**GP** **(*GraphPlan*)**

Uno de los principales problemas que presentan los algoritmos de búsqueda es el **elevado factor de ramificación**, esto es, el máximo número de hijos por cada nodo. Además, realizar una búsqueda *hacia atrás* puede suponer desarrollar multitud de nodos que probablemente no lleguen al objetivo, lo que se traduce en una pérdida de tiempo y cómputo.

Por tanto, ¿Cómo podemos reducir el factor de ramificación? Para ello, debemos crear un problema *más relajado*:

* Eliminando en primer lugar, algunas restricciones del problema original. Lo que queremos es un problema que se pueda resolver en tiempo polinómico.
* Por tanto, las soluciones del *problema relajado* incluirán todas aquellas del problema original.

Una vez llegada a la solución, aplicamos la búsqueda *hacia atrás*, teniendo en cuenta todas las restricciones y conflictos.

Veamos el funcionamiento del planificador. *Graphplan* divide el proceso de planificación en dos fases:

* La primera fase empieza en el estado inicial, en el cual se disponen todas las proposiciones que son ciertas en ese momento y se genera un primer conjunto de acciones que se pueden ejecutar en el estado inicial. De esta forma, se va expandiendo con niveles alternados de proposiciones y acciones hasta que en un determinado nivel de proposiciones aparecen como alcanzables las metas del problema. Este primer proceso se denomina **análisis de alcanzabilidad**.
* Después de ese primer proceso, se realiza un proceso de búsqueda en el que se intenta encontrar un plan válido que alcance esas metas a través de buscar en ese conjunto de niveles de proposición y de acciones. Es importante remarcar lo de **válido**, puesto que tenemos tener en cuenta cualquier conflicto o contradicción entre predicados/acciones en la planificación. En caso de que no logre alcanzar el siguiente nivel sin violar ninguna restricción, se vuelve hacia atrás, expandiendo de nuevo el grafo desde otro camino. A este proceso se le llama **extracción de la solucion**.

procedimiento extraer-solucion(g <- metas, j <- nivel del grafo):

# 0 equivale al estado inicial

si j = 0 entonces

devolver la solucion

fsi

para cada predicado l en g hacer

elegir\_accion

fpara

si mutex(accion) entonces

volver atras

fsi

g' = {listado con las precondiciones de las acciones elegidas}

extraer-solucion(g', j-1)

fprocedimiento

El proceso para extraer solución consiste en lo siguiente:

1. Comenzamos desde el estado final o meta (j ≠ 0). Como condición de parada, si j = 0 significa que estamos en el estado inicial y, por ello, obtenemos la solución al problema.
2. En caso contrario, debemos recorrer cada uno de los niveles del grafo hasta llegar al estado inicial. En cada nivel, elegimos de forma no determinista aquellas acciones, usadas en el siguiente estado *sj-1*, que permitan alcanzar cada uno de los nodos objetivo.
3. Una vez elegidas estas acciones, debemos comprobar si existen parejas de acciones *mutex*, es decir, excluyentes entre sí. En caso afirmativo, volvemos hacia atrás, escogiendo otras acciones.
4. Por último, escogemos aquellas precondiciones de las acciones que hemos elegido, llamando recursivamente con ese listado de precondiciones y el nuevo nivel a explorar.

Esta búsqueda de un plan se basa en un **grafo de planificación**. Este tipos de grafos son **dirigidos** y separados por **capas**. Por un lado, distinguimos dos tipos de nodos:

* **Nodos de proposición**: aparecen en los niveles impares, comenzando desde el estado inicial (0).
* **Nodos de acción**: aparecen en los niveles pares.

Por otra parte, existen tres tipos de arcos:

* Arcos de añadido (*add*): para los efectos positivos de una acción.
* Arcos de borrado (*del*): para los efectos negativos de una acción.
* Arcos ficticios (*no-op*): para pasar de un estado a otro manteniendo la veracidad del predicado.

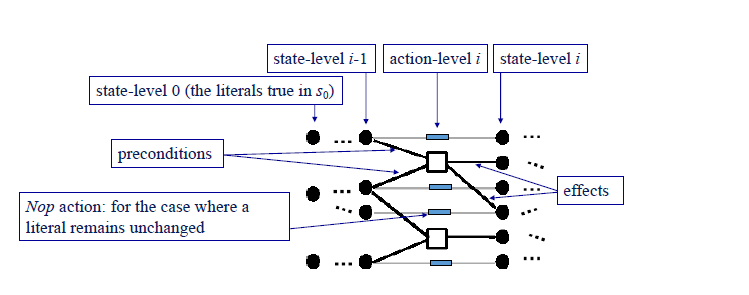


Ilustración 1. Tipos de nodos y arcos

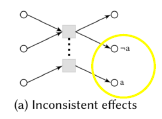
Como hemos comentado antes, a lo largo del proceso de extracción de la solución se pueden producir situaciones en las que las acciones sean **mutuamente excluyentes** (*mutex*). Esto ocurre cuando entre dos acciones:

* Tienen efectos opuestos
* Precondiciones incompatibles entre ambas
* Cuando el efecto de una acción niega la precondición de la otra

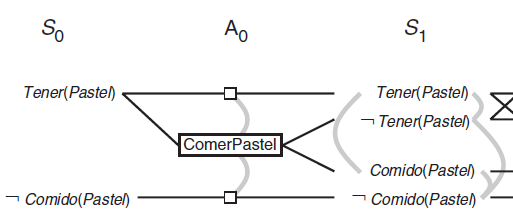
No solo se pueden dar en acciones, sino también en proposiciones, en el caso de que en un nivel aparezcan un predicado como su negado, o incluso todas las acciones que hagan que sean posible estos predicados sean *mutex* en el estado anterior.

