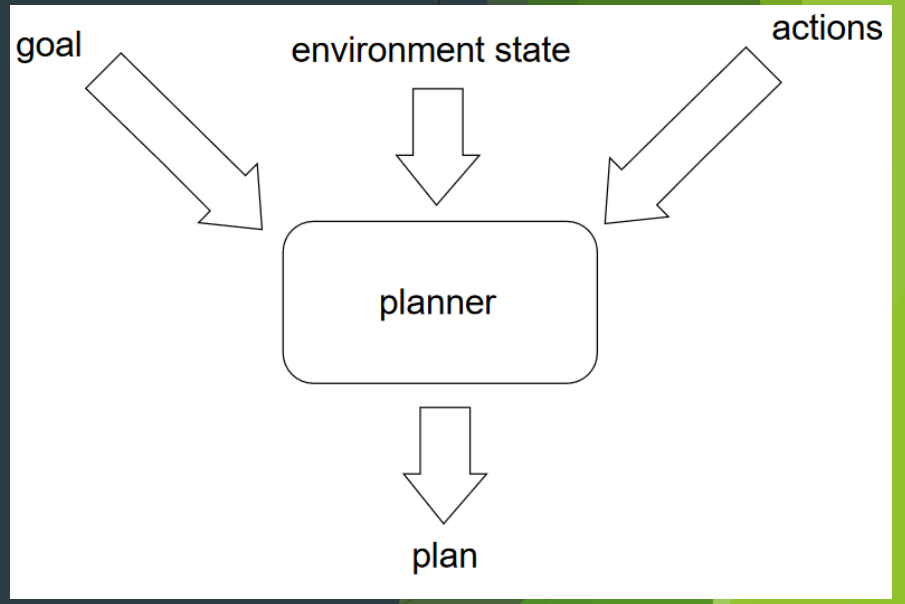
Técnicas de Inteligencia Artificial

Tema 6

Planificación clásica

Introducción

- La planificación es el diseño del curso de una acción para alcanzar un objetivo concreto.
- La idea básica consiste en tener un sistema de planificación:
  - Representación del objetivo/intención a conseguir.
  - Representación de las acciones que puede realizar.
  - Representación del entorno.
  - Generar un plan para alcanzar el objetivo.
- Dados los problemas con la búsqueda y el uso de lógica simple en problemas complejos, se buscaron formas de conseguir una representación más factorizada.
  - STRIPS. Stanford Research Institute Problem Solver.
  - PDDL. Similar a STRIPS, pero pudiendo usar literales negativos en precondiciones y objetivos.



Representación

- Para planificar, es necesario representar:
  - El objetivo a ser conseguido.
  - El estado del entorno.
  - Acciones disponibles para el agente.
  - El propio plan.
- Para ello, se debe usar la lógica o un sistema muy parecido.
- Para ilustrarlo, el ejemplo más famoso es el de los bloques.
  - El objetivo es generar un plan para un brazo robótico para construir torres de bloques sobre una mesa en un determinado orden.

Representación

- Para representar el entorno, se necesita una **ontología**:
  - On(x, y). Objeto x encima de objeto y.
  - OnTable(x). Objeto x en la mesa.
  - Clear(x). Nada está encima del objeto x.
  - Holding(x). El brazo robótico está sujetando x.
- La representación del mundo de los bloques de la imagen:
  - Clear(A)
  - On(A,C)
  - OnTable(B)
  - Clear(B)
  - OnTable(C)
- Se utiliza la asunción de mundo cerrado.
  - Todo lo que no se indica se asume como falso.

Representación

- Un **objetivo** se representa como un conjunto de fórmulas.
  - Ejemplo: {OnTable(A),OnTable(B),OnTable(C)}
- Las acciones se representan de la siguiente forma (alguna de las cuales puede contener variables):
  - **Nombre**.
    - Puede tener argumentos.
  - **Lista de precondiciones**.
    - Lista de hechos que deben ser verdad para que la acción se ejecute.
  - **Lista de borrado**.
    - Lista de hechos que dejan de ser verdad cuando la acción se ejecute.
  - **Lista de adición**.
    - Lista de hechos que se convierten en verdad al ejecutar la acción.

Representación

- Ejemplo de acción: stack.
  - El brazo robótico coloca al objeto x que está sujetando encima del objeto y.
  - ArmEmpty es una abreviación para indicar que el brazo no está sujetando ningún objeto.
  - *Stack(x,y)*
    - *pre Clear(y) ^ Holding(x)*
    - *del Clear(y) ^ Holding(x)*
    - *add ArmEmpty ∧ On(x, y)*
- Podemos pensar que las variables se cuantifican universalmente.

Representación

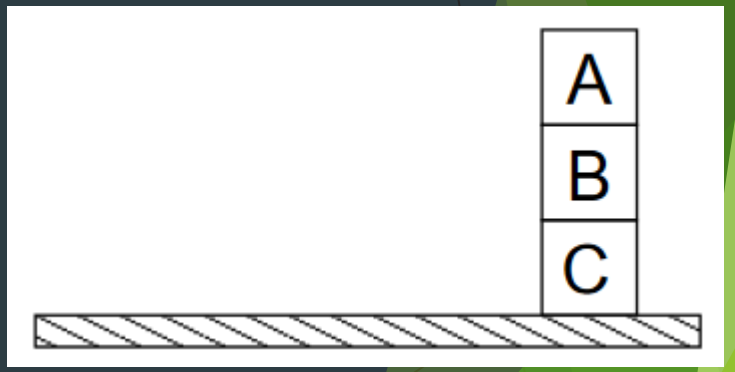
- Ejemplo de acción: unstack.
  - El brazo robótico toma un objeto x que está encima de otro objeto y.
  - *UnStack(x,y)*
    - *pre On(x, y) ∧ Clear(x) ∧ ArmEmpty*
    - *del On(x, y) ∧ ArmEmpty*
    - *add Holding(x) ∧ Clear(y)*
- Stack y UnStack son operaciones inversas.

Representación

- Ejemplo de acción: pickup.
  - El brazo robótico toma un objeto x que está encima de la mesa.
  - *Pickup(x)*
    - *pre Clear(x) ∧ OnTable(x) ∧ ArmEmpty*
    - *del OnTable(x) ∧ ArmEmpty*
    - *add Holding(x)*
- Ejemplo de acción: putdown.
  - El brazo robótico deja un objeto x sobre la mesa.
  - *PutDown(x)*
    - *pre Holding(x)*
    - *del Holding(x)*
    - *add Clear(x) ∧ OnTable(x) ∧ ArmEmpty*

Planificación

- Un plan es una secuencia de acciones, con variables reemplazadas por constantes.
- Para pasar de la figura 1 a la 2 necesitamos elaborar el siguiente plan:
  - Unstack(A)
  - Putdown(A)
  - Pickup(B)
  - Stack(B, C)
  - Pickup(A)
  - Stack(A, B)

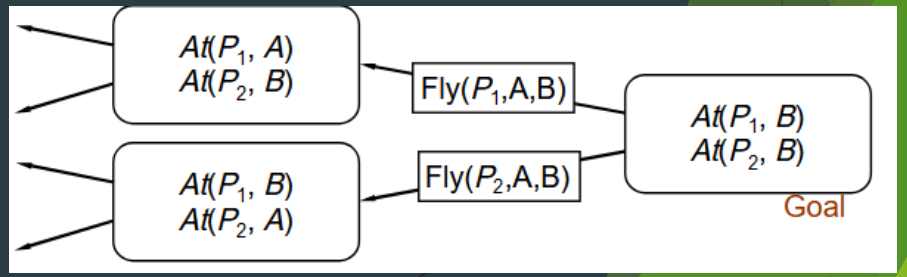


Planificador ingenuo

- En problemas reales, los planes contienen condiciones y bucles, pero los planificador más simples no pueden manejar esas estructuras, sino que crean **planes lineales**.
- La aproximación más simple consiste en un análisis medios-fines (**means-end analysis**).
  - Se comienza donde se quiere llegar (fin) y se aplican acciones (medios) para conseguir ese estado.
  - Implica encadenamiento regresivo (backward chaining) desde el objetivo al estado original.
  - Comienza encontrando una acción consistente con tener el objetivo como post-condición, y se asume que es la última acción del plan.
  - Entonces, se busca cuál sería el estado previo a la última acción, y se busca la acción que tiene este estado como post-condición.
  - Se realiza recursión hasta finalizar en el estado original (si se consigue).
  - Una acción a se puede ejecutar en el estado s si s cumple las precondiciones pre(a) de a.
    - s |= pre(a)
      - Esto es verdad si todos los literales positivos de pre(a) están en s, y todos los literales negativos de pre(a) no.

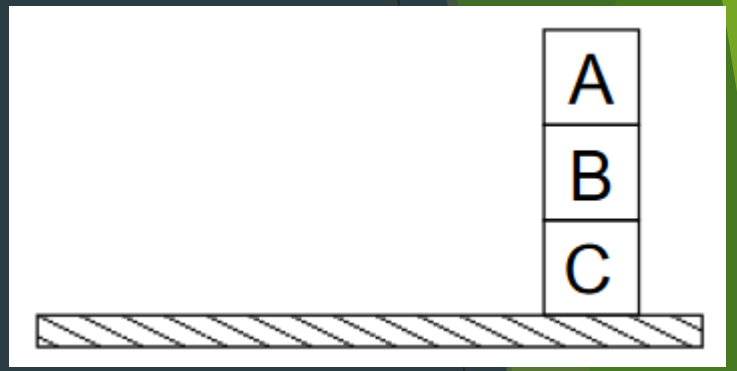
Planificador ingenuo

```
function plan(
        d : WorldDesc,          // environment state
        g : Goal,               // current goal
        p : Plan,               // plan so far
        A : set of actions      // actions available)
1.  if d ⊨ g then
2.          return p
3.  else
4.          choose some a in A with g ⊨ add(a)
5.          set g = (g − add(a)) ∪ pre(a)
6.          append a to p
7.          return plan(d, g, p, A)
```



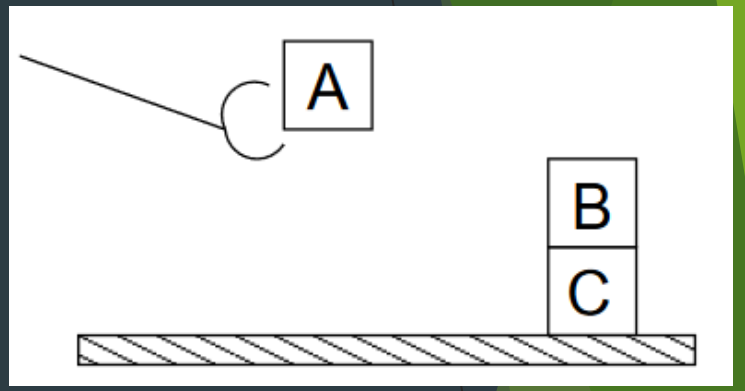
Planificador ingenuo

- En el ejemplo, comenzamos en el estado objetivo:
  - *On(A, B)*
  - *On(B, C)*
  - *OnTable(C)*
  - *ArmEmpty*
- Entonces, elegimos una acción que tiene una lista de adición que es satisfecha por este estado:
  - *Stack(A,B)*
- Para conseguir el estado antes de esta acción, se elimina la lista de adición y se añaden las precondiciones.



