos se llama escenario. Así, en nuestro caso tenemos el caso de uso Resolver problema, donde un usuario, el actor, introduce un mundo y un dominio, llama al algoritmo GraphPlan y éste le devuelve una planificación o error si no ha podido encontrarla. Podemos describir algunos casos de uso básicos:

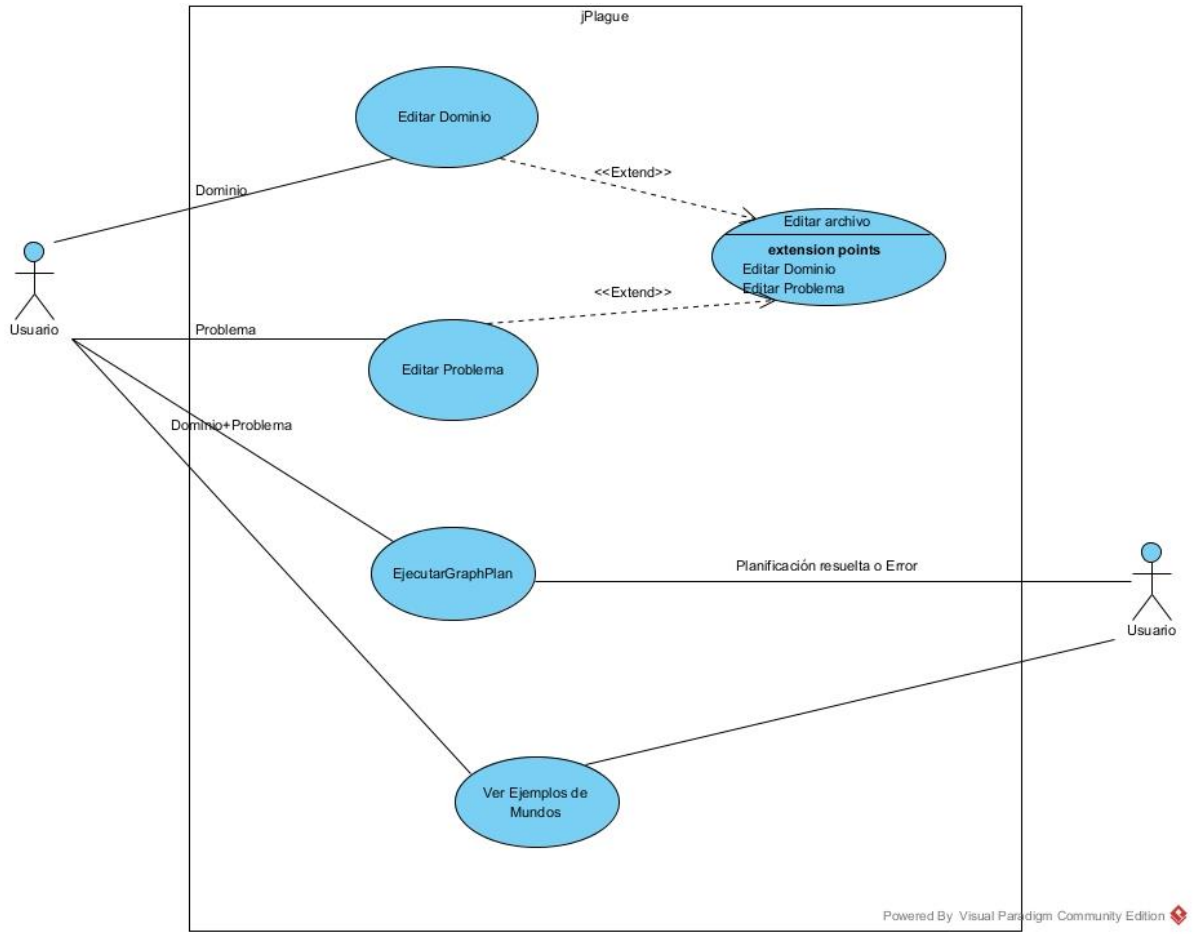


Figura 5.1. Diagrama de casos de uso general

5.2. Secuencia de los casos de uso

CU1: Editar Dominio

Este caso de uso extiende el caso de uso Editar archivo, con la única salvedad de que se le puede poner una extensión como DOM, aunque la extensión más usada es PDDL.

CU2: Editar Problema

Este caso de uso extiende el caso de uso Editar archivo, con la única salvedad de que se le puede poner una extensión como PRO, aunque la extensión más usada es PDDL.

CU3: Editar archivo

Este caso de uso se puede extender gráficamente del siguiente modo.

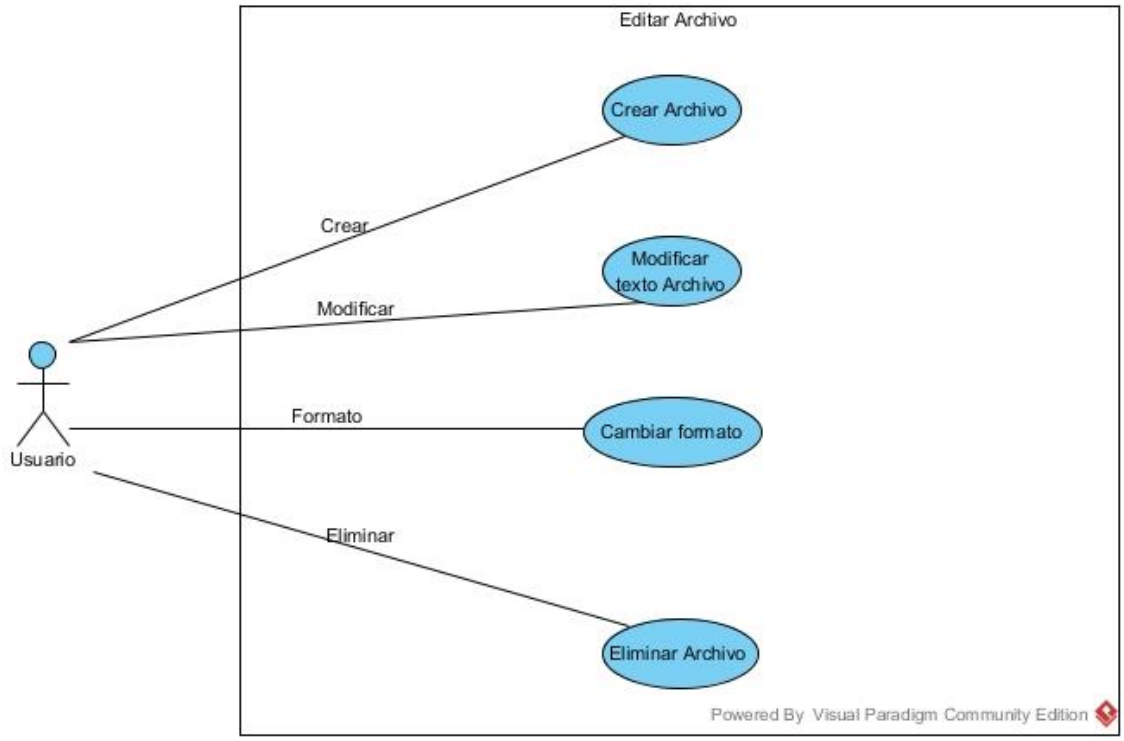


Figura 5.2. Caso de uso Editar Archivo

CU3: Ejecutar GraphPlan

Éste es el caso de uso principal de esta aplicación. Este caso de uso toma un archivo de dominio y un archivo de problema, los manda al paquete elegido, ya sea el GraphPlan de PDDL4J o JAVAGP y nos devuelve la salida.

1. Editar archivo de dominio.
2. Editar archivo de problema.
3. Ejecutar GraphPlan enviando una línea de comandos con los dos archivos y las opciones seleccionadas al compilador elegido: javagp o PDDL4J.
4. Obtener la salida.
5. Formatearla convenientemente.
6. Mostrarla en pantalla.

5.3. Refinar los diagramas de clases

En este diagrama de clases se pone de manifiesto que un mundo está formado por un dominio pero pueden plantearse varios problemas para ese dominio.

En este diagrama más detallado especificamos qué es un dominio, que es un nombre más un texto en STRIPS o PDDL y qué es un problema, también un texto en STRIPS o PDDL.

Así que nuestra clase jPlague tendrá, al menos, un dominio y un problema, y un operador resolverPlanificación que hará uso de las clases JAVAGP y PDDL4J para obtener la respuesta y sacarla por pantalla o impresora.