A continuación, vamos a explicar cada uno de los *mutex* que pueden generarse:

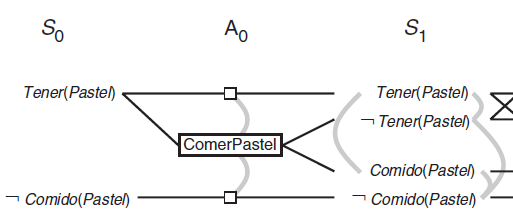
* **Efectos inconsistentes** (*inconsistent effects*): se produce cuando los efectos de una acción niegan los efectos de otra.



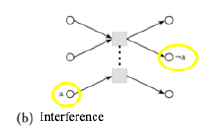
Veamos un ejemplo:



Por un lado, tenemos la acción *no-op* *tener(pastel)* en el estado inicial, que pasa al siguiente nivel *S1*, manteniendo el predicado. Por otro lado, en *A0* se realiza la acción *comerPastel* ya que requiere como precondición que tengamos el pastel ya preparado. Como consecuencia de esta acción, se genera en *S1* el predicado *no(tener(pastel))*, el cual es excluyente con el anterior. Lo mismo ocurre con *comido(pastel)* y *no(comido(pastel))*.

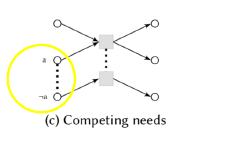


* **Interferencia** (*interference*): se produce cuando las precondiciones de una acción anulan los efectos de otra.

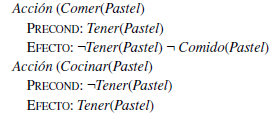


Como ejemplo, la acción de *comer(pastel)* produce como efecto la negación de *tener(pastel)*, anulando los efectos de la acción *no-op* *tener(pastel)*.

* **Necesidades que entran en competencia** (*competing needs*): se produce cuando la precondición de una acción anula la precondición de otra acción en un mismo nivel.



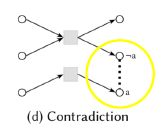
Por ejemplo, supongamos que tenemos las siguientes acciones:



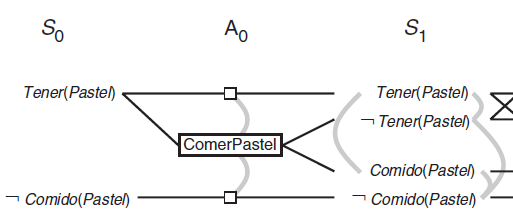
En este caso, las acciones *comer(pastel)* y *cocinar(pastel)* entran en competencia, ya que la precondición *tener(pastel)* de la primera entra en conflicto con la precondición de la segunda acción: *no(tener(pastel))*.

Por otro lado, también existen *mutex* entre proposiciones:

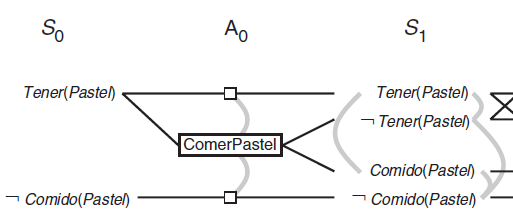
* **Contradicción** (*contradiction*): se produce cuando los efectos de una acción anulan los efectos de otra.



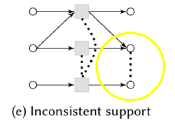
Si volvemos con el ejemplo anterior, podemos comprobar como en *S1* tenemos tanto *tener(pastel)* como *no(tener(pastel))*, lo que supone una contradicción entre ambas proposiciones:



Lo mismo ocurre entre las proposiciones *comido(pastel)* y *no(comido(pastel))*.



* **Soporte de inconsistencia** (*inconsistent support*): ocurre cuando todas las formas de alcanzar de conseguir ambos efectos son *mutex* entre sí.



Pongamos el ejemplo anterior: por un lado, tenemos un soporte de inconsistencia entre los efectos *tener(pastel)* y *comido(pastel)*. Para tener la acción *tener(pastel)* necesitamos de la acción *no-op* *tener(pastel)* que es *mutex* con *comer(pastel)*. Por su parte, *comido(pastel)* necesita de la acción *comer(pastel)* y que es mutex con *tener(pastel)*. Lo mismo ocurriría con *no(tener(pastel))* y *no(comido(pastel))*.

Una vez visto el funcionamiento de un planificador de grafos, veamos un ejemplo. Tenemos las siguientes acciones:

(:action Eat

:parameters (?cake)

:precondition (and (have ?cake))

:effect (and (eaten ?cake) (not (have ?cake)))

(:action Bake

:parameters (?cake)

:precondition (not (have ?cake))

:effect (and (have ?cake))))

Por otro lado, tenemos el estado inicial y las metas:

Initial State

Have (cake) & (not (Eaten(cake))

Goal

Eaten (cake) and Have (cake)

En primer lugar, definimos el estado inicial *S0* en el que tenemos los predicados marcados que son verdaderos en ese instante.

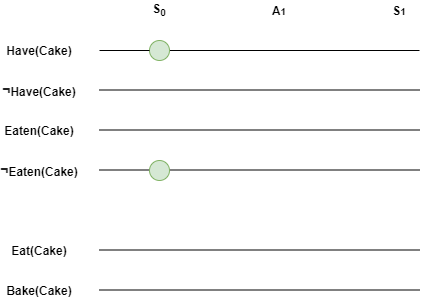


Ilustración 2. Estado S0

Como podemos observar en la imagen anterior, los nodos marcados en verde corresponden con los predicados que son verdaderos en el estado inicial. A continuación, en *A1* analizamos **qué acciones son posibles dadas las precondiciones del estado anterior**. En este caso, solo es posible la acción *eat(cake)*,cuyas precondición es *have(cake)*. Una vez ejecutada la acción, los efectos se propagan a *S1*. En este caso, los efectos serán tanto *not(have(cake))* como *eaten(cake)*.