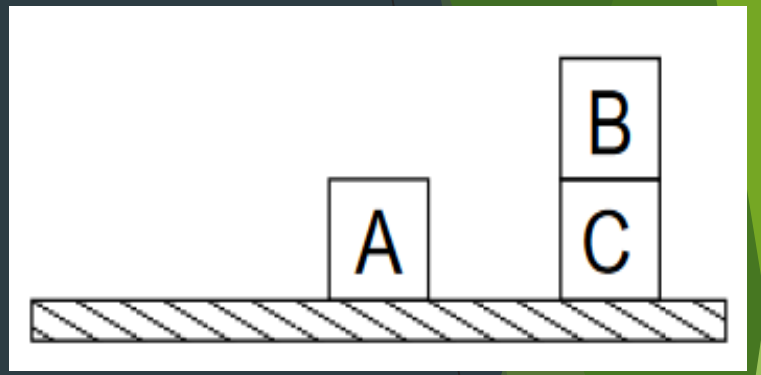
Planificador ingenuo

- El nuevo estado conseguido:
  - *Clear(B)*
  - *On(B, C)*
  - *OnTable(C)*
  - *Holding(A)*
- Seleccionar la acción previa en el plan, que ahora es aquella satisfecha por el estado anterior:
  - *Pickup(A)*



Planificador ingenuo

- El nuevo estado conseguido:
  - *Clear(B)*
  - *On(B, C)*
  - *OnTable(C)*
  - *OnTable(A)*
  - *ArmEmpty*
- Se continúa realizando la búsqueda hacia atrás hasta conseguir el estado inicial.



Planificador ingenuo

- El algoritmo no garantiza encontrar un plan que satisfaga el objetivo.
- Sin embargo, el algoritmo es sólido, si encuentra un plan éste es correcto.
- Problemas:
  - Objetivos negativos.
    - ¿Cómo indicar que se quiere cualquier torres de bloques donde B no esté en la mesa sin enumerar todas las torres que se podrían construir?
  - Objetivos de mantenimiento.
    - ¿Cómo indicar que siempre se deben mantener al menos dos bloques en la mesa?
  - Condicionales y bucles.
  - Espacio de búsqueda exponencial.
    - El espacio de búsqueda puede crecer enormemente y hacer impracticable la búsqueda.
  - Pruebas de consecuencias lógicas.
    - Dependiendo de la lógica subyacente, el problema de decidir la validez de una fórmula puede variar de ser trivial a imposible.

Problemas con el espacio de estados

- Otro problema con el espacio de estados consiste en cómo seleccionar una acción.
  - En el caso ingenuo, asumimos que podemos elegir una buena acción, aunque en general esto no suele ser una buena táctica.
  - **Podemos aplicar heurísticas y usar A*** como si fuese un problema de búsqueda.
    - La diferencia con los problemas de búsqueda es que los operadores de búsqueda factorizados permiten utilizar heurísticas independientes del dominio que funcionen para cualquier problema de planificación.
  - **Podemos ignorar precondiciones**.
    - Podemos establecer heurísticas que relajen las restricciones del problema asegurándonos de que sean admisibles.
  - **Podemos ignorar listas de borrado**.
    - Ninguna acción deshace el efecto de otra acción.

Problemas con el espacio de estados

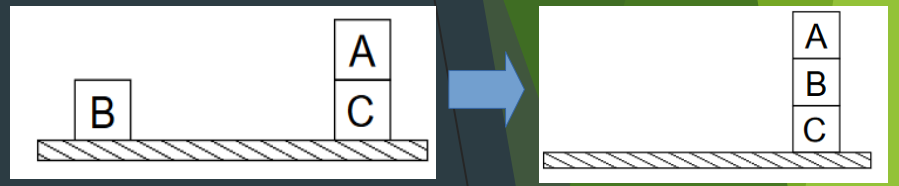
- Otro problema con el espacio de estados consiste en cómo seleccionar una acción.
  - A pesar de aplicar heurísticas, el espacio de búsqueda puede ser todavía demasiado grande.
- Se pueden realizar abstracciones de estados.
  - Planificación en un espacio en el que ciertos estados se agrupen juntos.
- Se puede simplificar el problema añadiendo nuevas restricciones que acoten mucho más el espacio de búsqueda, encontrar una solución y después expandirla al problema más grande, mediante composición de soluciones.
  - No es óptimo pero es una solución más sencilla.
- Una idea clave definiendo heurísticas es la descomposición del problema en partes, resolver cada subproblema de forma independiente y combinar sus partes.
  - La asunción de la independencia de subobjetivos es que el coste de resolver una conjunción de subobjetivos es aproximadamente la suma de resolver los costes de cada uno de manera independiente.

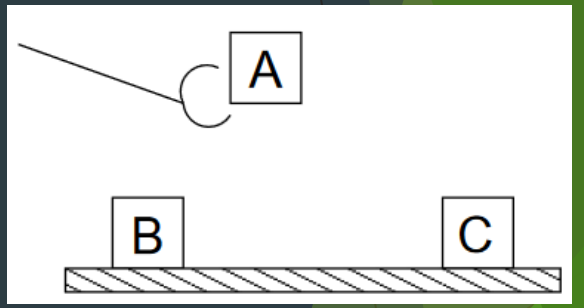
El problema del marco

- El problema del marco es un problema general que aparece al representar las propiedades de las acciones.
  - ¿Cómo saber exactamente qué cambia como resultado de una acción?
- Una solución consiste en escribir axiomas de marco.
  - Un axioma de marco determina que un estado permanece inalterable en todas las situaciones que resultan de una acción determinada.
  - No resultan viables para problemas reales, ya que habría que hacer un axioma de marco para cada variable y cada posible acción.
- STRIPS resuelve el problema del marco asumiendo que todo lo que no se indica explícitamente que ha cambiado permanece sin cambios.

Anomalía de Sussman

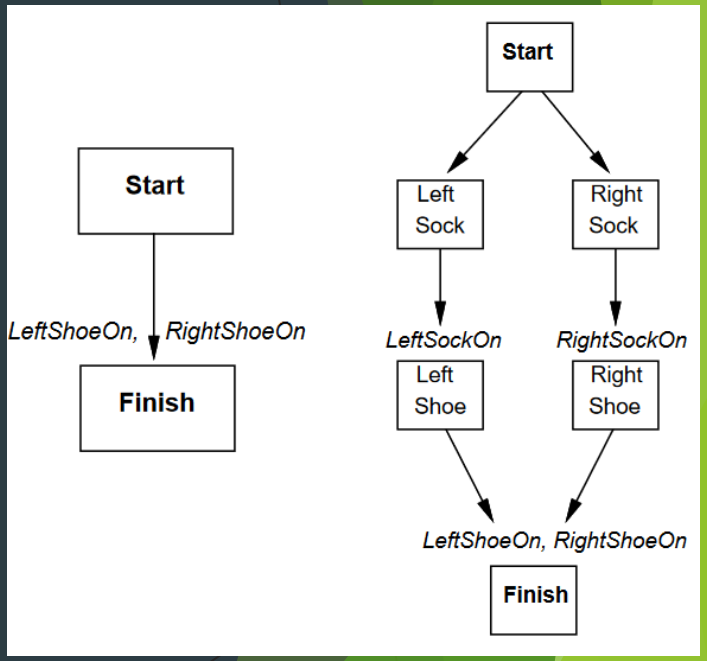
- Si queremos pasar del estado inicial al estado objetivo, claramente la primera operación es desapilar A de C.
- Una vez realizada la operación de desapilar, si el planificador considera que el estado final es tener:
  - *On(A, B)*
  - *On(B, C)*
- Entonces, realizar el siguiente movimiento Stack(A,B) puede parecer que está más cerca del objetivo.
  - Sin embargo, esto no está más cerca del objetivo real.
  - Supone un camino más largo al objetivo que requiere volver atrás al estado anterior.
- Esta anomalía es un gran problema de los planificadores lineales.



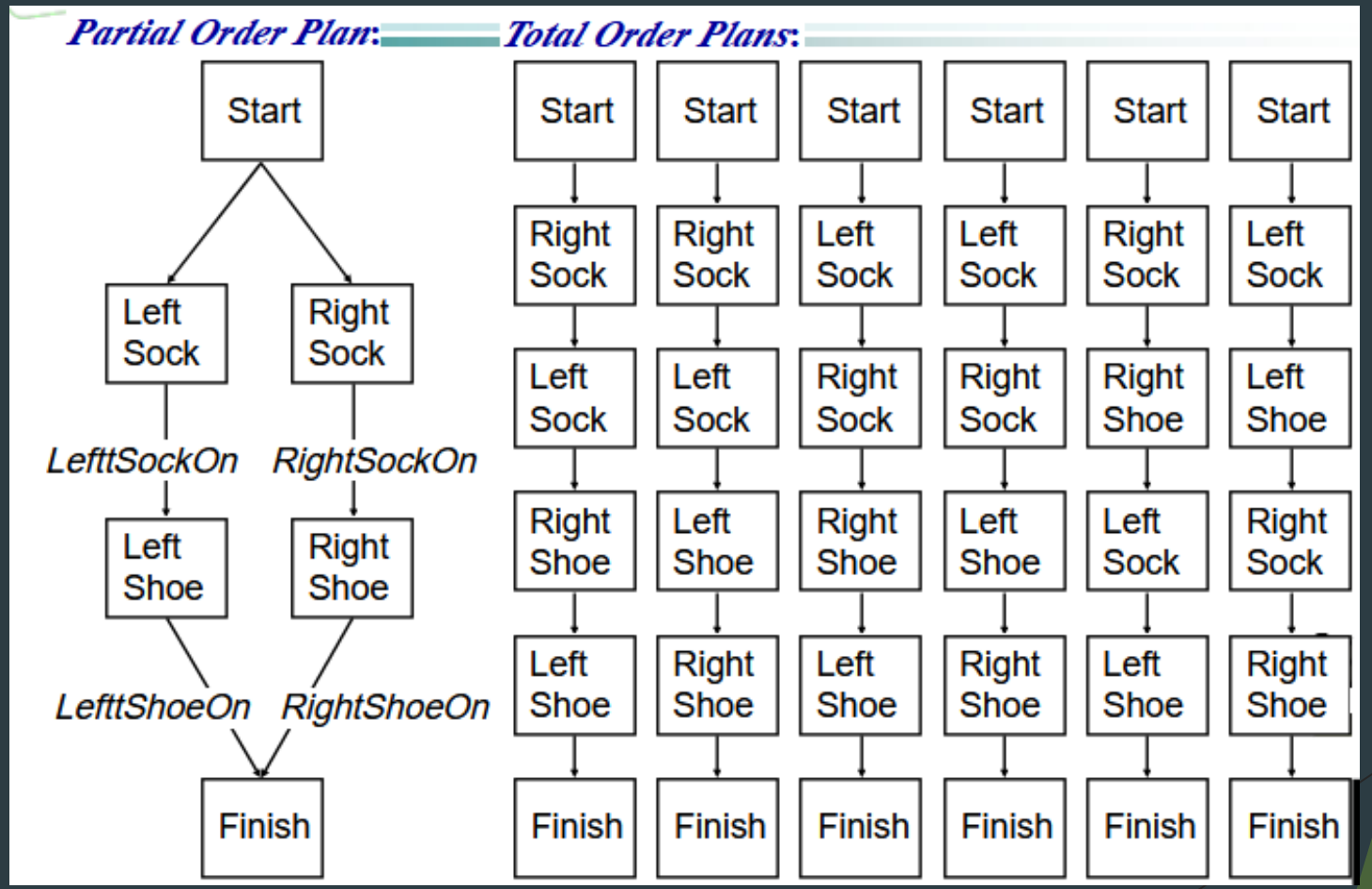


Planificación parcialmente ordenada

- La solución a la anomalía consiste en usar una **planificación parcialmente ordenada**.
  - Esto nos permite comprobar antes de añadir una acción al plan si esta no desbarata el resto del plan.
  - El problema con el proceso recursivo usado por STRIPS es que no conoce cuál es el resto del plan.
  - Se requiere una nueva representación para **planes parcialmente ordenados**.
- Un **plan parcialmente ordenado** tiene las siguientes partes:
  - Una colección parcialmente ordenada de pasos:
    - **Paso inicial** donde con la descripción del estado inicial y su efecto.
    - **Paso final** con la descripción del objetivo y su precondición.
    - **Enlaces causales** desde la salida de un paso a la precondición de otro.
    - **Ordenación temporal** entre distintos pares de pasos.