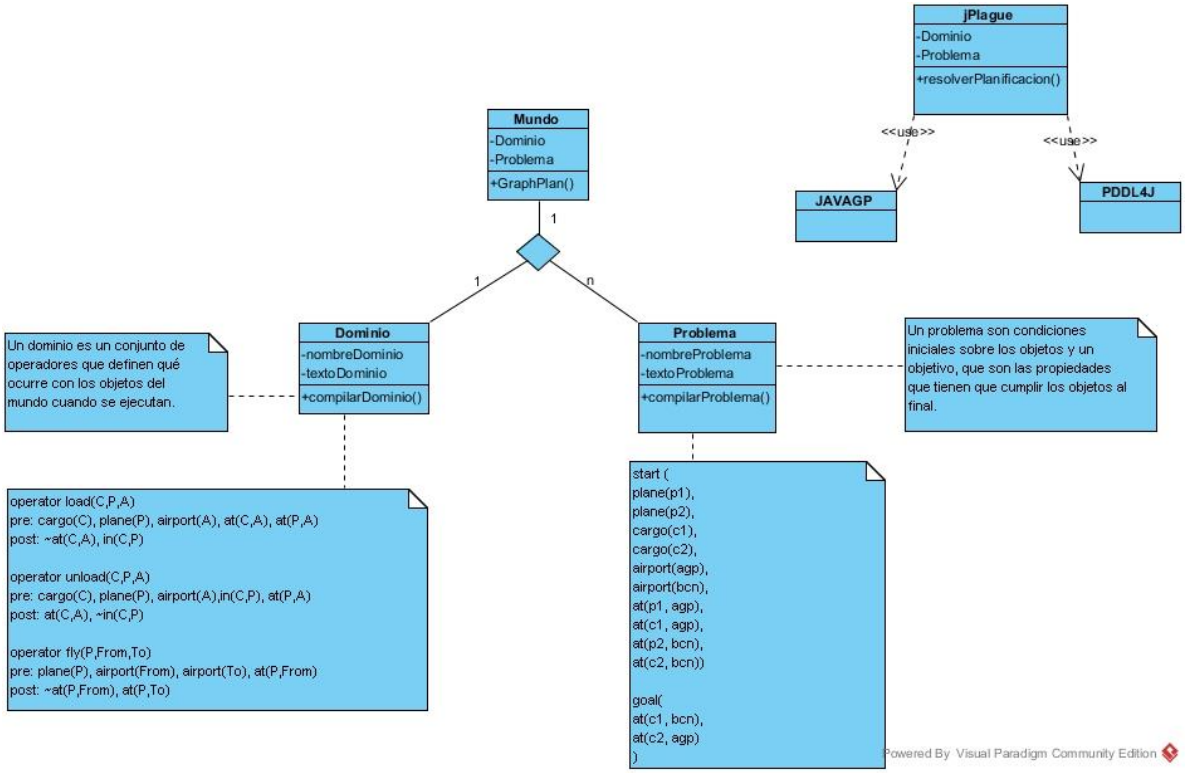


Figura 5.3. Diagrama de clases refinado

5.4. Analizar cambios en el estado de los objetos

Un diagrama de estados nos permite representar gráficamente como los objetos que forman un sistema van cambiando de estado conforme se van produciendo sucesos y el tiempo va transcurriendo. En este caso hemos realizado un diagrama de estados general de la aplicación: al principio se van a editar el dominio y el problema, luego se va a llamar a la aplicación GraphPlan, que nos va a dar como resultado una planificación o error.

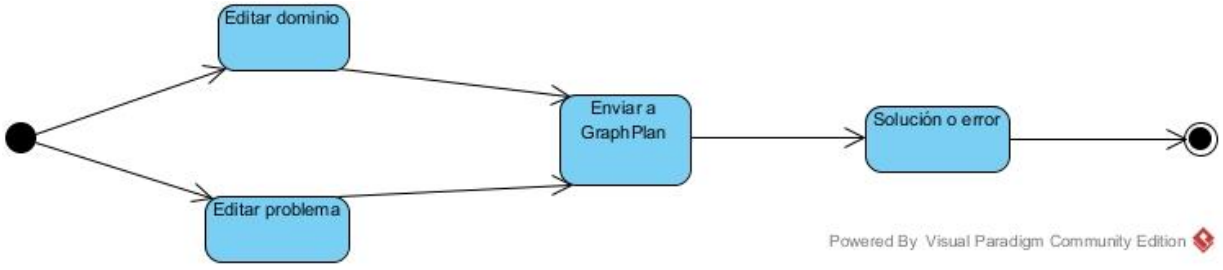


Figura 5.4. Diagrama de estados básico

5.5. Definir las interacciones entre objetos

En UML hay dos tipos de diagrama que nos permiten expresar las interacciones entre objetos, los diagramas de secuencia y los diagramas de colaboración.

En un **diagrama de secuencia** se muestran rectángulos con nombre que representan objetos, líneas continuas con una punta de flecha que representan mensajes y el tiempo, que avanza en progresión vertical.

Los objetos se colocan en la zona superior de izquierda a derecha de forma que el diagrama sea más entendible. Hacia abajo se representa la línea de vida de cada objeto, que comienza con su activación, un rectángulo hacia abajo que representa la duración de la vida del objeto desde que se activa hasta que desaparece.

Los objetos se pueden enviar mensajes entre ellos, que pueden ser simples, donde se transfiere el control de un objeto a otro, síncrono, donde un objeto espera la respuesta a su mensaje para continuar o, asíncrono, cuando no se espera antes de continuar.

Un diagrama de secuencia simple para el funcionamiento general de jPlague podría ser el siguiente:

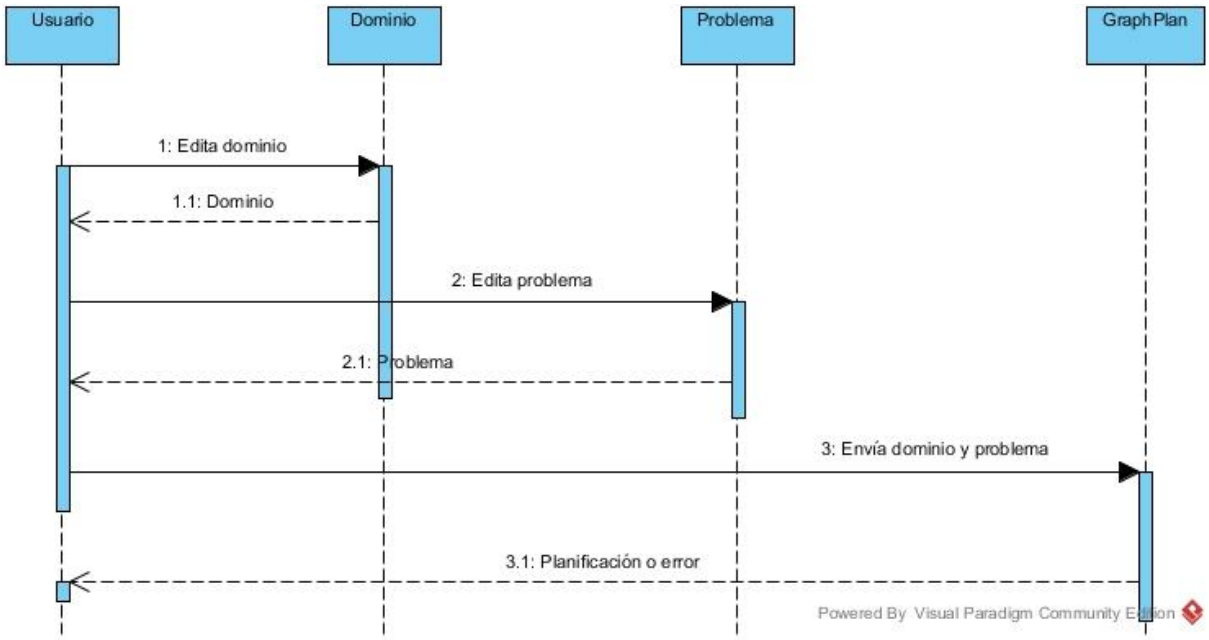


Figura 5.5. Diagrama de secuencia general

Un **diagrama de colaboración** es una extensión de un diagrama de objetos. Se muestran los objetos, así como los mensajes que los objetos se remiten entre sí. Los mensajes se representan como flechas que apuntan al objeto receptor: normalmente, los objetos envían mensajes a otros objetos para que estos últimos ejecuten cierta acción. Cada mensaje tiene los parámetros que un objeto envía a otro entre paréntesis.