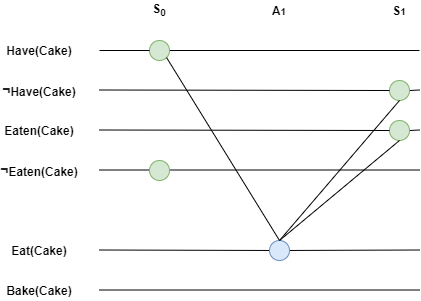


Ilustración 3. Ejecución de eat(cake)

No obstante, no solo aparecen en *A1* todas las acciones que son posibles en *S0*, sino que además siguen siendo verdaderos los predicados del estado anterior, a través de acciones *no-op* en *A1*.

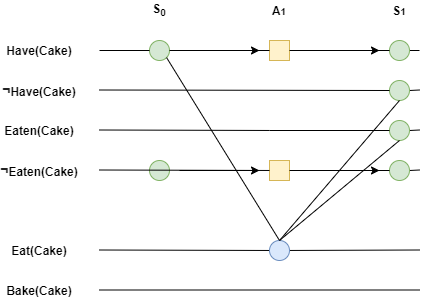


Ilustración 4. Acciones no-op

Una vez propagadas estas acciones, comprobamos que, efectivamente, en el nivel *S1* tenemos todos los predicados del nodo meta. Por tanto, la primera parte de la planificación ya está completada. Ahora, nos queda obtener un plan, partiendo desde el objetivo hasta el estado inicial, **observando cualquier mutex o inconsistencia en el grafo**. Por ello, analizamos los mutex existentes:

* **Efectos inconsistentes**: se produce entre los predicados *eat(cake)* y la acción *no-op* *have(cake)*, así como con *eat(cake)* y la acción *no-op not(eaten(cake))*.
* **Contradicciones**: al igual que en los **efectos inconsistentes**, se produce entre *have(cake)* y *not(have(cake))*, además de *eaten(cake)* y *not(eaten(cake))*.
* **Soporte inconsistente**: por un lado, *have(cake)* y *eaten(cake)* son *mutex* ya que la única forma de conseguir *have(cake)* es a través de la acción *no-op* *have(cake)* que es inconsistente con *eat(cake)*, mientras que *eat(cake)* necesita la acción *eaten(cake)* que es inconsistente con la acción *no-op* *have(cake)*.

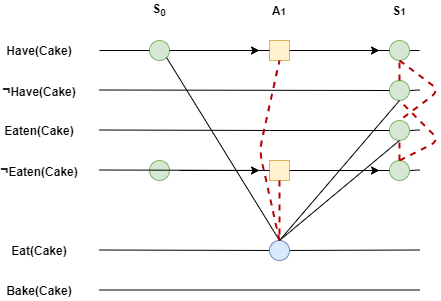


Ilustración 5. acciones mutex en S1

Una vez tengamos todos los *mutex*, debemos volver hacia atrás sin contradecir proposiciones. Si nos fijamos en *have(cake)* la única manera de volver hacia atrás es mediante la acción *no-op* *have(cake)*. Sin embargo, esta acción es *mutex* con *eat(cake)*, por lo que no podemos volver a *S0*. Lo mismo ocurriría con *eaten(cake)* ya que requiere de la acción *eat(cake)* y esta es *mutex* con *have(cake)*. En conclusión, desde *S1* no se puede llegar a una solución. Por tanto, debemos continuar expandiendo el grafo.

Desde *S1* analizamos qué acciones son posibles en *A2*, además de expandir los predicados de *S1*, a través de acciones *no-op*.

Nuevamente, comprobamos si desde *S2* hemos alcanzado los predicados del nodo meta. De nuevo, vemos que en *S2* se alcanzan ambos predicados*.* Ahora, nos queda otra vez obtener un plan, partiendo desde el objetivo hasta el estado inicial, **observando cualquier mutex o inconsistencia en el grafo**. Por ello, analizamos de nuevo los mutex existentes:

* **Efectos inconsistentes**:
  + *have(cake)* y *not(have(cake))*
  + *eaten(cake)* y *not(eaten(cake))*
  + *bake(cake)* y *not(have(cake))*
  + *have(cake)* y *eat(cake)*
  + *not(eaten(cake))* y *eat(cake)*
  + *bake(cake)* y *eat(cake) -> realmente es competing need aunque podria se tambien un efecto inconsistente al generar dos efectos inconsistentes*
* **Necesidades que entran en competencia**: entre las acciones *bake(cake)* y *eat(cake)*, puesto que necesitan tanto *have(cake)* como *not(have(cake))*, respectivamente.
* **Contradicción**: entre *have(cake)* y *not(have(cake))* además de *eaten(cake)* y *not(eaten(cake))*.