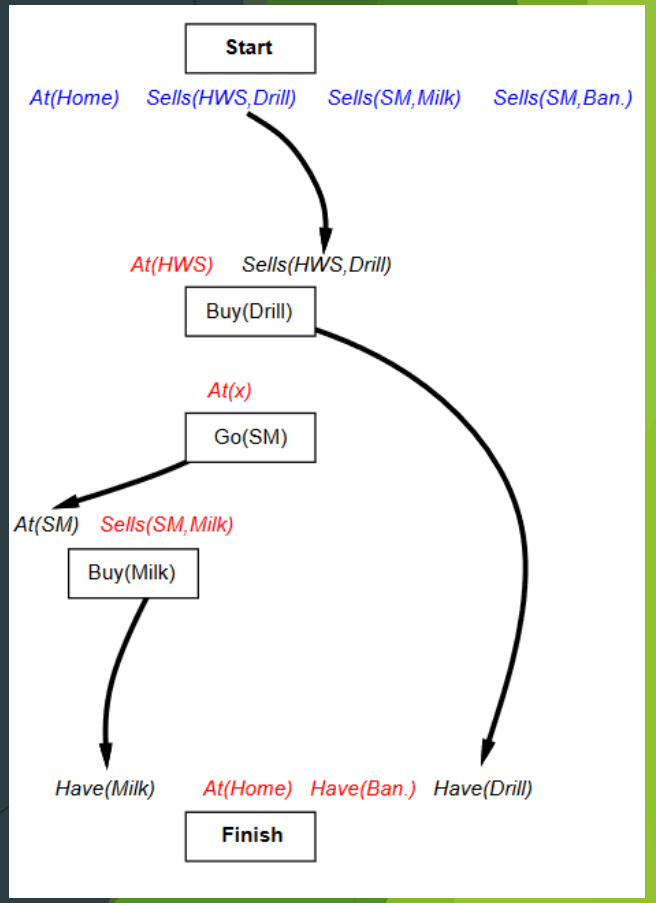


Planificación parcialmente ordenada



Planificación parcialmente ordenada

- Un **plan parcialmente ordenado** tiene las siguientes partes:
  - **Condición abierta**.
    - Precondición de un paso no vinculado causalmente todavía.
  - Un plan está completo si y solo si cada precondición se cumple.
  - Una precondición se cumple si y solo si es el efecto de un paso anterior y ningún posible paso intermedio puede deshacerlo.
- **Para construir un plan**:
  - Empezamos con los estados inicial y final.
  - Añadimos acciones que consigan:
    - O bien las precondiciones del estado final.
    - O bien las precondiciones de acciones que ya hemos añadido.
  - Vincular las precondiciones con las postcondiciones que coincidan.

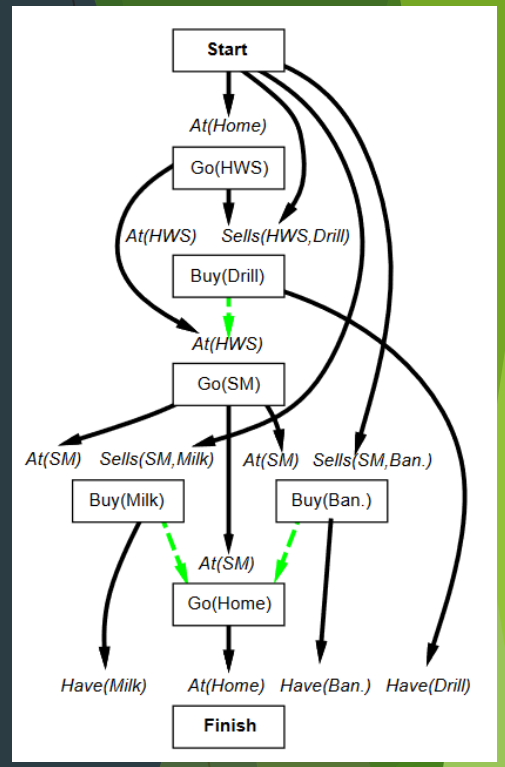


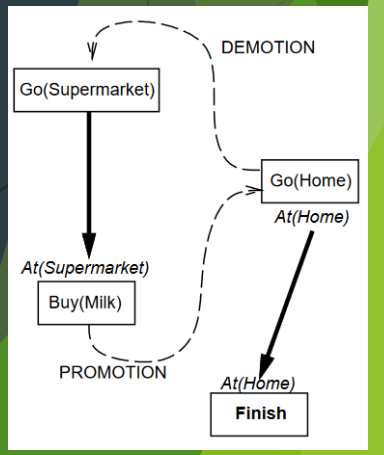
Planificación parcialmente ordenada

1. if $d \models g$ then
2. return p
3. else
4. choose some a in A
4a. if $no\_clobber(a, rest\_of\_plan)$
5. set $g = (g - add(a)) \cup pre(a)$
6. append a to p
7. return $plan(d, g, p, A)$

Planificación parcialmente ordenada

- Algunas acciones pueden introducir restricciones de orden en otras acciones mediante postcondiciones que conviertan las precondiciones de esas otras acciones en falsas.
  - Esta circunstancia fuerza a ordenar algunas acciones respecto a otras.
  - Los enlaces causales entre acciones nos permiten detectar el arruinamiento (**clobbering**) del camino.
    - Estos nos sugieren la ordenación de las etapas.
    - Un arruinador (o **clobberer**) es una potencial etapa intermedia que destruye la condición conseguida por un enlace causal.
      - Ejemplo: Go(Home) arruina At(Supermarket).
      - Degradación: anteponer a Go(Supermarket).
      - Promoción: poner tras Comprar(Leche)





Planificación parcialmente ordenada

- El **proceso de planificación** funciona de la siguiente forma:
  - Operadores de planes parciales:
    - Añadir un enlace desde una acción existente para abrir una condición.
    - Añadir una etapa para rellenar la condición abierta.
    - Ordenar una etapa con respecto a otro para eliminar posibles conflictos.
  - Gradualmente, evolucionar desde planes incompletos y vagos hasta planes completos y correctos.
  - Realizar una operación de backtracking si una condición abierta es inalcanzable o si hay un conflicto que no se puede solucionar.

Planificación parcialmente ordenada

```
function POP(initial, goal, operators) returns plan
    plan ← MAKE-MINIMAL-PLAN(initial, goal)
    loop do
        if SOLUTION?(plan) then return plan
        S_need, c ← SELECT-SUBGOAL(plan)
        CHOOSE-OPERATOR(plan, operators, S_need, c)
        RESOLVE-THREATS(plan)
    end
```

```
function SELECT-SUBGOAL(plan) returns S_need, c
    pick a plan step S_need from STEPS(plan)
        with a precondition c that has not been achieved
    return S_need, c
```

Planificación parcialmente ordenada

procedure CHOOSE-OPERATOR($plan, operators, S_{need}, c$)

  choose a step S_{add} from *operators* or STEPS(*plan*) that has c as an effect

  if there is no such step **then fail**

  add the causal link $S_{add} \xrightarrow{c} S_{need}$ to LINKS(*plan*)

  add the ordering constraint $S_{add} \prec S_{need}$ to ORDERINGS(*plan*)

  if S_{add} is a newly added step from *operators* **then**

    add S_{add} to STEPS(*plan*)

    add $Start \prec S_{add} \prec Finish$ to ORDERINGS(*plan*)

Planificación parcialmente ordenada

procedure RESOLVE-THREATS(*plan*)

  for each S_{threat} that threatens a link $S_i \xrightarrow{c} S_j$ in LINKS(*plan*) **do**

    choose either

      Demotion: Add $S_{threat} \prec S_i$ to ORDERINGS(*plan*)

      Promotion: Add $S_j \prec S_{threat}$ to ORDERINGS(*plan*)

    if not CONSISTENT(*plan*) **then fail**

  end

Planificación parcialmente ordenada

- **Propiedades de la función POP**:
  - Algoritmo no determinista.
    - Realiza backtracking sobre puntos de elección fallidos:
      - Elección de S_{add} para conseguir S_{need}.
      - Elección de degradación o promoción para el clobberer.
      - La selección de S_{need} es irrevocable.
  - El algoritmo es robusto, completo y sistemático (no genera repeticiones).
  - Extensiones para disyunciones, cuantificadores universales, negaciones y condicionales.
  - Se puede hace eficiente con buenas heurísticas derivadas de la descripción del problema.
  - Particularmente bueno con problemas con muchos subobjetivos poco relacionados.