Todo diagrama de colaboración se puede expresar como un diagrama de secuencia. De hecho, la aplicación que estamos usando, Visual Paradigm, lo hace automáticamente, dando el siguiente resultado:

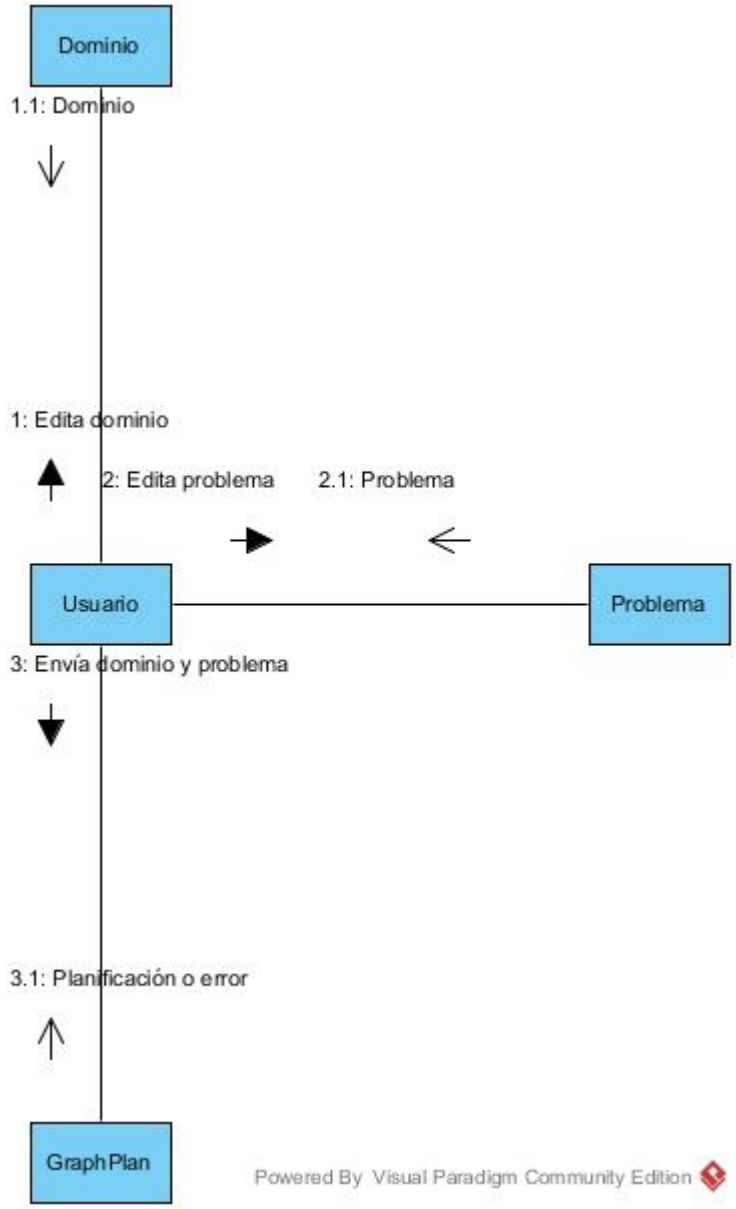


Figura 5.6. Diagrama de colaboración a partir del de secuencia

5.6. Análisis de integración con sistemas de cooperación

jPlague hace uso de JAVAGP y PDDL4J para compilar el archivo de dominio y el problema. Se han hecho modificaciones en el código de ambos proyectos para adecuar la salida. Por un lado, se han traducido los resultados de la salida, y, por otro, se ha añadido la opción de que el usuario pueda ir comprobando paso a paso como el algoritmo GraphPlan va desarrollando los niveles hasta llegar a una planificación correcta o devuelva error.

6. DISEÑO

6.1. Desarrollar y refinar el diagrama de objetos:

En el diagrama de objetos se toma el diagrama de clases y se generan todos los diagramas de objetos examinando cada operación y desarrollando un diagrama de actividades correspondiente.

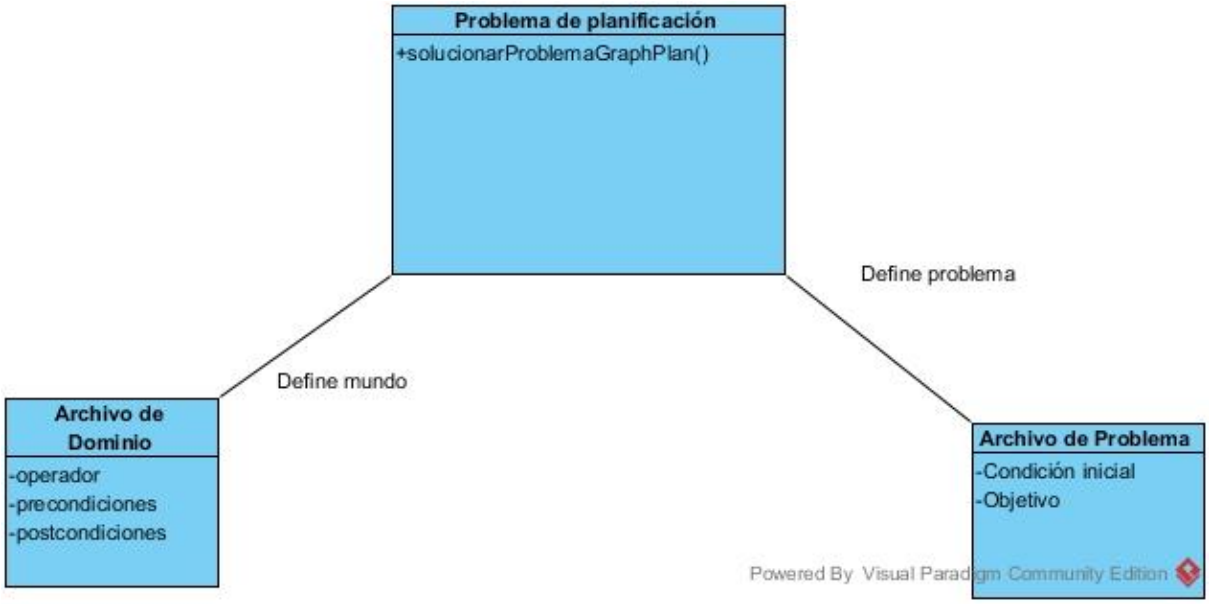


Figura 6.1. Diagrama de objetos

6.2. Diagrama de componentes

En UML, un diagrama de componentes relaciona componentes de software como tablas, archivos de datos, ejecutables, bibliotecas, archivos… Es importante destacar que un componente puede implementar más de una clase. En nuestro caso, JAVAGP y PDDL4J contienen una serie de clases que colaboran para compilar archivos STRIP y PDDL y generar planificaciones con GRAPHPLAN.