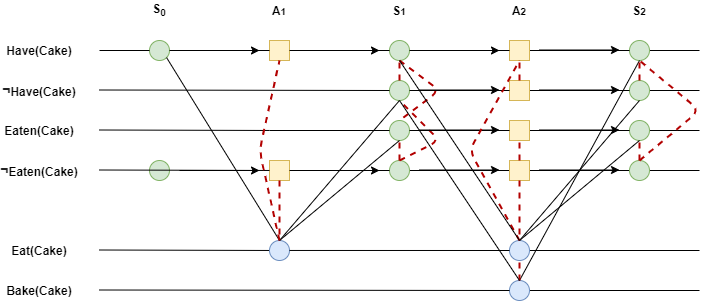


Ilustración 6. acciones mutex en S2

Una vez tengamos todos los *mutex*, comprobamos si existe un camino entre *S2* y *S1*. En este caso, desde *have(cake)* se puede llegar a *S1* por *bake(cake)*, mientras que *eaten(cake)* puede alcanzar *S1* mediante la acción *no-op* *eaten(cake)*, evitando exclusiones mutuas. Una vez en *S1*, vemos que las precondiciones necesarias para ir desde *S2* hasta *S1* son *not(have(cake))* y *eaten(cake)*.

Finalmente, tanto desde *not(have(cake))* como *eaten(cake)* se puede llegar a *S0* mediante la acción *eat(cake)*. Una vez hayamos llegado a *S0*, hemos obtenido un plan:

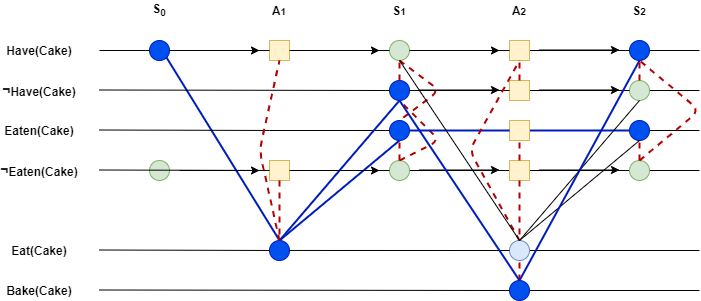
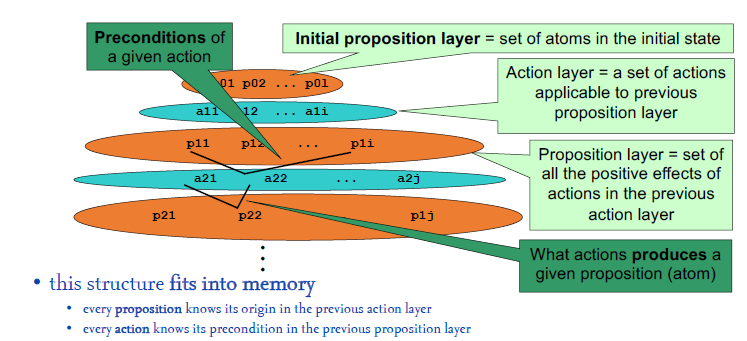


Ilustración 7. Planificación final

***eat(cake)* 🡪 *bake(cake)***

**Como hemos podido comprobar, cada proposición conoce la acción de la capa anterior y cada acción conoce la precondición de la anterior capa**.



Una vez visto el funcionamiento de *graphplan*, veamos una comparación con la búsqueda en espacio de planes:

* **Ventajas**
  + Por un lado, este tipo de planificación, en el proceso de extracción de la solución **busca sólo las acciones y estados previos desde el cual el objetivo puede ser alcanzado**, podando muchas de las acciones.
  + Además, el espacio de búsqueda es mucho más reducido que en la planificación en espacio de planes y más rápida.
* **Inconvenientes**
  + El principal inconveniente de *graphplan* es el hecho de genera una enorme cantidad de predicados durante el análisis de alcanzabilidad. Esto provoca que muchos de esos literales sean insignificantes.

Una posible solución para solventar este inconveniente sería asignar tipos de datos a las variables y a las constantes. De esta forma, sólo se instanciarán aquellas que sean del mismo tipo.

**En general, *graphplan* resuelve problemas de planificación clásicos mucho más rápido que PSP**

Existen varios planificadores que utilizan la técnica de *graphplan*:

* **SGP**: planificadores de contingencia
* **TGP**: incluye razonamiento temporal
* **IPP**: permite razonamiento con recursos
* **TPSYS**: combina tanto *graphplan* como **TGP**
* **SAPA**: usa varias heurísticas basadas en distancias, con el fin de controlar la búsqueda
* **STAN**: extrae heurísticas admisibles desde una herramienta de análisis del dominio llamada TIM
* **LPG**: combina *graphplan* y *SAT*

Antes de finalizar con la planificación en grafos, veamos un ejemplo más. Supongamos que queremos preparar una cena sorpresa, y para ello tenemos las siguientes acciones:

|  |  |  |
| --- | --- | --- |
| Accion | Precondición | Efectos |
| *cook()* | *cleanHands* | *dinner* |
| *wrap()* | *quiet* | *present* |
| *carry()* | *-* | *¬garbage, ¬cleanHands* |
| *dolly()* | *-* | *¬garbage, ¬quiet* |

Por otro lado, partimos del siguiente estado inicial:

Y el objetivo consistiría en tener la cena preparada, el regalo envuelto y sacada la basura.

En primer lugar, definimos el estado 0 con los predicados del estado inicial. Hay que remarcar que en *graphplan* se debe añadir todo aquello que sea cierto en el nivel 0. En este caso, serían tanto *garbage*, *cleanHands* y *quiet* como *¬dinner* y *¬present*.