Solución PDDL del problema de la rueda de repuesto

- Ejemplo con el problema de la rueda de repuesto (Spare Tire Problem)

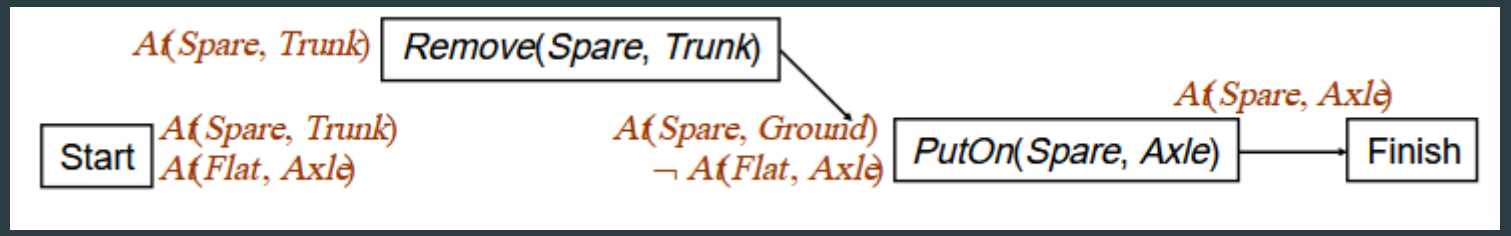
Init($At(Flat, Axle) \wedge At(Spare, trunk)$)
Goal($At(Spare, Axle)$)
Action(Remove($Spare, Trunk$)
PRECOND: $At(Spare, Trunk)$
EFFECT: $\neg At(Spare, Trunk) \wedge At(Spare, Ground)$)
Action(Remove($Flat, Axle$)
PRECOND: $At(Flat, Axle)$
EFFECT: $\neg At(Flat, Axle) \wedge At(Flat, Ground)$)
Action(PutOn($Spare, Axle$)
PRECOND: $At(Spare, Groundp) \wedge \neg At(Flat, Axle)$
EFFECT: $At(Spare, Axle) \wedge \neg At(Spare, Ground)$)
Action(LeaveOvernight
PRECOND:
EFFECT: $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, trunk) \wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle)$)

Solución PDDL del problema de la rueda de repuesto

- Se elige una precondición abierta: At(Spare, Axle)
- Solo se puede aplicar: PutOn(Spare, Axle)
- Se añade el enlace causal: $PutOn(Spare, Axle) \xrightarrow{At(Spare, Axle)} Finish$
- Se añade la restricción: PutOn(Spare, Axle) $\prec$ Finish

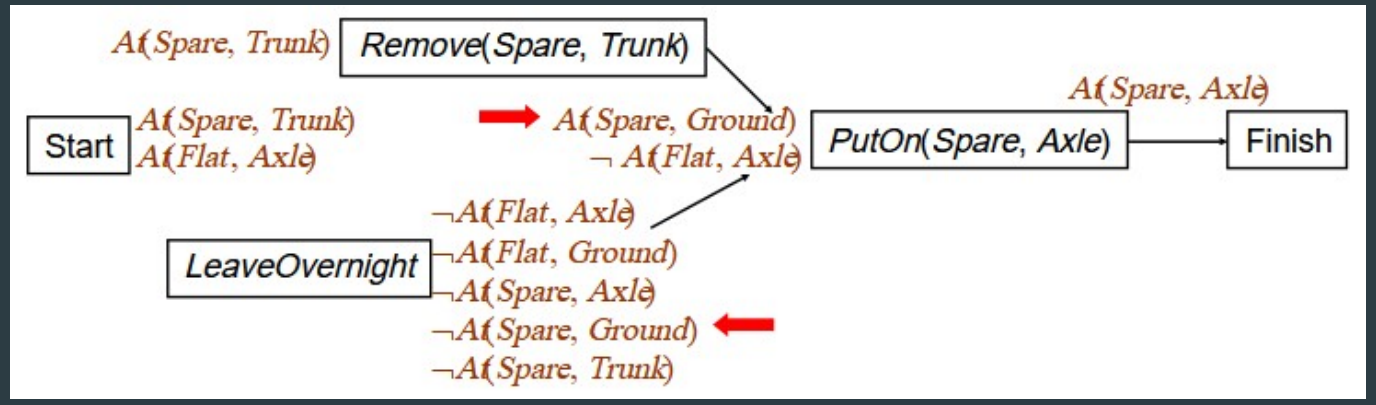
Solución PDDL del problema de la rueda de repuesto

- Se elige una precondición abierta: At(Spare, Ground)
- Solo se puede aplicar: Remove(Spare, Trunk)
- Se añade el enlace causal: $Remove(Spare, Trunk) \xrightarrow{At(Spare, Ground)} PutOn(Spare, Axle)$
- Se añade la restricción: Remove(Spare, Trunk) $\prec$ PutOn(Spare, Axle)



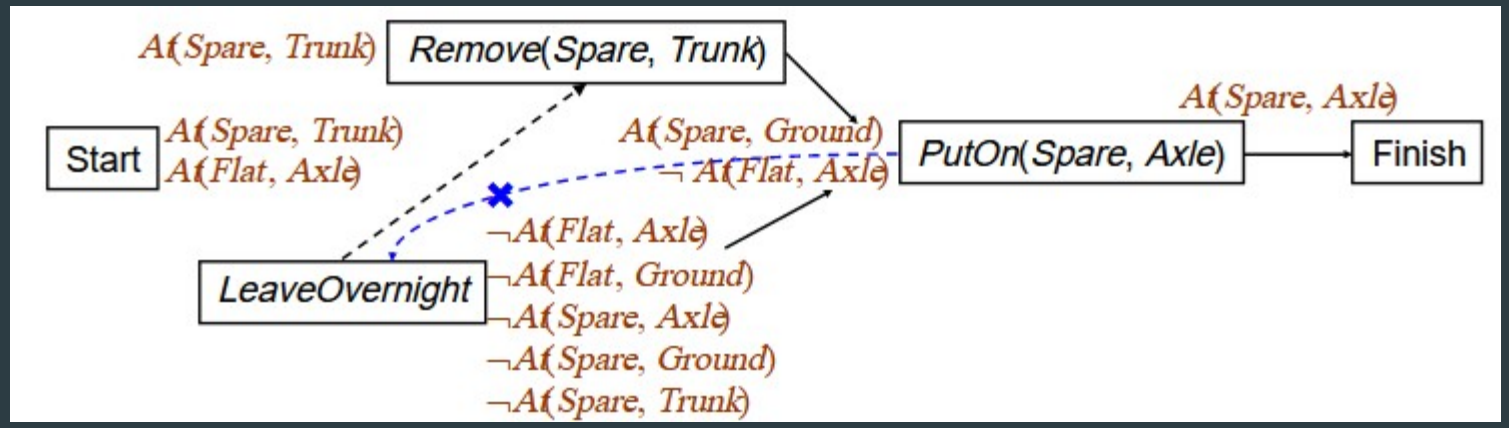
Solución PDDL del problema de la rueda de repuesto

- Se elige una precondición abierta: ¬At(Flat, Axle)
- Se puede aplicar: LeaveOverNight
- Conflicto con $Remove(Spare, Trunk) \xrightarrow{At(Spare, Ground)} PutOn(Spare, Axle)$



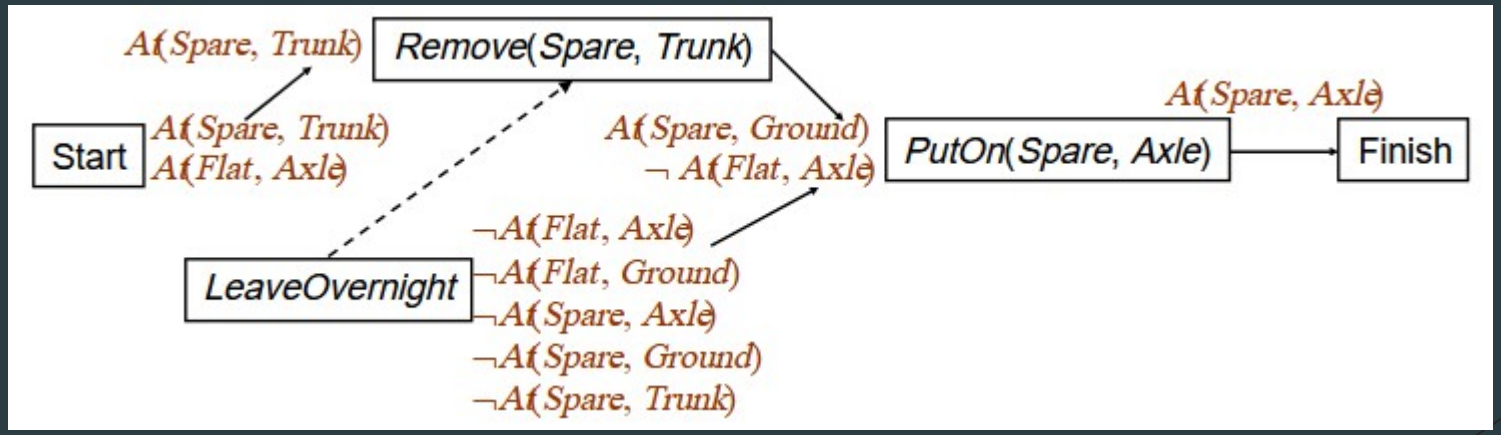
Solución PDDL del problema de la rueda de repuesto

- Para resolver, añadir la restricción: LeaveOverNight $\prec$ Remove(Spare, Trunk)
- Añadir el enlace causal: $LeaveOvernight \xrightarrow{\neg At(Spare,Ground)} PutOn(Spare, Axle)$



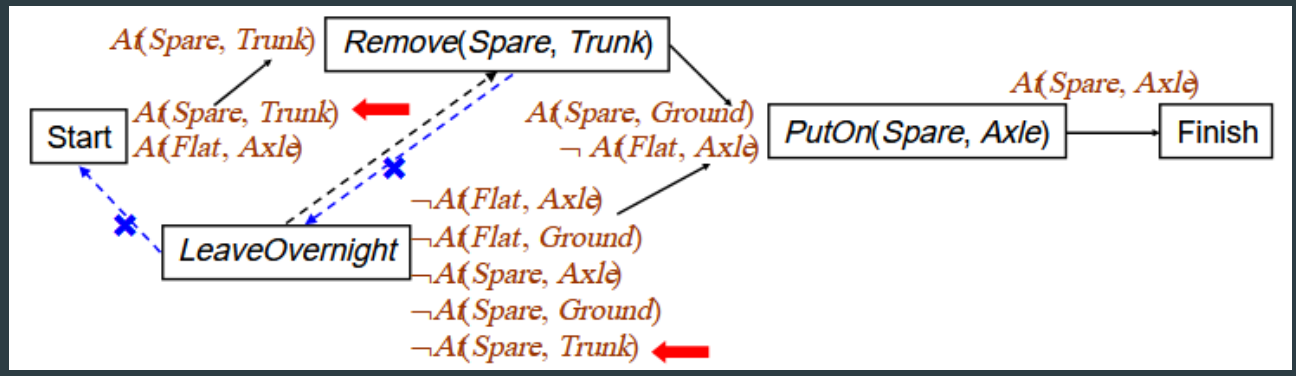
Solución PDDL del problema de la rueda de repuesto

- Se elige una precondición abierta: At(Spare, Trunk)
- Solo se puede aplicar: Start
- Se añade el enlace causal: $Start \xrightarrow{At(Spare,Trunk)} Remove(Spare,Trunk)$



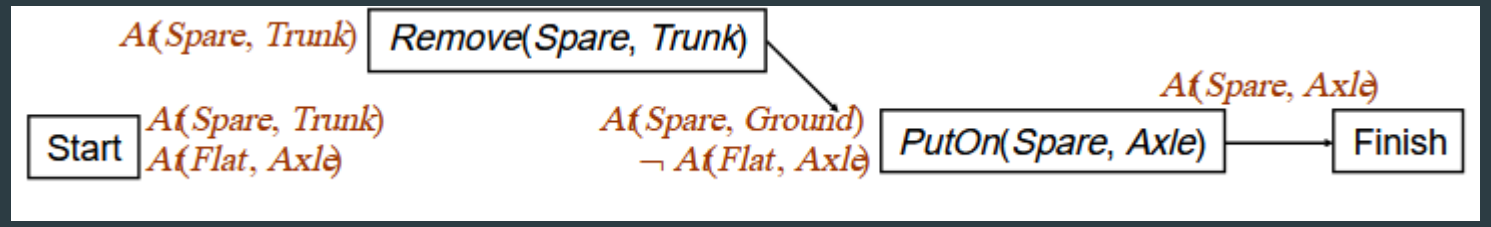
Solución PDDL del problema de la rueda de repuesto

- Conflicto del link causal con efecto At(Spare,Trunk) en LeaveOverNight
- No existe reordenación posible, así que se debe hacer backtracking.