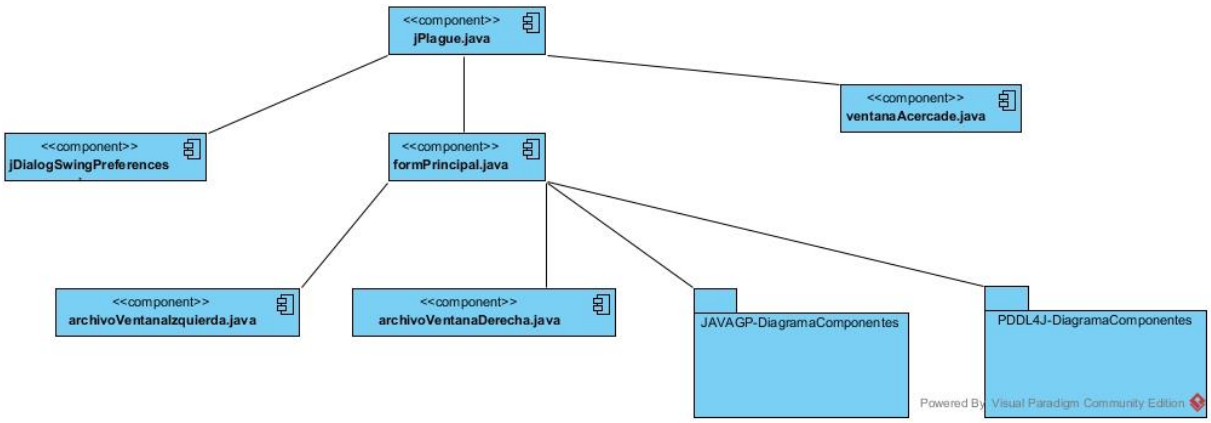


Figura 6.2. Diagrama de componentes general

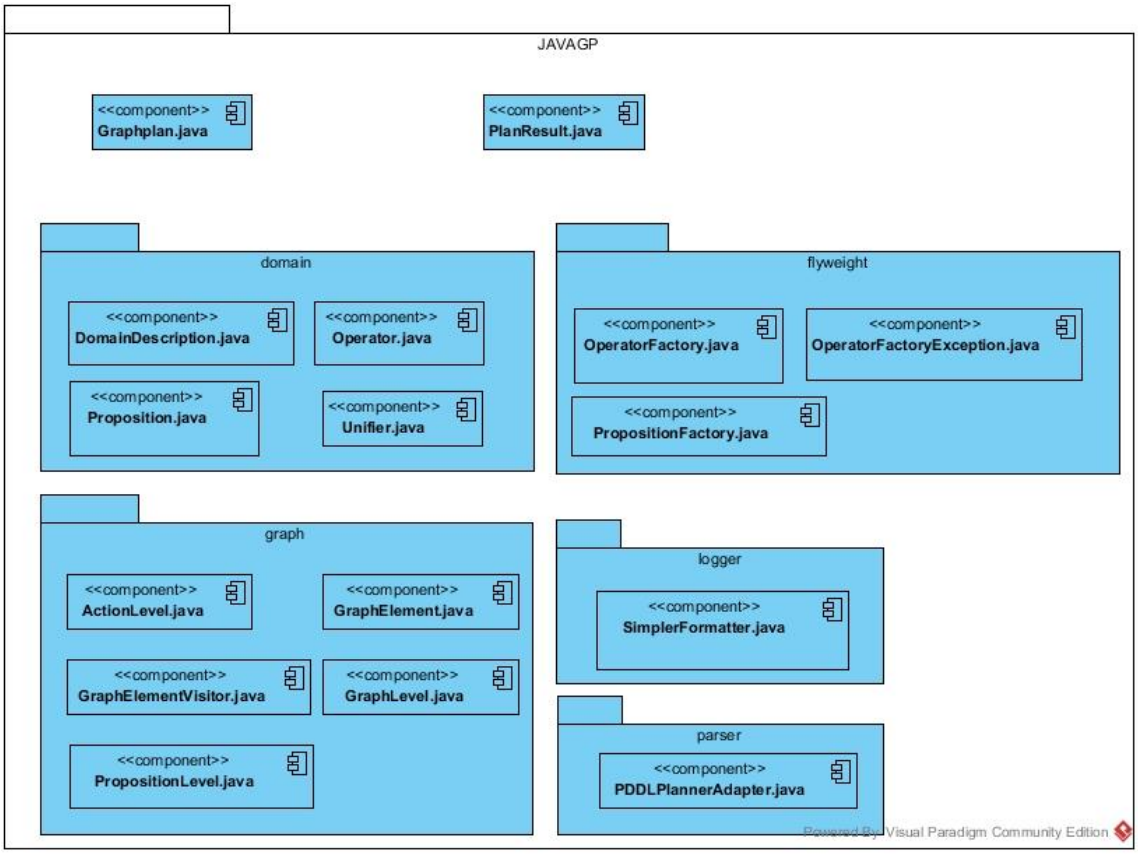


Figura 6.3. Diagrama de componentes JAVAGP

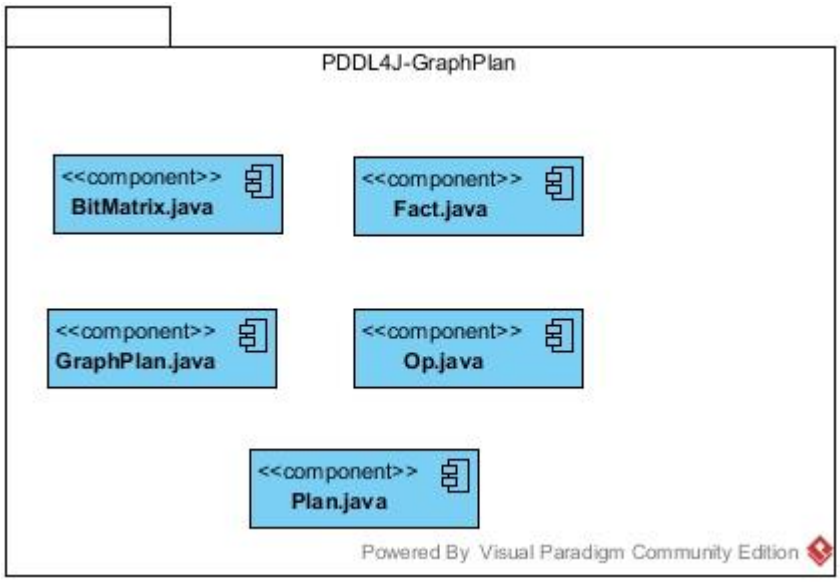


Figura 6.4. Diagrama de componentes PDDL4J

6.3. Plan de liberación: diagrama de liberación

Todos los componentes y módulos del sistema se liberarán en el mismo dispositivo. Por tanto, no es necesario especificar un diagrama de liberación.

6.4. Diseño y prototipo de interfaces de usuario

En un primer prototipo de la interfaz, seguimos un diseño clásico con una barra de menú con todas las opciones del programa, una barra de herramientas con las opciones más usadas, como es lanzar GraphPlan con las dos implementaciones usadas.

La zona inferior está dividida en dos zonas. En la parte superior tenemos la zona de salida del programa, donde el usuario va a poder ver el resultado de la aplicación del algoritmo a sus archivos. La parte de abajo está dividida en dos partes. A la izquierda tenemos el archivo de dominio y a la derecha el archivo de problema. De este modo, el usuario tiene siempre los dos archivos presentes y puede editarlos a voluntad. Además, como cada cuadro es una ventana MDI, puede tener varios archivos abiertos en cada zona a la vez.

Cada una de estas zonas de edición tiene los botones clásicos: nuevo, abrir, guardar, guardar como, deshacer, rehacer, cortar, copiar, pegar, imprimir y un botón para introducir los códigos más comunes en un archivo de dominio y un archivo de problema.

Además, se contempla en el menú el caso de uso de Ejemplos para que el usuario pueda probar la aplicación con distintos ejercicios que haya podido realizar en clase.