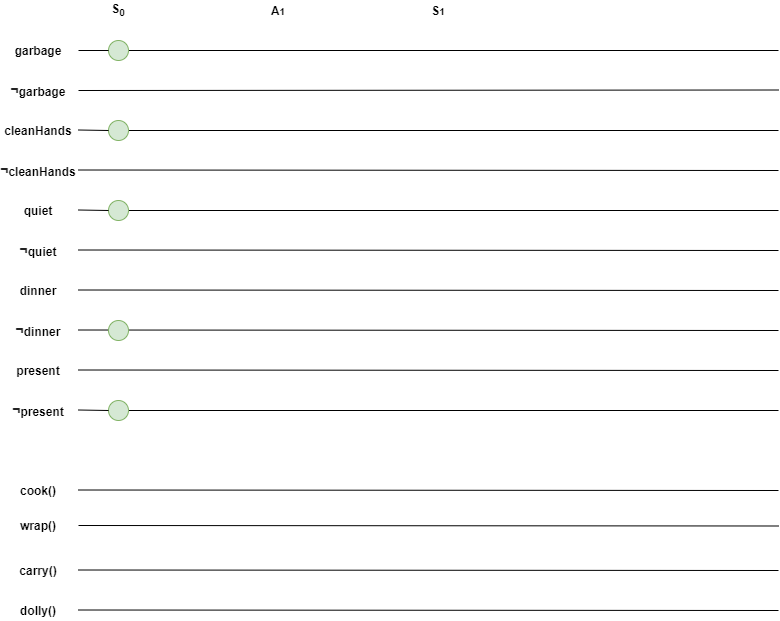


Ilustración 8. Estado inicial S0

A continuación, si nos vamos al nivel 1 de acciones tenemos todas aquellas que son posibles con las precondiciones del nivel de estado 0. En este caso, tanto *cook()* como *wrap()* son posibles dados los literales del estado inicial. Además, como *carry()* y *dolly()* no necesitan precondiciones, se pueden ejecutar sin ningún inconveniente.

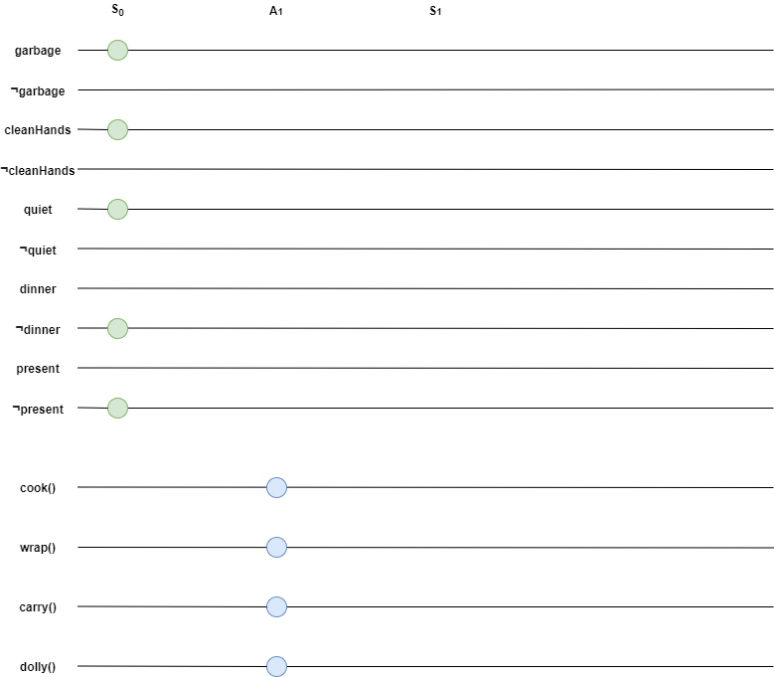
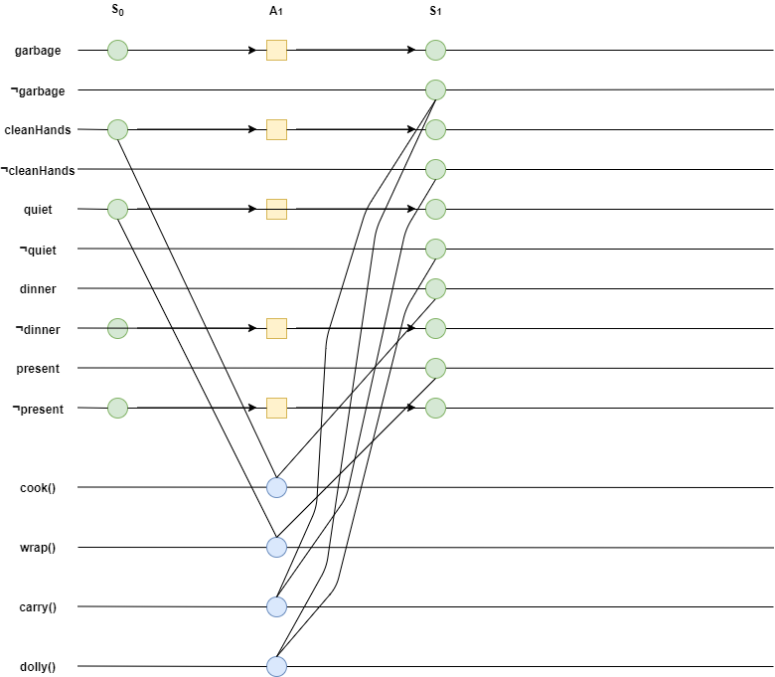


Ilustración 9. acciones en A1

Tras ejecutar las acciones, los efectos de cada una se propagan hacia el siguiente nivel de estado (1), además de los literales que se propagan mediante acciones *no-op*.



Una vez que hayamos propagado los efectos y acciones *no-op*, comprobamos si desde el estado 1 hemos alcanzado los predicados del nodo meta. Como podemos comprobar, en el nivel 1 tenemos todos los literales, incluyendo los del nodo objetivo. A continuación, debemos detectar todos los mutex del grafo.