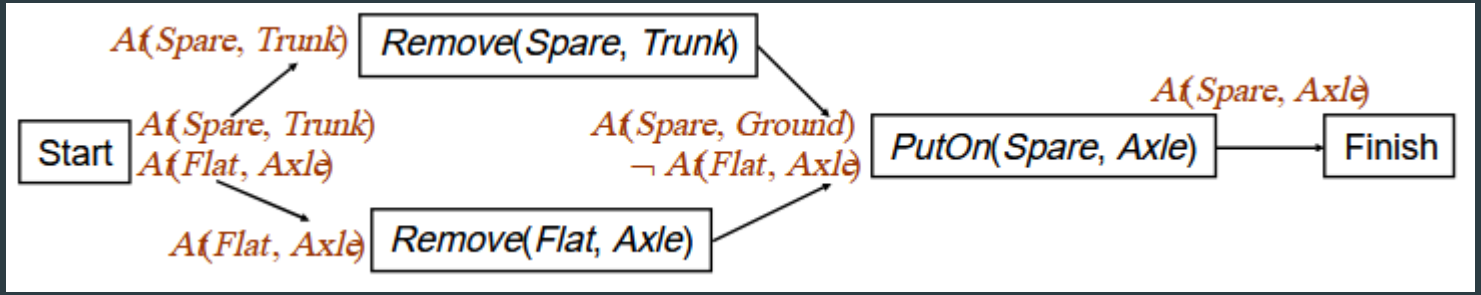
Solución PDDL del problema de la rueda de repuesto

- Eliminar: LeaveOverNight
- Eliminar el enlace causal: $LeaveOvernight \xrightarrow{\neg At(Spare,Ground)} PutOn(Spare, Axle)$



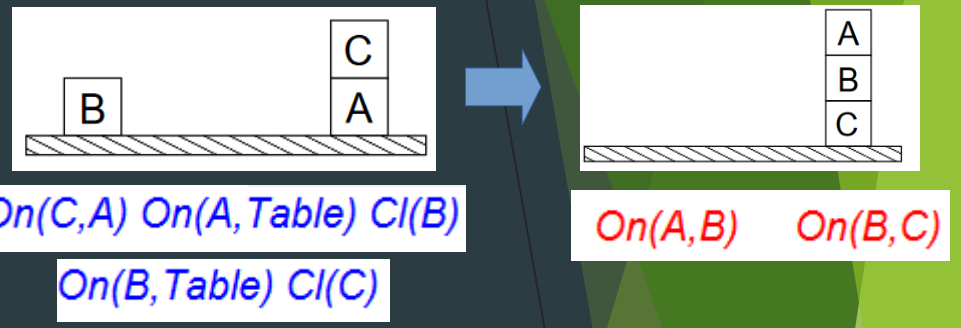
Solución PDDL del problema de la rueda de repuesto

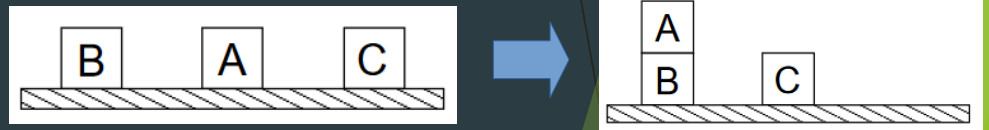
- Considerar de nuevo la precondición abierta: ¬At(Flat, Axle)
- Se puede aplicar: Remove(Flat, Axle)
- Elegir Start para conseguir At(Spare, Trunk) y At(Flat, Axle)
- Finalizar, se ha alcanzado el comienzo sin conflictos

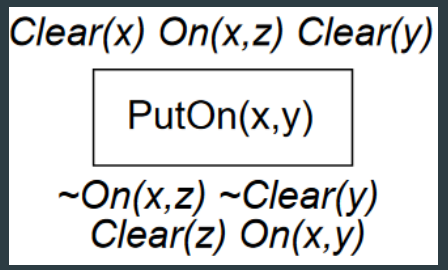


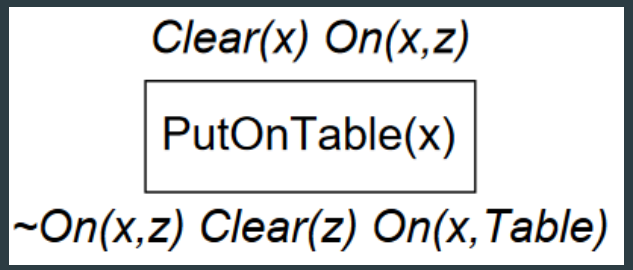
Otra versión de la anomalía de Sussman

- Otra versión de la anomalía de Sussman aparece al querer resolver el problema desde el inicio al objetivo mostrado en las figuras.
- En este caso, el problema surge una vez plantados los bloques en la mesa.
- Sin ninguna heurística adicional, apilar A sobre B parece un movimiento igual de bueno que apilar B sobre C.





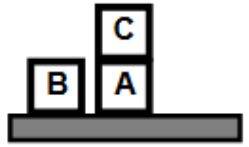




Otra versión de la anomalía de Sussman

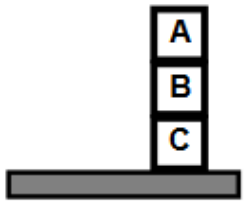
START

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

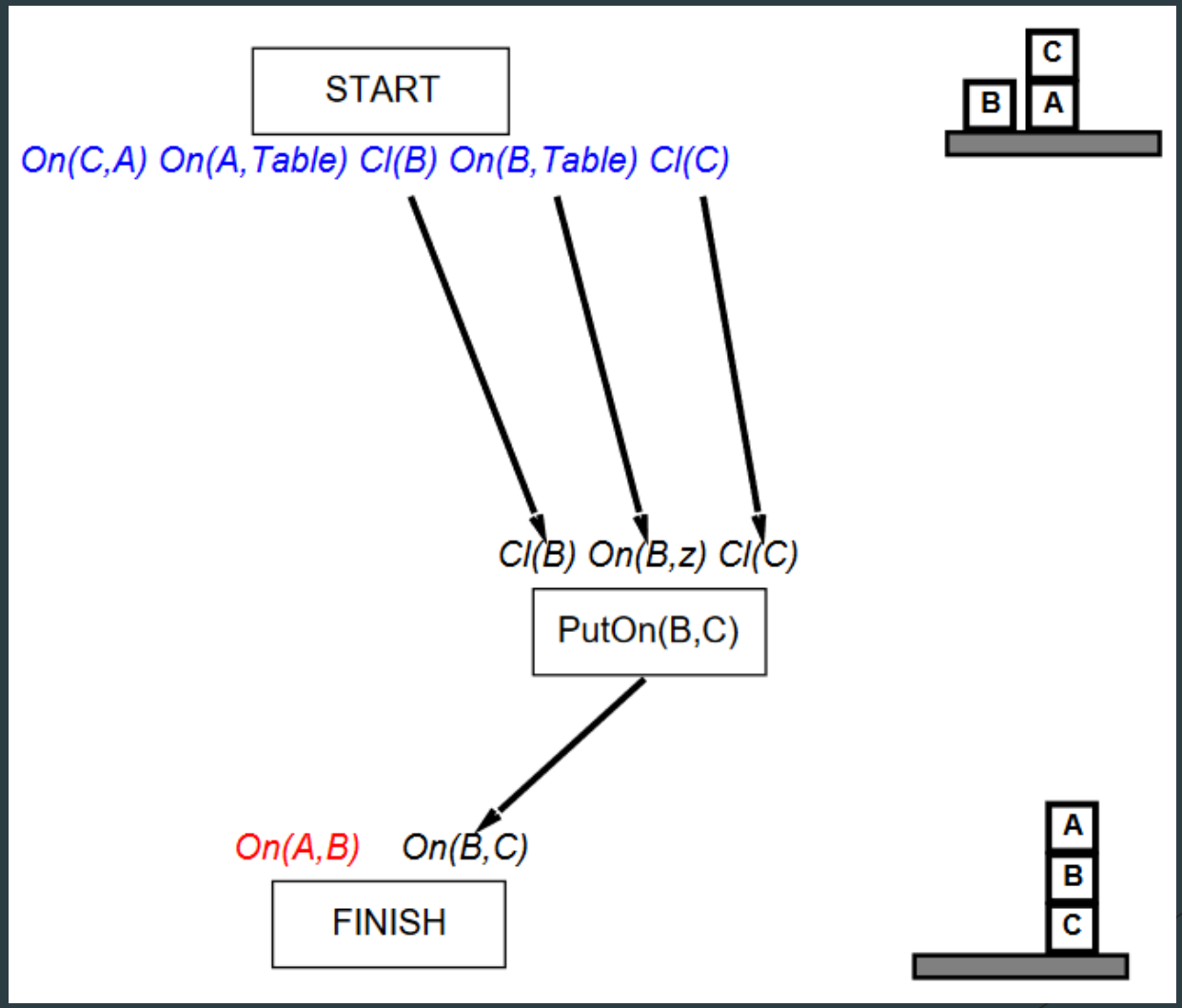


On(A,B) On(B,C)