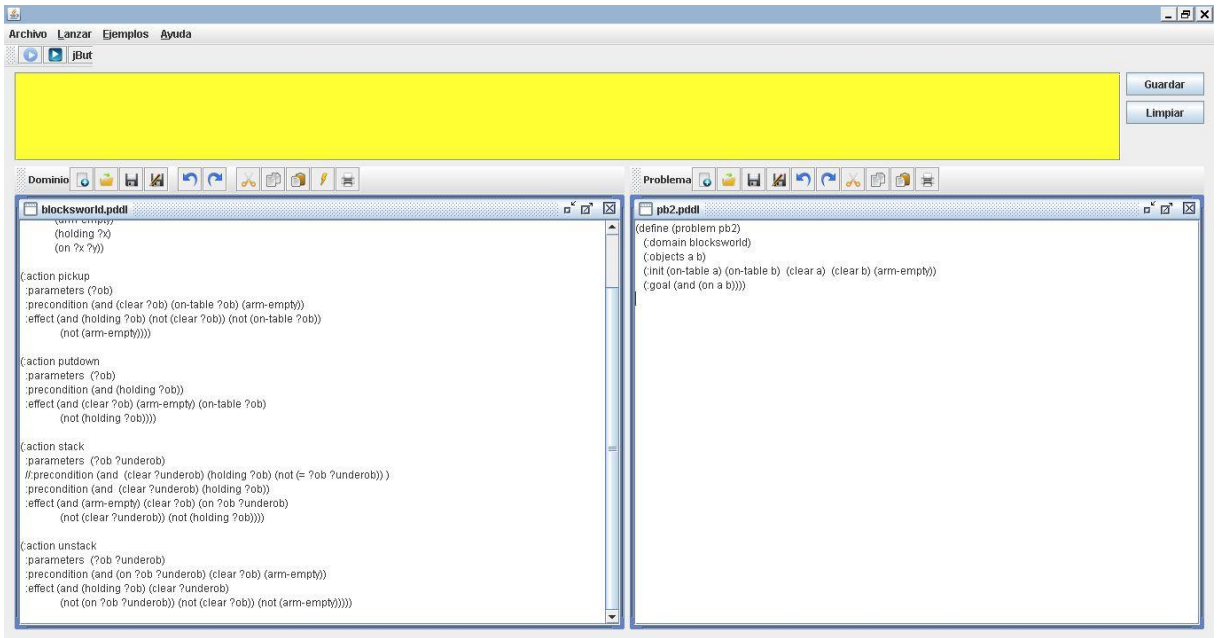


Figura 6.5. Prototipo de interfaz

7. MANUAL DE USUARIO

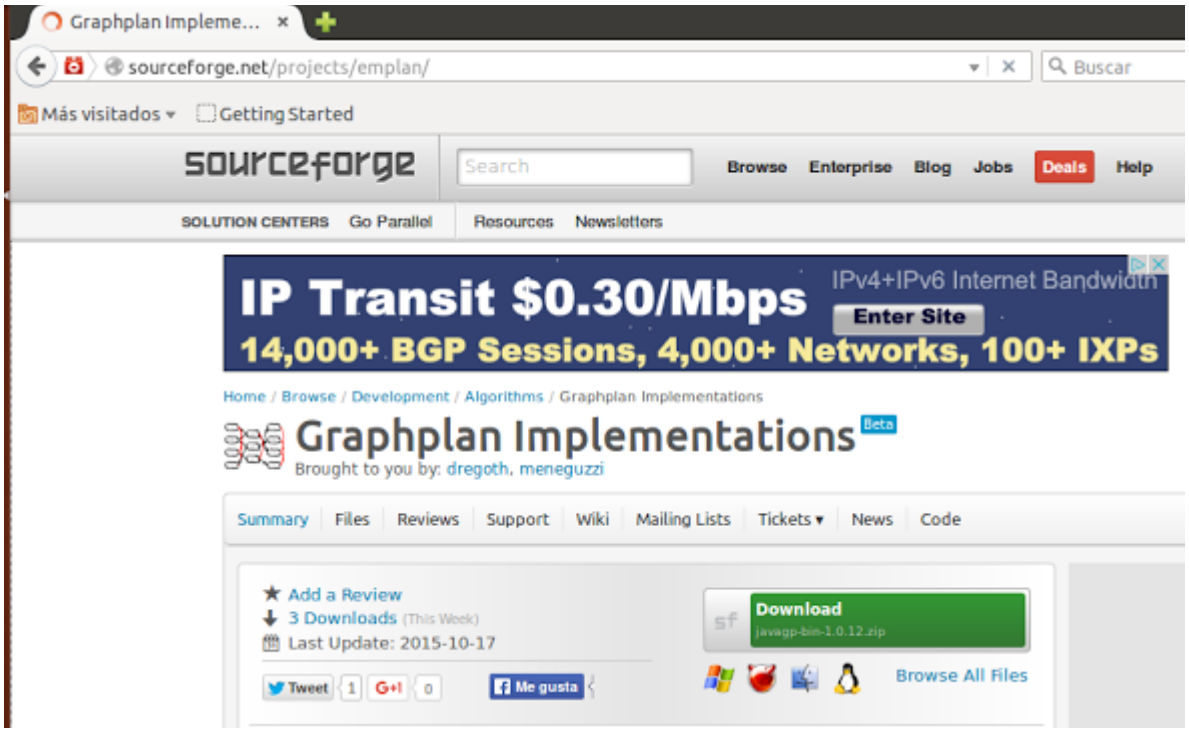
7.1. Documentación del sistema

La documentación del sistema que consta de una guía de instalación y una serie de tutoriales que estarán contenidos en el CD así como en Youtube para que sea más fácil su acceso.

7.2. Instalación

Para instalar el programa, hay que seguir los siguientes pasos:

1. Descargar JavaGP desde la dirección http://sourceforge.net/projects/emplan/ o usar el que viene en la carpeta JAVAGP de la instalación.



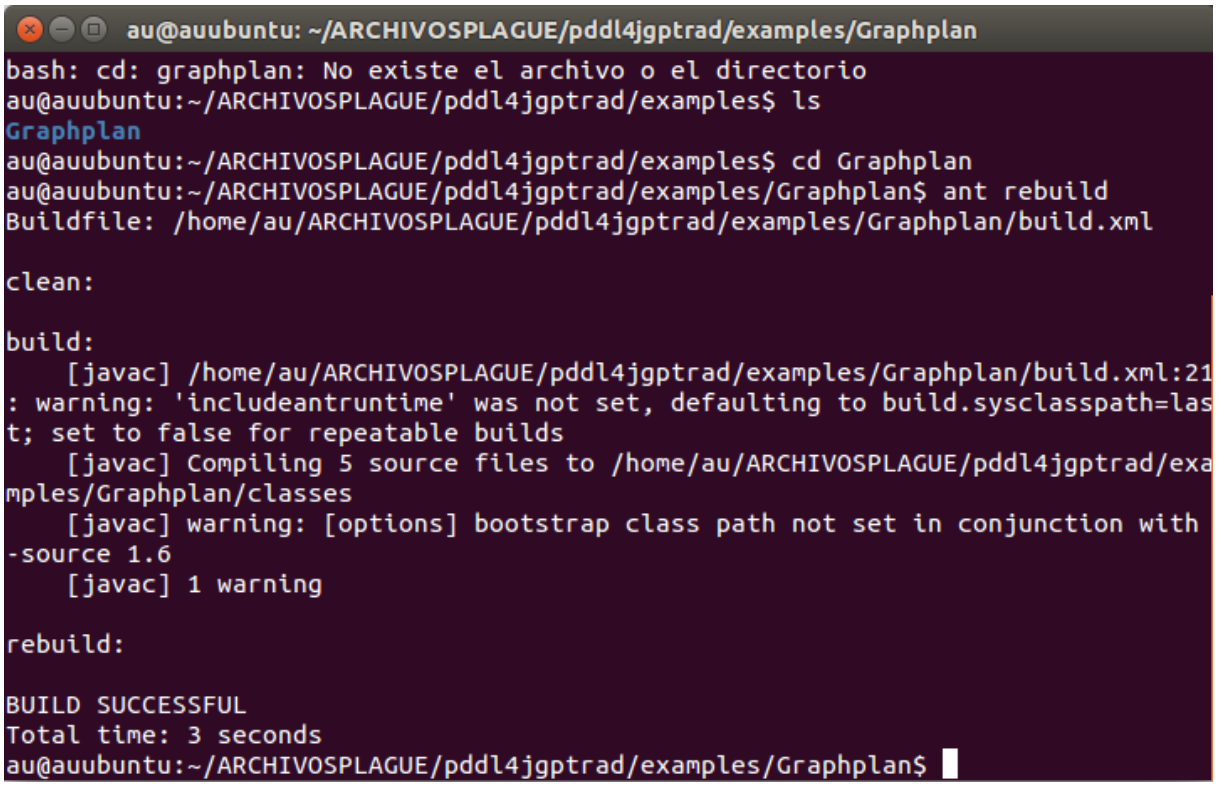
2. Descomprime el archivo en la carpeta que quieras, pero acuérdate de ella, porque la usaremos más adelante para configurar PLAGUE.

Nosotros lo hemos hecho en una carpeta llamada ARCHIVOSPLAGUE.

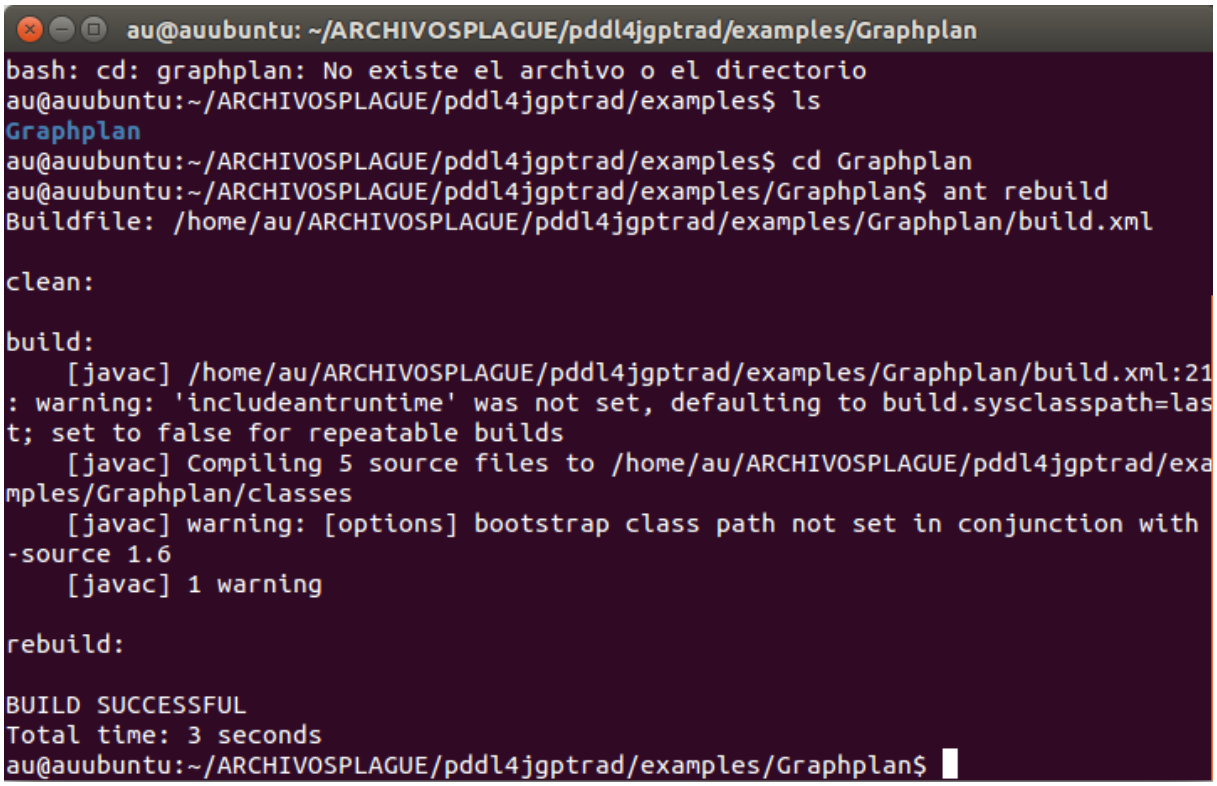
3. Luego, en un derroche de imaginación, hemos cambiado el nombre de la carpeta donde está JAVAGP a JAVAGP

4. En el instalador encontrarás la carpeta *pddl4jgptrad* donde tienes PDDL4J. Cópiala a la misma carpeta donde has puesto JAVAGP (no es necesario pero así mantenemos cierta coherencia) y descomprímela.