* **Efectos inconsistentes**:
  + Entre la acción *no-op* *garbage* y *carry()*, ya que genera como efecto *¬garbage*.
  + Entre la acción *no-op* *garbage* y *dolly()*, ya que genera como efecto *¬garbage*.
  + Entre la acción *no-op* *cleanHands* y *carry()*, ya que genera como efecto *¬cleanHands*.
  + Entre la acción *no-op* *quiet* y *dolly()*, ya que genera como efecto *¬quiet*.
  + Entre la acción *no-op* *¬dinner* y *cook()*, ya que genera como efecto *dinner*.
  + Entre la acción *no-op* *¬present* y *wrap()*, ya que genera como efecto *present*.
* **Interferencias**:
  + Entre la acción *dolly()*, que tiene como efecto *¬quiet*, y *wrap()* cuya precondición es *quiet*.
  + Entre la acción *carry()*, que tiene como efecto *¬cleanHands*, y *cook()* cuya precondición es *cleanHands*.
* **Necesidades que entran en competencia**: en este caso no tenemos ningún caso.
* **Contradicciones**:
  + Entre *garbage* y *¬garbage*.
  + Entre *cleanHands* y *¬cleanHands*.
  + Entre *quiet* y *¬quiet*.
  + Entre *dinner* y *¬dinner*.
  + Entre *present* y *¬present*.
* **Soporte de inconsistencia**:
  + Entre *present* y *¬quiet*. Esto se debe a que para obtener el literal *present* es necesaria la acción *wrap()*, y para obtener el literal *¬quiet* se necesita la acción *dolly()*. Sin embargo, ambas acciones son *mútex* entre sí.
  + Entre *¬cleanHand* y *dinner*. Esto se debe a que para obtener el literal *¬cleanHand* es necesaria la acción *carry()*, y para obtener el literal *dinner* se necesita la acción *cook()*. Sin embargo, ambas acciones son *mútex* entre sí.

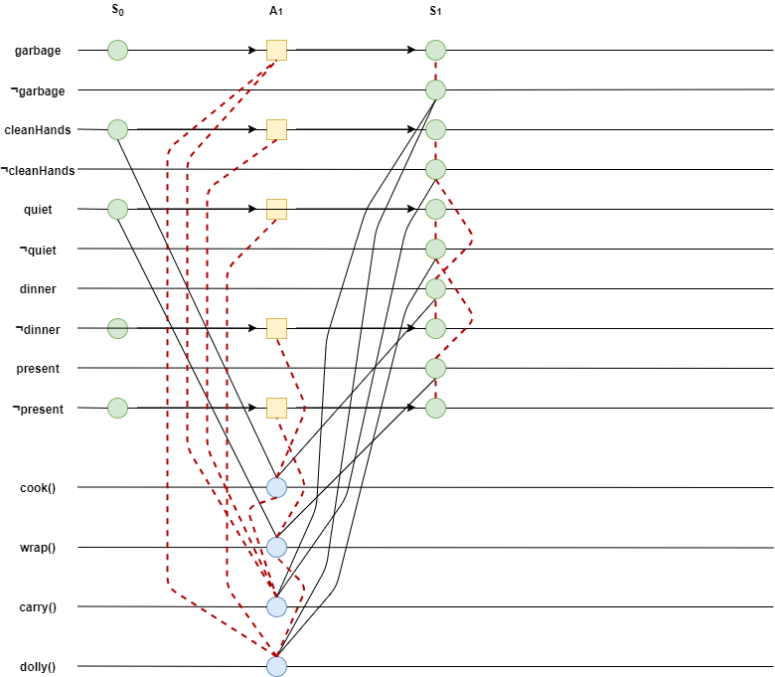


Ilustración 10. acciones y literales mutex

Una vez calculados los *mútex*, pasamos a la fase de extracción de la solución. Lo primero es comprobar que las metas (*¬garbage, dinner*) sean *mútex* entre si. Si nos fijamos, no tenemos ningún *mútex* entre los predicados de la meta. No obstante, tenemos dos problemas al realizar la búsqueda *hacia-atrás*:

* Para obtener *¬garbage* se puede obtener o bien a través de *carry()* o bien por *dolly()*.
  + Si vamos por la acción *carry()* vemos que es *mútex* con *cook()*, que tiene como efecto *dinner* (siendo esta acción la única para obtener como efecto *dinner*).
  + Si vamos por la acción *dolly()* vemos que es *mútex* con *wrap()* que tiene como efecto *present* (siendo esta acción la única para obtener como efecto *present*).

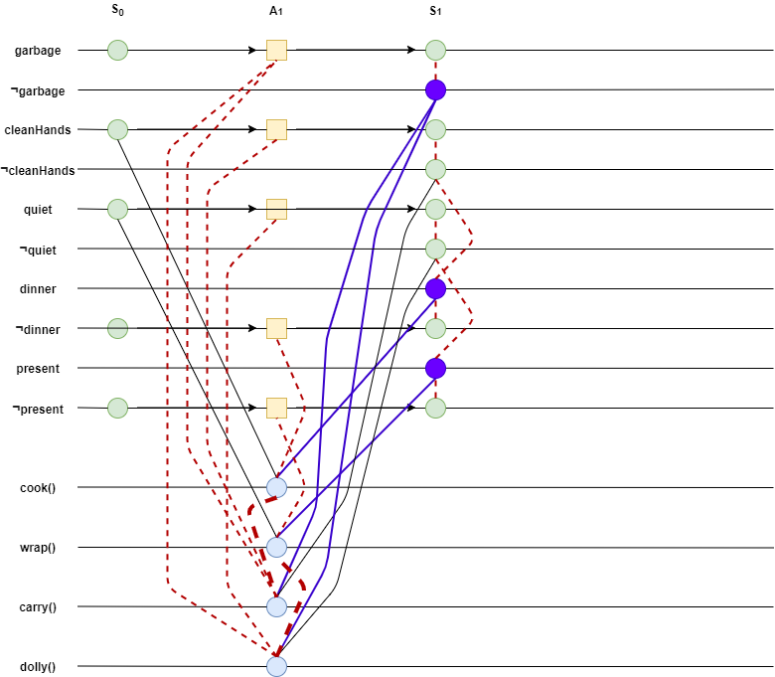


Ilustración 11. mutex entre los predicados de la meta

En conclusión, en caso de que exista una solución, esta no es alcanzable desde el nivel de estado 1. Por tanto, tenemos que volver a expandir el grafo.