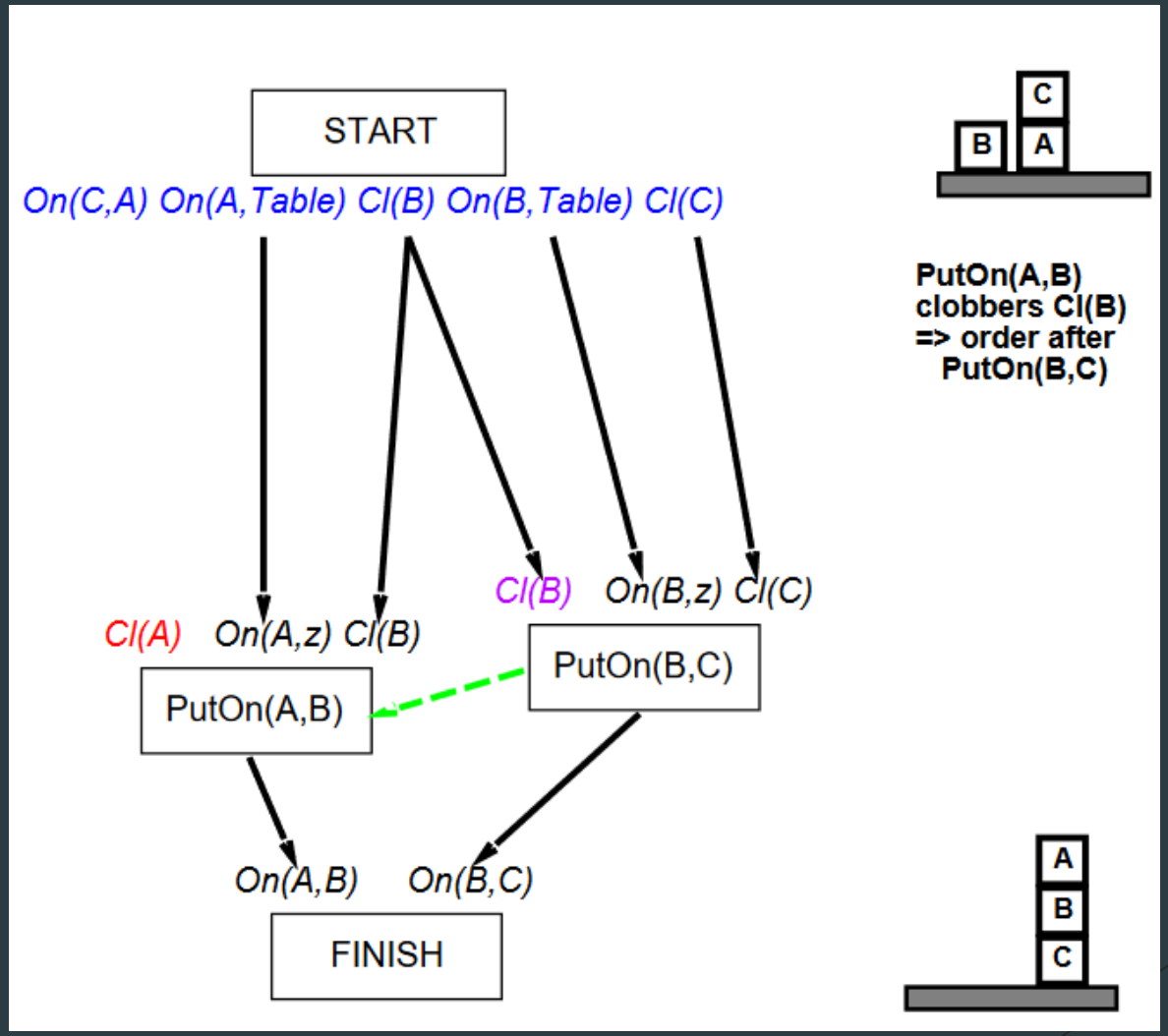
FINISH



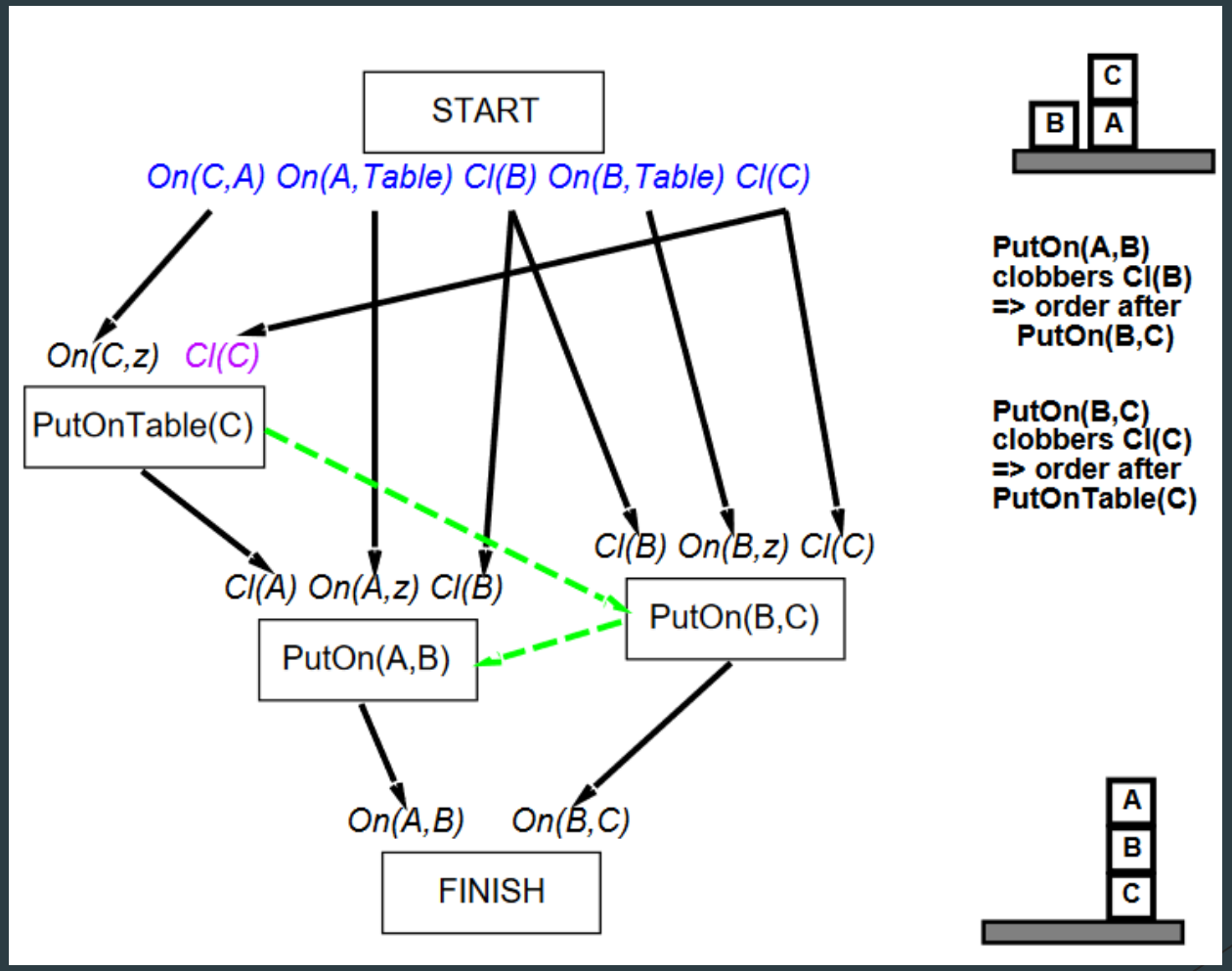
Otra versión de la anomalía de Sussman



Otra versión de la anomalía de Sussman



Otra versión de la anomalía de Sussman



Solución con PDDL

$Init(\ On(A, Table) \wedge On(B,Table) \wedge On(C,Table) \wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(A) \wedge Clear(B) \wedge Clear(C)\ )$

$Goal(\ On(A,B) \wedge On(B,C)\ )$

$Action(\ Move(b,x,y),$ PRECOND: $On(b,x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge (b \neq x) \wedge (b \neq y) \wedge (x \neq y)$

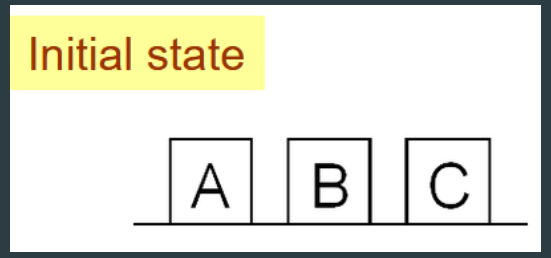
EFFECT: $On(b,y) \wedge Clear(x) \wedge \neg On(b,x) \wedge \neg Clear(y)\ )$

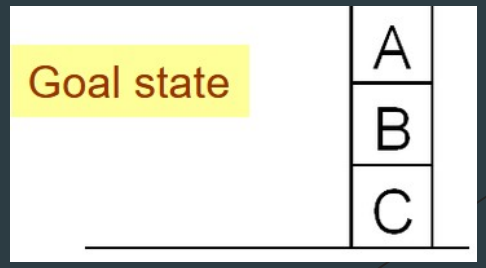
$Action(\ MoveFromTable(b,x),$ PRECOND: $On(b,Table) \wedge Clear(b) \wedge Clear(x) \wedge Block(b) \wedge (b \neq x)$

EFFECT: $On(b,x) \wedge \neg On(b,Table) \wedge \neg Clear(x)\ )$

$Action(\ MoveToTable(b,x),$ PRECOND: $On(b,x) \wedge Clear(b) \wedge Block(b) \wedge Block(x) \wedge (b \neq x)$

EFFECT: $On(b,Table) \wedge Clear(x) \wedge \neg On(b,x)\ )$





Solución con PDDL

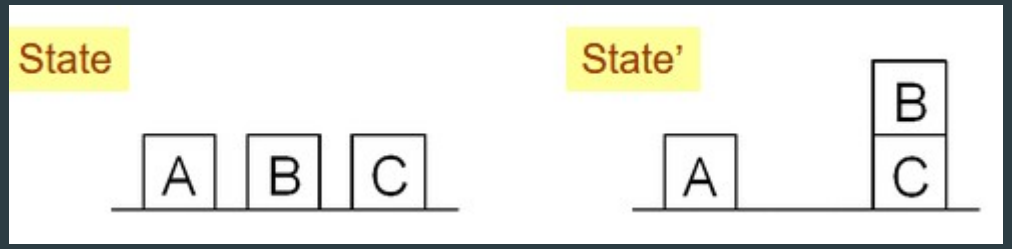
$State = On(A, Table) \wedge On(B, Table) \wedge On(C, Table) \wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(A) \wedge Clear(B) \wedge Clear(C)$

$Action($ $MoveFromTable(B,C)$

PRECOND: $On(B, Table) \wedge Clear(B) \wedge Clear(C) \wedge Block(B) \wedge (B \neq C)$

EFFECT: $On(B,C) \wedge \neg On(B, Table) \wedge \neg Clear(C)$)

$State' = On(A, Table) \wedge On(C, Table) \wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(A) \wedge Clear(B) \wedge On(B,C)$



Solución con PDDL

State' = $On(A, Table) \wedge On(C, Table) \wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(A) \wedge Clear(B) \wedge On(B,C)$

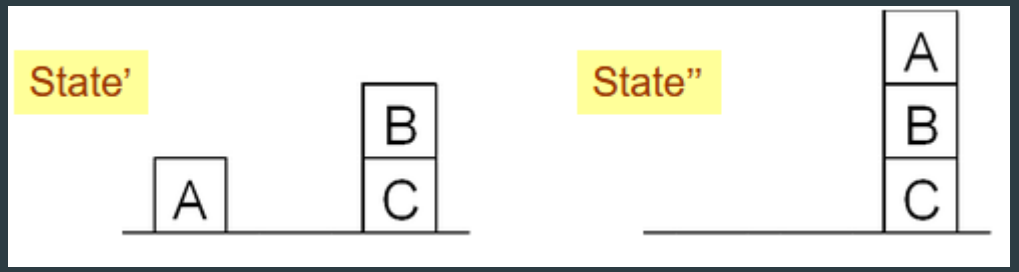
Action($MoveFromTable(A,B)$

PRECOND: $On(A, Table) \wedge Clear(A) \wedge Clear(B) \wedge Block(A) \wedge (A \neq B)$

EFFECT: $On(A,B) \wedge \neg On(A, Table) \wedge \neg Clear(B)$)

State" = $On(C, Table) \wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(A) \wedge On(B,C) \wedge On(A,B)$

Passed Goal Test: $On(A,B) \wedge On(B,C)$



Grafo de planificación (Planning graph)

- Todas las heurísticas utilizadas pueden sufrir de imprecisiones.
- Existe una estructura especial, llamada **grafo de planificación**, que permite representar el problema y proporcionar mejores estimaciones para las heurísticas.
  - Es una aproximación al árbol con todas las posibles acciones desde el estado inicial al resto de estados sucesores a todos los niveles.
  - Permite estimar cuántas etapas se necesitan para ir desde un estado a otro.
  - Es un grafo dirigido organizado en niveles:
    - El nivel S_0 para el estado inicial, consistente en los nodos que representan cada flujo que se mantiene en S_0.
    - El nivel A_0 consistente en los nodos derivados de cada acción aplicable en S_0.
    - Después se alternan niveles S_i con A_i hasta alcanzar la condición de terminación.
  - Los niveles S_i contienen todos los literales que pueden estar activos en el instante i.
  - Los niveles A_i contienen todas las acciones que podrían tener sus precondiciones satisfechas en el instante i.