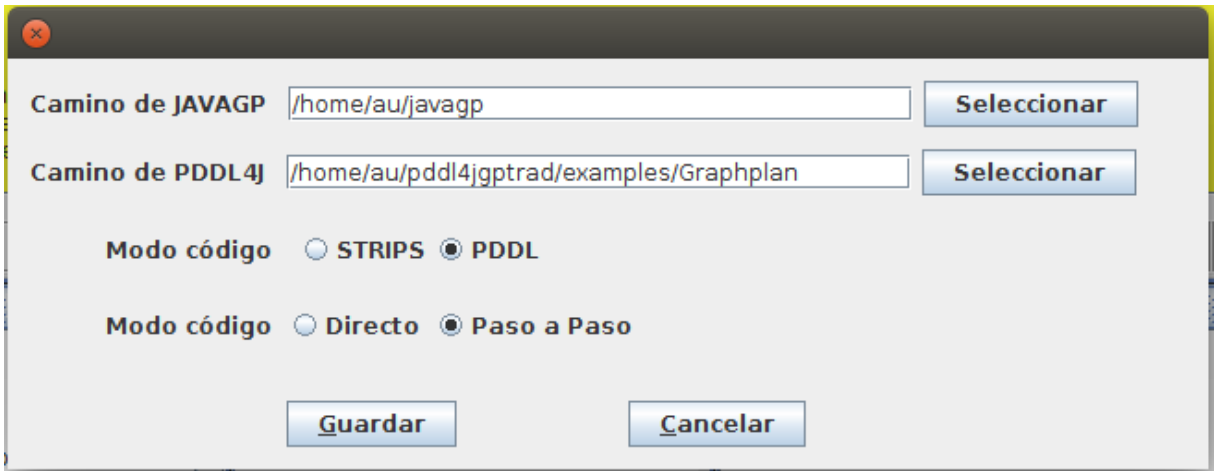
5. Ahora tenemos que hacer encaje de bolillos. JAVAGP funciona bien, pero PDDL4J da ciertos problemillas si no se recompila, así que vamos a hacerlo. Ve a Símbolo de sistema.
6. Entra en el directorio PDDL4JGPTRAD.
7. Entra en el directorio EXAMPLES.
8. Entra en el directorio GRAPHPLAN.
9. Ejecuta **ant rebuild**



10. A continuación, ejecuta **ant jar.**



11. Recuerda que, cuando ejecutes Plague por primera vez, tendrás que especificar la carpeta donde lo acabas de instalar seguido de examples/Graphplan.



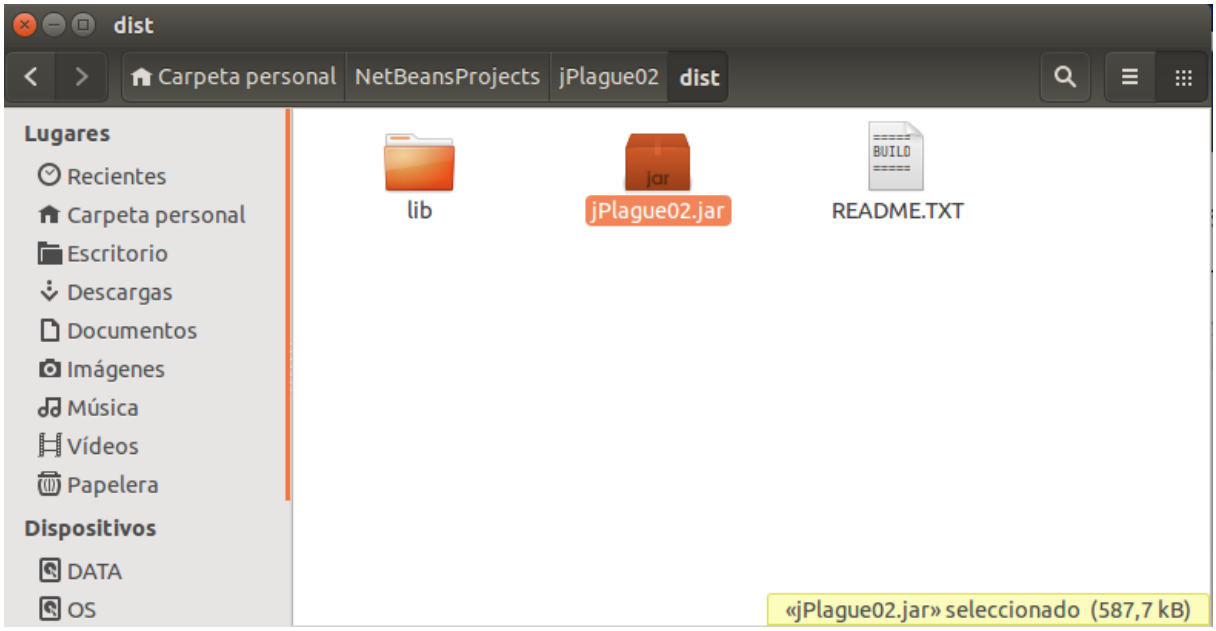
12. Entra en la carpeta donde has instalado PDDL4J.
13. Entra en la carpeta examples.
14. Entra en la carpeta Graphplan.
15. Escribe **ant rebuild** y pulsa la tecla Intro.

```
au@auubuntu: ~/ARCHIVOSPLAGUE/pddl4jgptrad/examples/Graphplan
bash: cd: graphplan: No existe el archivo o el directorio
au@auubuntu:~/ARCHIVOSPLAGUE/pddl4jgptrad/examples$ ls
Graphplan
au@auubuntu:~/ARCHIVOSPLAGUE/pddl4jgptrad/examples$ cd Graphplan
au@auubuntu:~/ARCHIVOSPLAGUE/pddl4jgptrad/examples/Graphplan$ ant rebuild
Buildfile: /home/au/ARCHIVOSPLAGUE/pddl4jgptrad/examples/Graphplan/build.xml

clean:

build:
    [javac] /home/au/ARCHIVOSPLAGUE/pddl4jgptrad/examples/Graphplan/build.xml:21
: warning: 'includeantruntime' was not set, defaulting to build.sysclasspath=las
t; set to false for repeatable builds
    [javac] Compiling 5 source files to /home/au/ARCHIVOSPLAGUE/pddl4jgptrad/exa
mples/Graphplan/classes
    [javac] warning: [options] bootstrap class path not set in conjunction with
-source 1.6
    [javac] 1 warning

rebuild:

BUILD SUCCESSFUL
Total time: 3 seconds
au@auubuntu:~/ARCHIVOSPLAGUE/pddl4jgptrad/examples/Graphplan$
```