A continuación, generamos un nuevo nivel tanto de acción como de proposición.

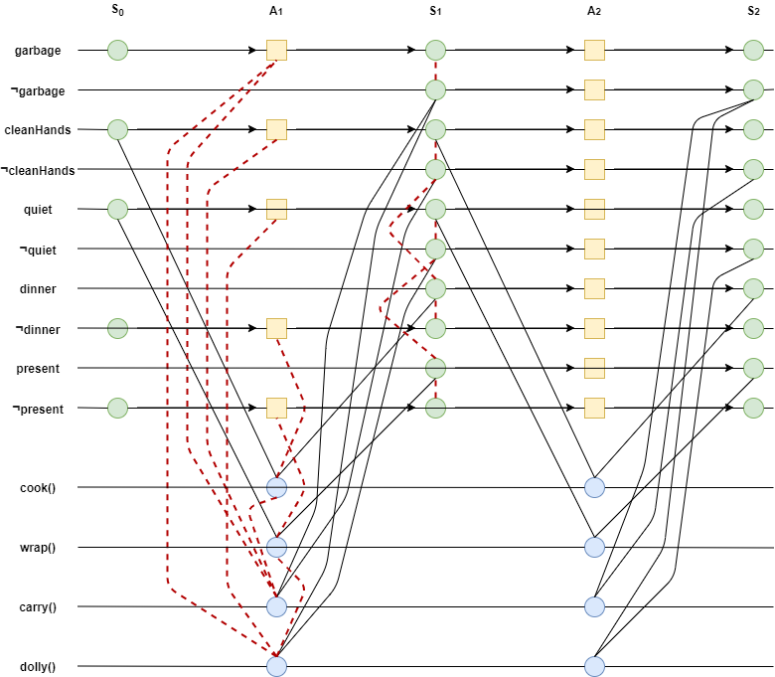


Ilustración 12. Siguiente nivel de acción y de proposición (A2 y S2)

A continuación, vemos que en el nivel de estado 2 tenemos los predicados del nodo meta. Por tanto, el siguiente paso será calcular los *mútex*.

* **Efectos inconsistentes**:
  + Entre la acción *no-op* *garbage* y *carry()*, ya que genera como efecto *¬garbage*.
  + Entre la acción *no-op* *garbage* y *dolly()*, ya que genera como efecto *¬garbage*.
  + Entre la acción *no-op* *cleanHands* y *carry()*, ya que genera como efecto *¬cleanHands*.
  + Entre la acción *no-op* *quiet* y *dolly()*, ya que genera como efecto *¬quiet*.
  + Entre la acción *no-op* *¬dinner* y *cook()*, ya que genera como efecto *dinner*.
  + Entre la acción *no-op* *¬present* y *wrap()*, ya que genera como efecto *present*.
* **Interferencias**:
  + Entre la acción *dolly()*, que tiene como efecto *¬quiet*, y *wrap()* cuya precondición es *quiet*.
  + Entre la acción *carry()*, que tiene como efecto *¬cleanHands*, y *cook()* cuya precondición es *cleanHands*.
* **Necesidades que entran en competencia**: realmente habría pero son contradicciones del nivel 1, de modo que ya se han añadido.
* **Contradicciones**:
  + Entre *garbage* y *¬garbage*.
  + Entre *cleanHands* y *¬cleanHands*.
  + Entre *quiet* y *¬quiet*.
  + Entre *dinner* y *¬dinner*.
  + Entre *present* y *¬present*.
* **Soporte de inconsistencia**:
  + Ya no hay *mútex e*ntre *present* y *¬quiet*. A modo de ejemplo, para obtener el literal *present* se puede ir a través de la acción *no-op* *present* y evitar la acción *wrap()*.
  + Ya no hay *mútex* entre *¬cleanHand* y *dinner*. A modo de ejemplo, para obtener el literal *¬cleanHand* se puede ir a través de la acción *no-op* *¬cleanHand*, evitando la acción *carry()*.