Grafo de planificación (Planning graph)

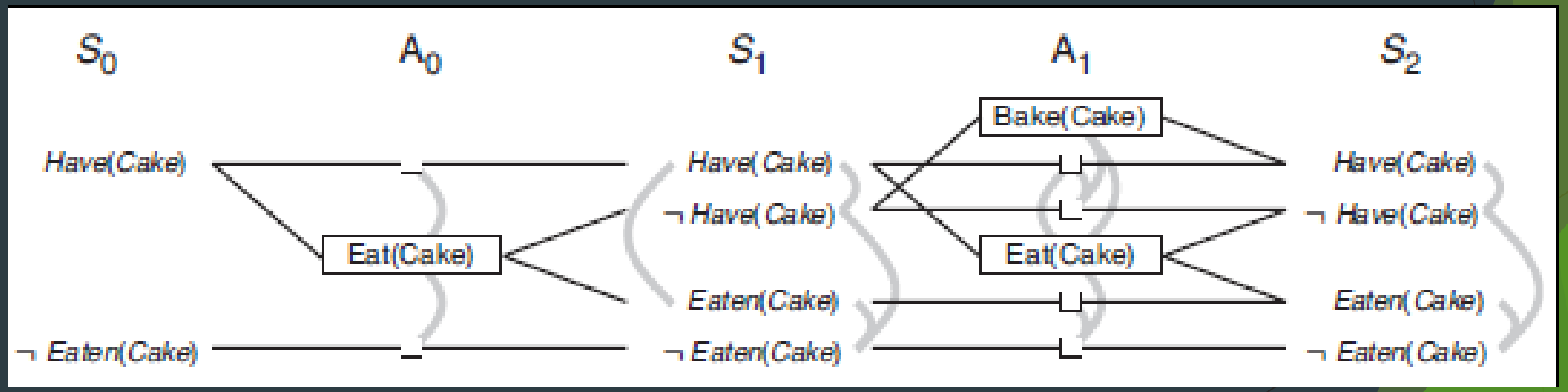
- Cada acción en el nivel A_i está conectada con las precondiciones en S_i y sus efectos en S_{i+1}.
- Un literal aparece porque una acción la ha causado.
  - Un literal persiste si no hay una acción que lo niegue, lo que se representa por una acción de persistencia (o no-op).
    - Para cada literal C, se añade una acción de persistencia con precondición C y efecto C.
- El nivel A_0 contiene todas las acciones que podrían ocurrir en el estado S_0, y también indica los conflictos entre acciones que previene que ocurran a la vez.
  - Las líneas grises indica enlaces de exclusión mutua (o mutex).
    - Por ejemplo, Eat(Cake) es mutuamente exclusivo con la persistencia de Have(Cake) o de ~Eaten(Cake).
- El nivel S_1 contiene todos los literales que podrían resultar de seleccionar cualquier conjunto de acciones en A_0, así como los enlaces mutex indicando los literales que no pueden aparecer juntos.
  - Representa un estado de creencia dependiendo de las acciones elegidas en A_0.

Grafo de planificación (Planning graph)

- Se continúa alternando entre niveles de estado Si y niveles de acción Ai hasta que se llega a un punto en que dos niveles consecutivos son idénticos.
- Finalmente, se consigue:
  - Cada nivel A_i contiene las acciones aplicables en S_i, así como las restricciones entre dos acciones que no pueden ser ejecutadas en el mismo nivel.
  - Cada nivel S_i contiene todos los literales que pueden resultar de las acciones en A_{i-1}, así como las restricciones de qué pares de literales no pueden suceder a la vez.
- La relación de mutex entre dos acciones sucede con alguna de estas condiciones:
  - *Efectos inconsistentes*. Una acción niega el efecto de la otra.
  - *Interferencia*. Uno de los efectos de una acción es la negación de la precondición de la otra.
  - *Necesidades en conflicto*. Una de las precondiciones de una acción es mutuamente exclusiva con la precondición de otra.
- La relación de mutex entre dos literales sucede cuando uno es negación del otro o cuando cada posible par de acciones que pueden conseguir los literales son mutuamente exclusivos, conocido como **soporte inconsistente**.

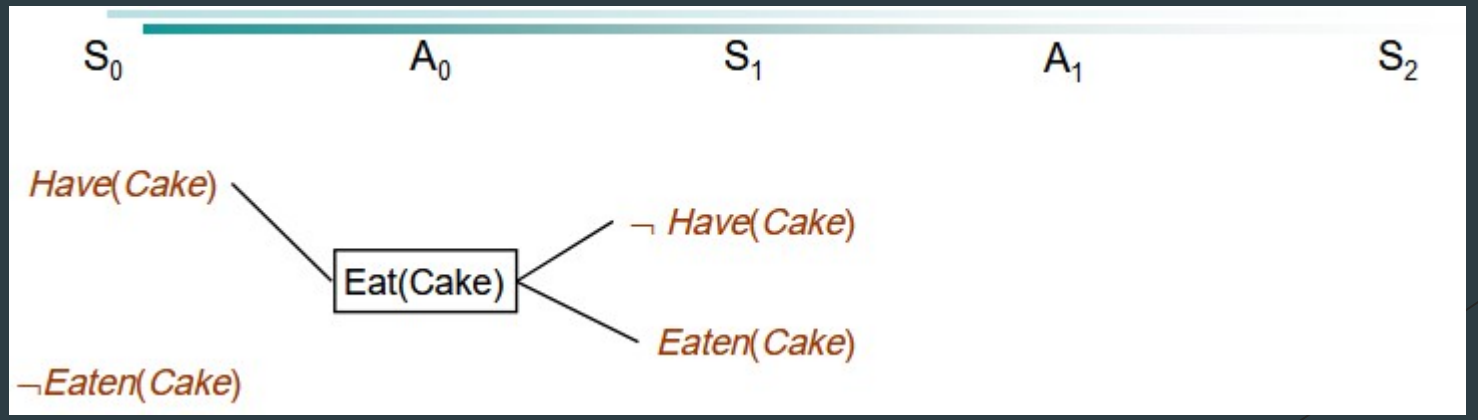
Ejemplo de la tarta

$Init(Have(Cake))$
$Goal(Have(Cake) \land Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg\, Have(Cake) \land Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg\, Have(Cake)$
EFFECT: $Have(Cake))$



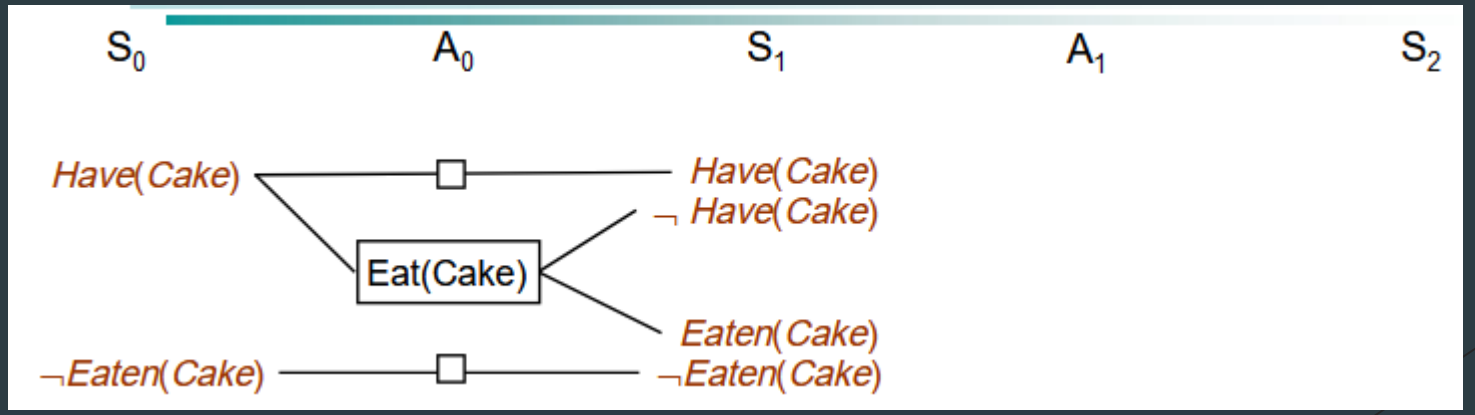
Ejemplo de la tarta

- Se comienza en el nivel S_0 y se determina el nivel de acción A_0 y el siguiente nivel S_1.
- A_0: todas las acciones cuyas precondiciones son satisfechas en el nivel previo.
  - Conecta las precondiciones en S_0 con los efectos en S_1 de las acciones.



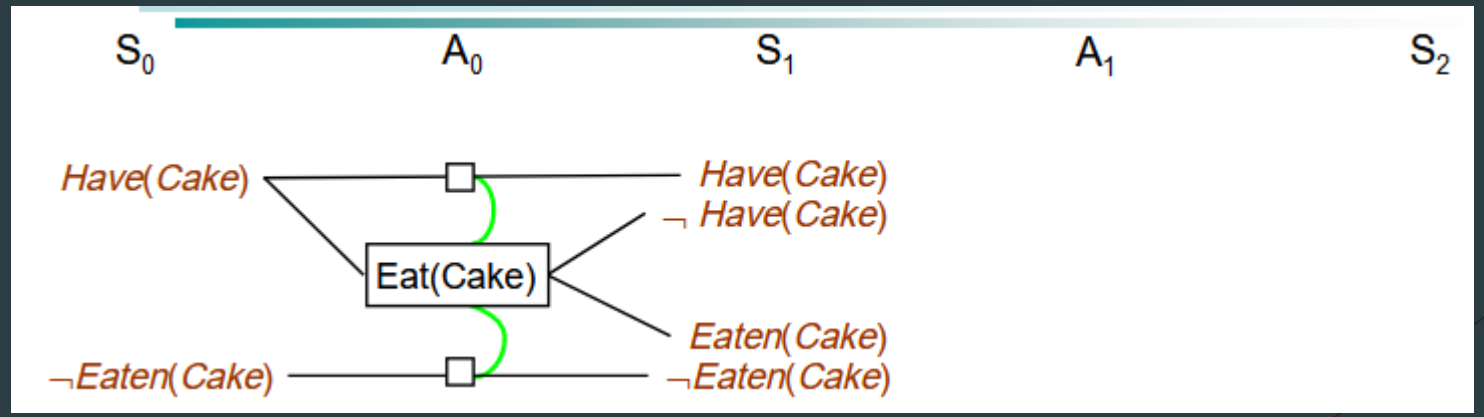
Ejemplo de la tarta

- La inacción se representa por acciones persistentes.
  - Inacción significa que las condiciones permanecen sin cambios.
  - Se usa un pequeño cuadrado para denotar las acciones persistentes.
- El nivel A0 contiene las acciones que pueden ocurrir.