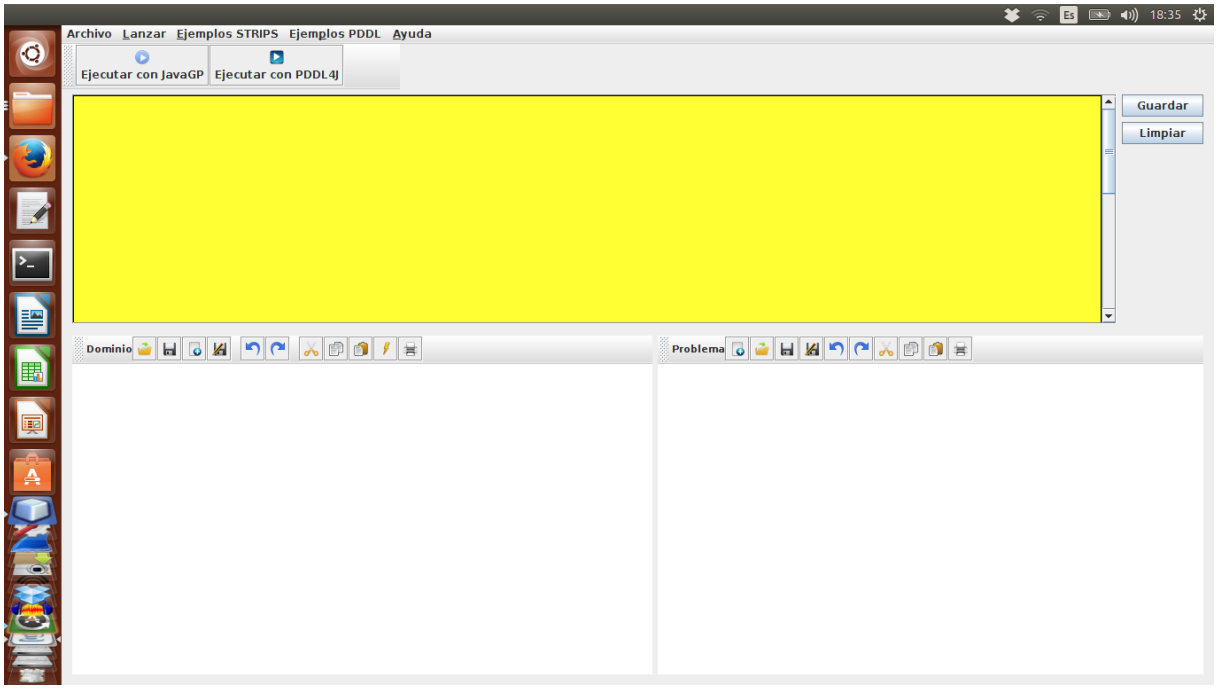
16. Escribe **ant jar** y pulsa la tecla Intro.

```
au@auubuntu:~/ARCHIVOSPLAGUE/pddl4jgptrad/examples/Graphplan$ ant jar
Buildfile: /home/au/ARCHIVOSPLAGUE/pddl4jgptrad/examples/Graphplan/build.xml

jar:
      [jar] Building jar: /home/au/ARCHIVOSPLAGUE/pddl4jgptrad/examples/Graphpl
n/graphplan.jar

BUILD SUCCESSFUL
Total time: 0 seconds
au@auubuntu:~/ARCHIVOSPLAGUE/pddl4jgptrad/examples/Graphplan$
```

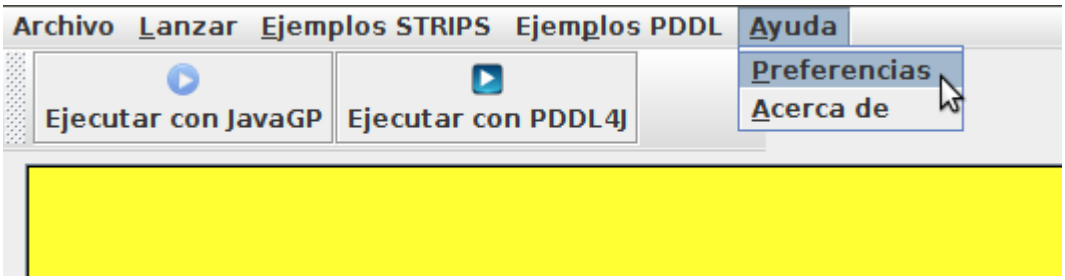
17. En la carpeta **dist** de la carpeta de proyecto se puede encontrar el archivo **jPlague02.jar** que permite ejecutar la aplicación. Para abrirlo basta con hacer doble clic sobre el icono.



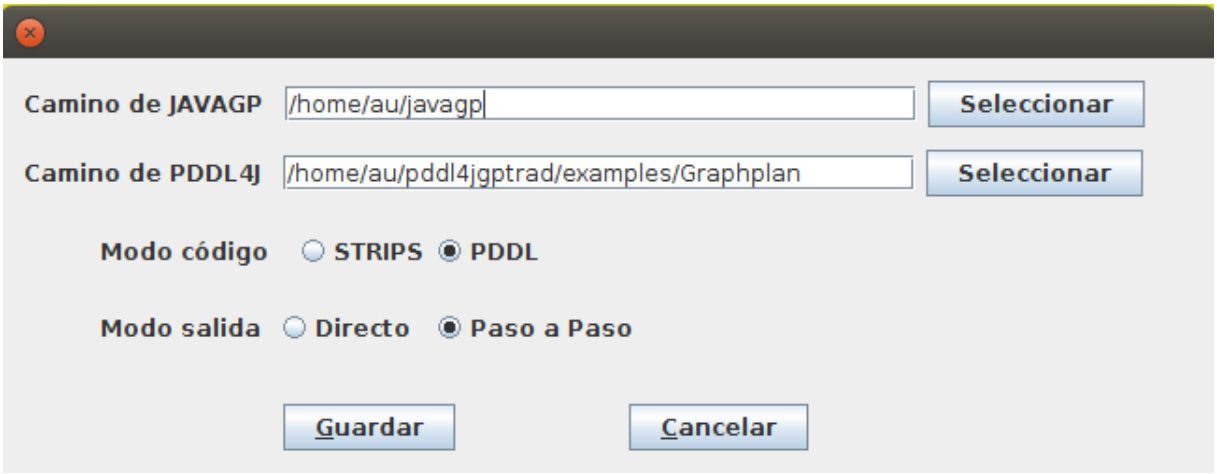
18. Y aquí tenemos el programa.



19. Para que funcione, lo primero que tenemos que hacer es abrir la ventana de configuración para indicar los directorios donde están JAVAGP y PDDL4J. Para ello, haz clic en el menú *Ayuda*.
20. Haz clic en la opción *Preferencias*.



21. Se abre una ventana donde se pueden especificar varias opciones. La primera que tienes que definir es el camino a la carpeta JAVAGP. Haz clic en el botón *Seleccionar* y busca la carpeta.
22. También has de seleccionar la carpeta donde está PDDL4J. Haz clic en el botón Seleccionar a la derecha y selecciona la carpeta donde has colocado PDDL4JTRAD/EXAMPLES/GRAPHAPLAN.

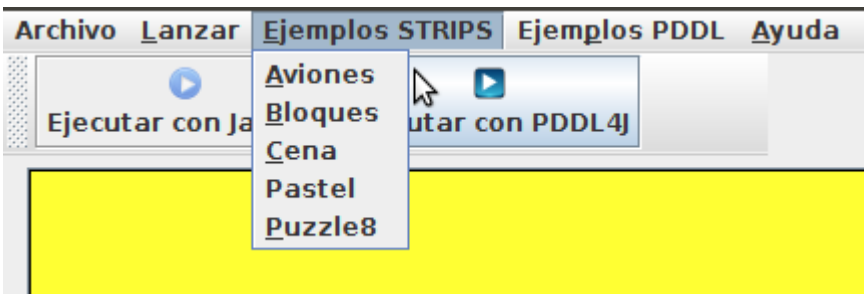


23. El programa también tiene la opción de poner el editor en STRIPS o en PDDL.
24. Por último, puedes seleccionar si quieres una salida directa en la zona superior de la ventana o una salida Paso a paso con ventanas que van indicando los distintos niveles que se van ejecutando en GraphPlan.
25. Una vez configurado haz clic en *Guardar*.

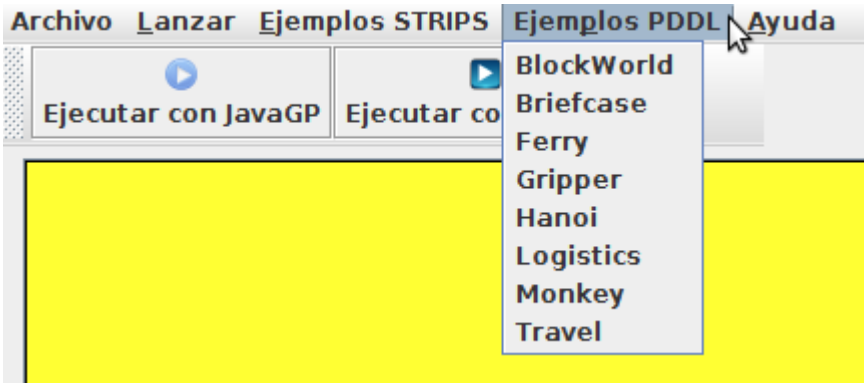
7.3. Ejemplos

En el menú de la aplicación puedes observar como tienes varios archivos de ejemplo de STRIPS y PDDL que se cargan automáticamente para que empieces a *cacharrear* un poco y te familiarices con ambos lenguajes.

Para STRIPS, tienes varios problemas clásicos como el de los aviones, el mundo de los bloques, la cena que hemos glosado en este trabajo, hacer un pastel y el Puzzle8.