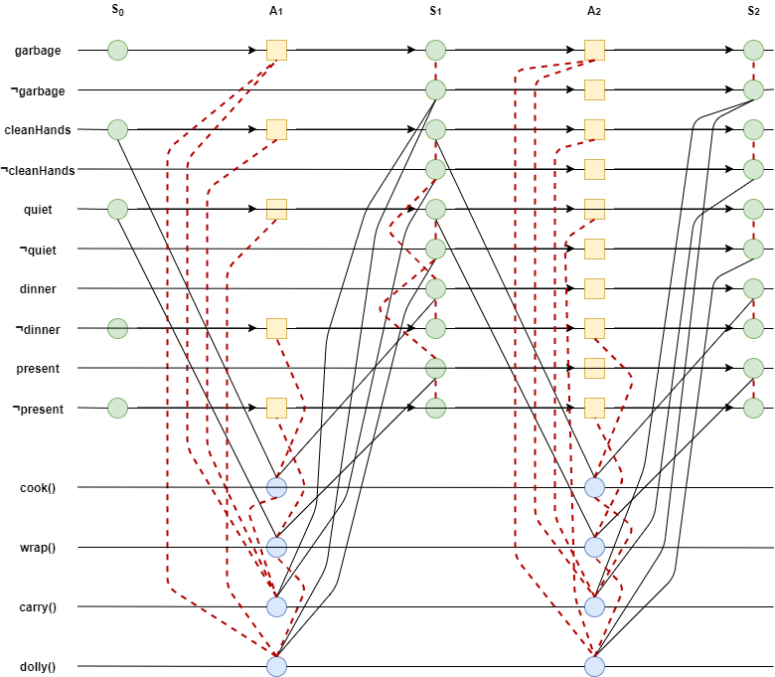


Ilustración 13. Acciones y literales mutex

Una vez calculados los *mútex*, pasamos a la fase de extracción de la solución. Lo primero es comprobar que las metas (*¬garbage, dinner*) sean *mútex* entre si. Si nos fijamos, no tenemos ningún *mútex* entre los predicados de la meta. Por otro lado, tenemos varias formas de pasar del nivel 2 al nivel 1:

* *¬garbage* se puede alcanzar bien desde la acción *carry()*, la *no-op* *¬garbage* y la acción *dolly()*.
* *dinner* se puede alcanzar bien desde la acción *cook()* o bien desde la acción *no-op* *dinner*.
* *present* se puede alcanzar bien desde la acción *wrap()* o bien desde la acción *no-op* *present*.

Ahora si podemos volver hacia atrás desde el nivel 2 hasta el nivel 1, tal y como podemos observar en la siguiente captura:

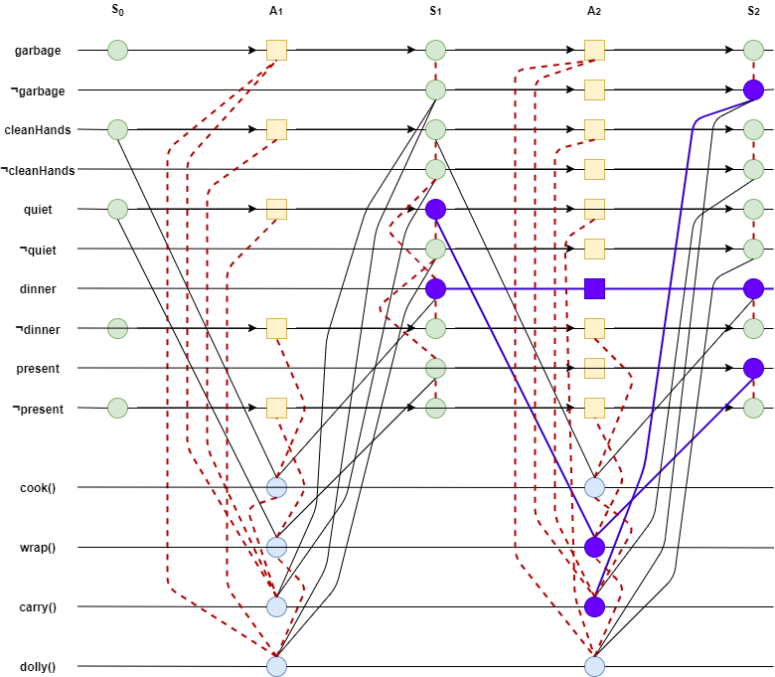


Ilustración 14. Camino a S1 desde S2

Desde el estado *S1* se puede volver a *S0*, desde los literales *dinner* y *quiet*, sin interferir en ningún *mútex*.

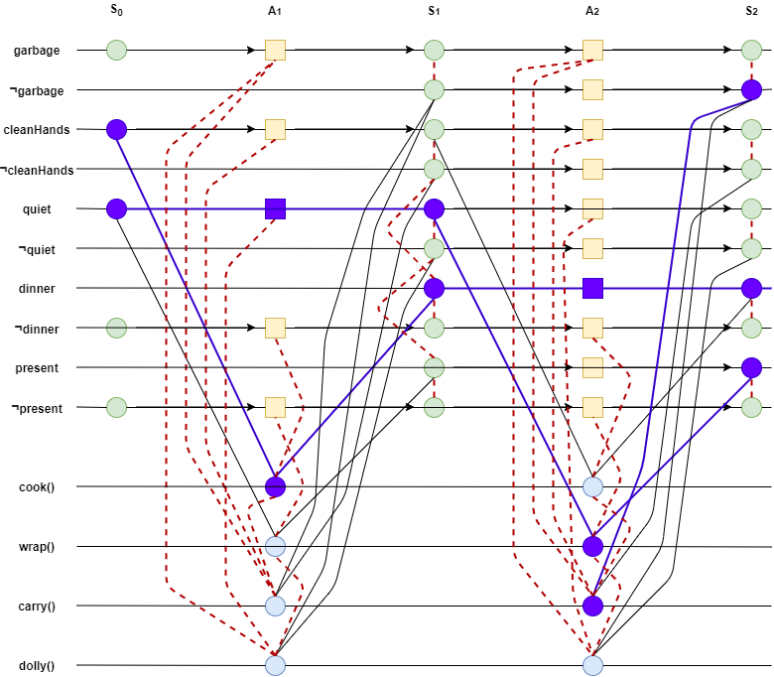


Ilustración 15. Camino desde S1 a S0

Como podemos comprobar, obtenemos un plan en dos pasos, en el cual se realiza la acción *cook()* en el primer paso mientas que en el segundo se ejecutan de forma paralela las acciones *wrap()* y *carry()*.