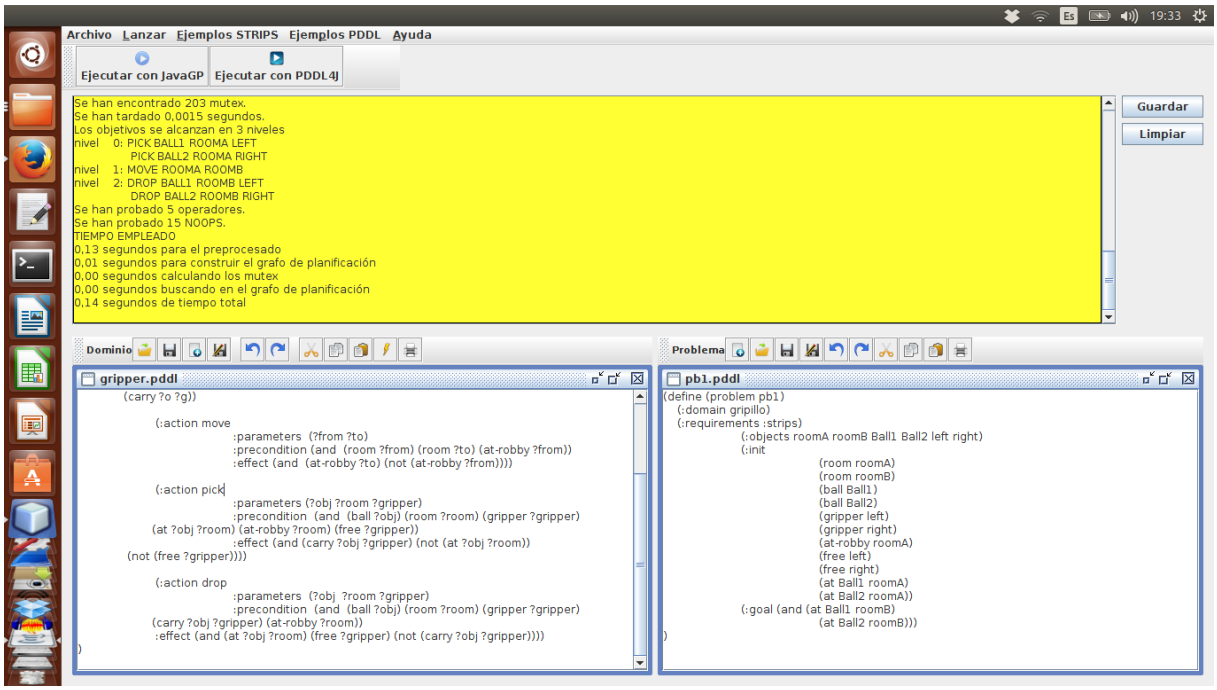


Para PDDL tienes algunos más, como la maleta o briefcase, el robot Gripper, las torres de Hanoi, el mono y el plátano, etc.



Para cargar los archivos, no tienes más que hacer clic sobre los mismos y se cargarán tanto el archivo de dominio como el archivo de problema.

Carga el ejemplo de PDDL Gripper y ejecútalo. Obtendrás la siguiente solución.



El botón *Guardar* en la parte derecha de la ventana de salida permite salvar en un archivo la salida en modo texto, de modo que se pueda adjuntar como solución a los ejercicios que el usuario/a o alumno/a tenga que realizar en clase.

Para ver tutoriales sobre el uso del programa y creación de archivos en PDDL y STRIPS, los usuarios/as o alumnos/as pueden acceder al canal de Youtube https://www.youtube.com/channel/UCTb3f9inGPUN_bj9pwGeaOg donde encontrarán información actualizada y tutoriales.

8. CONCLUSIONES

El aprendizaje de lenguajes como STRIPS y PDDL se realiza editando archivos de texto y lanzándolos mediante software de línea de comando como JAVAGP o PDDL4J. Con PLAGUE hemos pretendido hacer una aplicación en entorno gráfico que permita realizar la edición de archivos y su lanzamiento sin tener que acceder a línea de comando. La aplicación contiene lo necesario para que un alumno o alumna que esté aprendiendo STRIPS o PDDL pueda editar los archivos fácilmente, lanzarlos y obtener la salida rápidamente.

Java nos ha permitido desarrollar una aplicación usando la metodología RAD (Rapid Application Development) de forma rápida, haciendo uso de toda la potencia del lenguaje. Las tareas de edición de texto y de trabajo con archivos son provistas por librerías preparadas a tal efecto, con lo que el trabajo de desarrollo ha sido cómodo

La parte principal del desarrollo ha sido la interpretación de la salida que dan tanto JavaGP como PDDL4J, que ha sido analizado paso a paso y traducida en caso necesario de modo que cada paso del algoritmo quede claro al alumno o alumna.

Como conclusión, podemos decir que hemos realizado en este trabajo enfocado desde un punto de vista eminentemente pedagógico un programa orientado al aprendizaje de STRIPS y PDDL, así como la comprensión del algoritmo GraphPlan, que esperemos ayude al alumnado en la tarea.

9. MEJORAS

Para siguientes desarrollos, se observa que se pueden realizar las siguientes mejoras:

- Mostrar los grafos de planificación para cada paso de GraphPlan.
- Identificar los mutex en el grafo de planificación.
- Traducir el código STRIPS a código PDDL y vicecersa.
- Mejorar el editor para el uso de pestañas.

10. REFERENCIAS

1. **M. Ghallab, D.Nau, P. Traverso.** *Automated Planning, Theory and Practice.* San Francisco : Elsevier Inc., 2004.

2. **A. Blum, M. Furst, J. Langford.** Graphplan Home Page. [En línea] School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213-3891, Junio de 2001. http://www.cs.cmu.edu/~avrim/graphplan.html.

3. **Meneguzzi, F.** Graphplan Implementations. [En línea] SourceForge, 2010. [Citado el: 04 de 08 de 2015.] http://emplan.sourceforge.net/.

4. **Ambite, José Luis.** [En línea] [Citado el: 06 de 08 de 2015.] http://www.isi.edu/~blythe/cs541/Slides/2003-9-4-graphplan.pdf.

5. **S. Russell, P.Norvig.** *Artificial Intelligence, a Modern Approach 3rd Edition.* Upper Saddle River, New Jersey : Pearson Education Inc., 2010.

6. **Weld, Daniel S.** *Recent Advances in AI Planning.* Seattle : s.n., 1998.

7. *Fast planning through planning graph analysis.* **Brum, Avrim L. y Furst, Merrick L.** 90, Pittsburgh : Elsevier, 1996, Artificial Intelligence.

8. **Brian C. Williams, Maria Fox.** MIT Open Courseware. *Massachussets Institute of Technology.* [En línea] [Citado el: 15 de 08 de 2015.] http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-410-principles-of-autonomy-and-decision-making-fall-2010/lecture-notes/MIT16_410F10_lec08a.pdf.

9. **Veloso, M.** Department of Computer Science, University of Toronto. [En línea] 10 de 2002. [Citado el: 30 de 08 de 2015.] http://www.cs.toronto.edu/~sheila/384/w11/Assignments/A3/veloso-PDDL_by_Example.pdf.

10. **Alcalde, Alejandro.** github. [En línea] 30 de 05 de 2015. [Citado el: 30 de 08 de 2015.] https://github.com/algui91/PDDL-Problems/blob/master/Sesion1/ejemplos/PDDL%20by%20Example/typed-gripper-domain.pddl.

11. **Committee, AIPS 98 Planning Competition.** *The Planning Domain Definition Language v1.2.* New Haven : Yale University, 1998. Tech Report CVC TR-98-003/DCS TR-1165.

12. **Littman, Michael L. and Younes, Hakan L.S.** *PPDDL1.0: An Extension to PDDL for Expressing Planning Domains with Probabilistic Effects.* Rutgers University, Carnagie Mellon University. Pittsburgh, Piscataway : s.n., 2004. p. 17.

13. **Saigol, Dr Zeyn.** PDDL-driven GraphPlan implementation. [En línea] 2 de Octubre de 2008. [Citado el: 02 de Septiembre de 2015.] www.zeynsaigol.com/software/graphplanner.html.

14. **AIPS-98 Planning Competition Committee.** *PDDL - The Planning Domain Definition Language 1.2.* Yale : s.n., 1998.

15. **Lifschitz, Vladimir.** Página oficial de Vladimir Lifschitz. [En línea] 3 de 4 de 1998. [Citado el: 4 de 9 de 2015.] http://www.cs.utexas.edu/~vl/teaching/planning/pddl/domains/fridge.pddl.

16. **Martin, James.** *Rapid Application Development.* USA : MacMillan, 1991.

17. **OMG Object Management Group.** Unified Modeling Language™ (UML®) Resource Page. [En línea] OMG Object Management Group, 2015. [Citado el: 26 de 9 de 2015.] http://www.uml.org/.