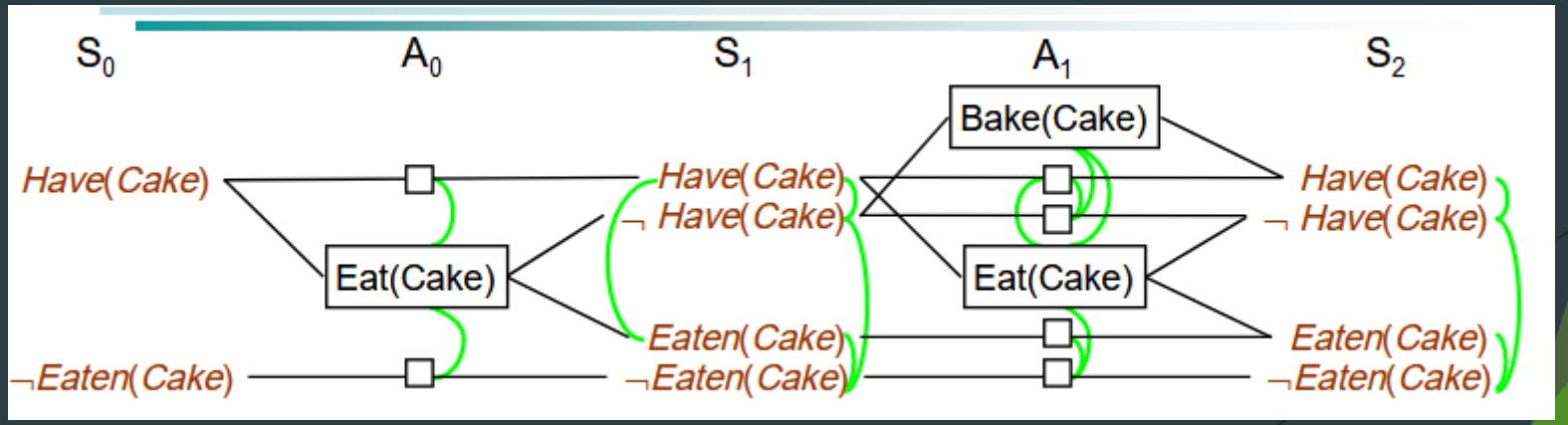
Ejemplo de la tarta

- Los conflictos entre acciones se representan mediante enlaces de exclusión mutua.
- Una relación de **exclusión mutua entre dos acciones** ocurre cuando:
  - **Efectos inconsistentes**. Una acción niega el efecto de otra.
  - **Interferencia**. Uno de los efectos de una acción es la negación de la precondición de otra.
  - Necesidades en conflicto. Una de las precondiciones de la acción es mutuamente excluyente con una precondición de otra.



Ejemplo de la tarta

- El algoritmo continua hasta que dos niveles consecutivos son idénticos (leveled off) o contiene el mismo número de literales.



Grafo de planificación (Planning graph)

- El grafo de planificación proporciona gran información sobre el problema.
  - Si algún literal objetivo no aparece en el último nivel del grafo, entonces el problema no tiene solución.
  - Se puede estimar el coste de conseguir cualquier literal g_i desde el estado s como el nivel en el que g_i aparece por primera vez en el grafo construido desde el estado inicial s. Esto se conoce como el coste de nivel de g_i.
    - Esta estimación puede no ser precisa porque el grafo de planificación permite varias acciones en el mismo nivel, pero se puede utilizar un grafo de planificación en serie, que solo permite una acción por nivel, para calcular las heurísticas.
  - Para calcular el coste de una conjunción de objetivos se pueden usar 3 enfoques:
    - **Heurística de máximo nivel**. Simplemente se escoge el mayor nivel de todos los objetivos.
    - **Heurística de suma de niveles**. Asumiendo la independencia de subobjetivos, es la suma del coste de nivel de cada objetivo.
    - **Heurística de conjunto-nivel**. Elige el nivel en el que todos los literales aparecen en el grafo de planificación sin ser mutuamente exclusivos.

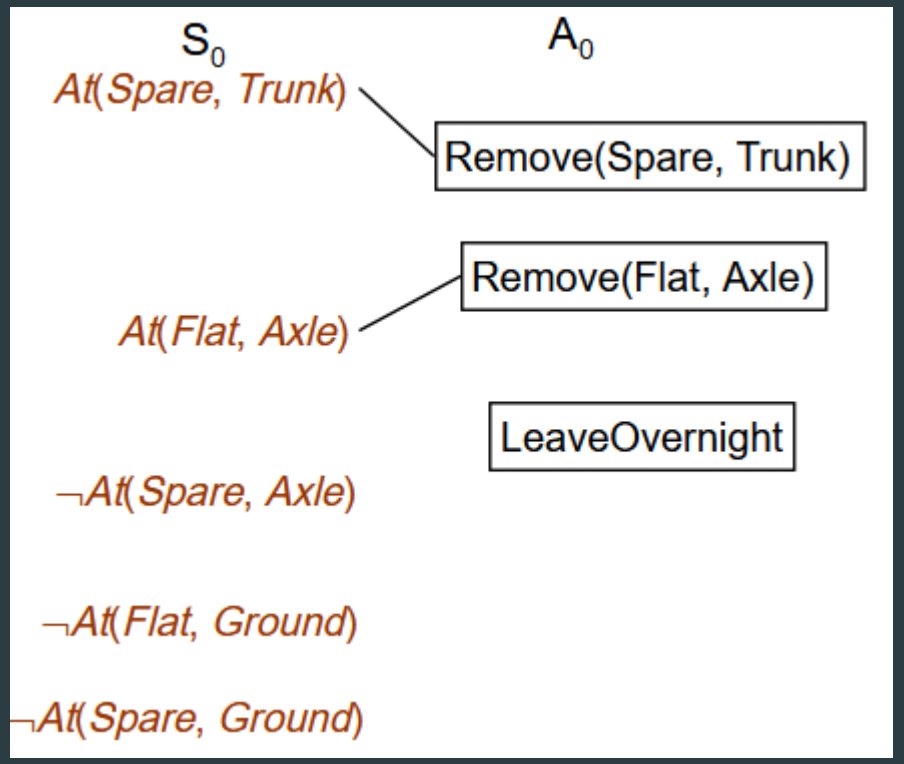
Algoritmo GraphPlan

- A parte de calcular heurísticas, se puede usar directamente el grafo de planificación para obtener un plan.
- El algoritmo GRAPHPLAN añade un nivel al grafo de planificación repetidamente con EXPAND-GRAPH.
  - Una vez todos los objetivos aparecen sin ser mutuamente exclusivos, GRAPHPLAN llama a EXTRACT-SOLUTION para buscar un plan que solucione el problema.
  - Si falla, se expande otro nivel y se vuelve a intentar, terminando en fallo cuando no hay más razones para continuar.

```
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)
```

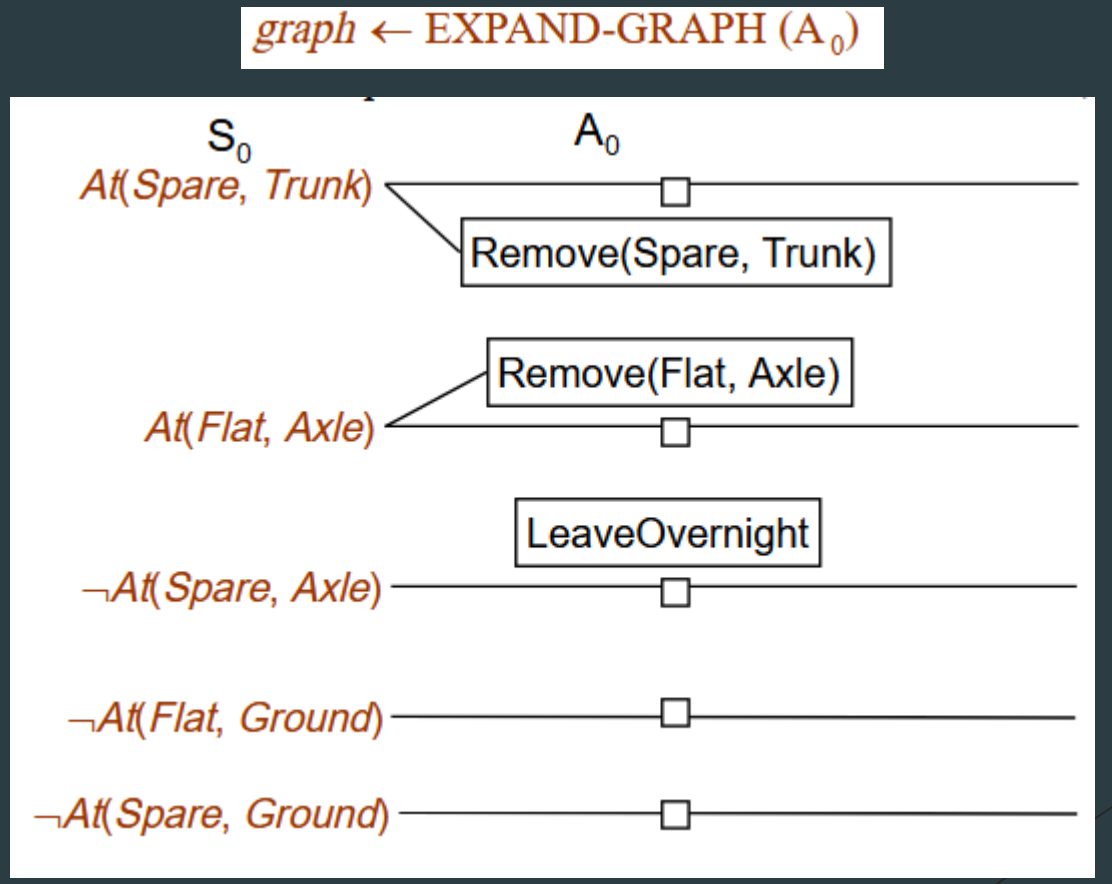
Ejemplo de GraphPlan

- El objetivo At(Spare, Axle) no se puede alcanzar.
- Añadir acciones cuyas precondiciones sean satisfechas por EXPAND-GRAPH(A_0)



Ejemplo de GraphPlan

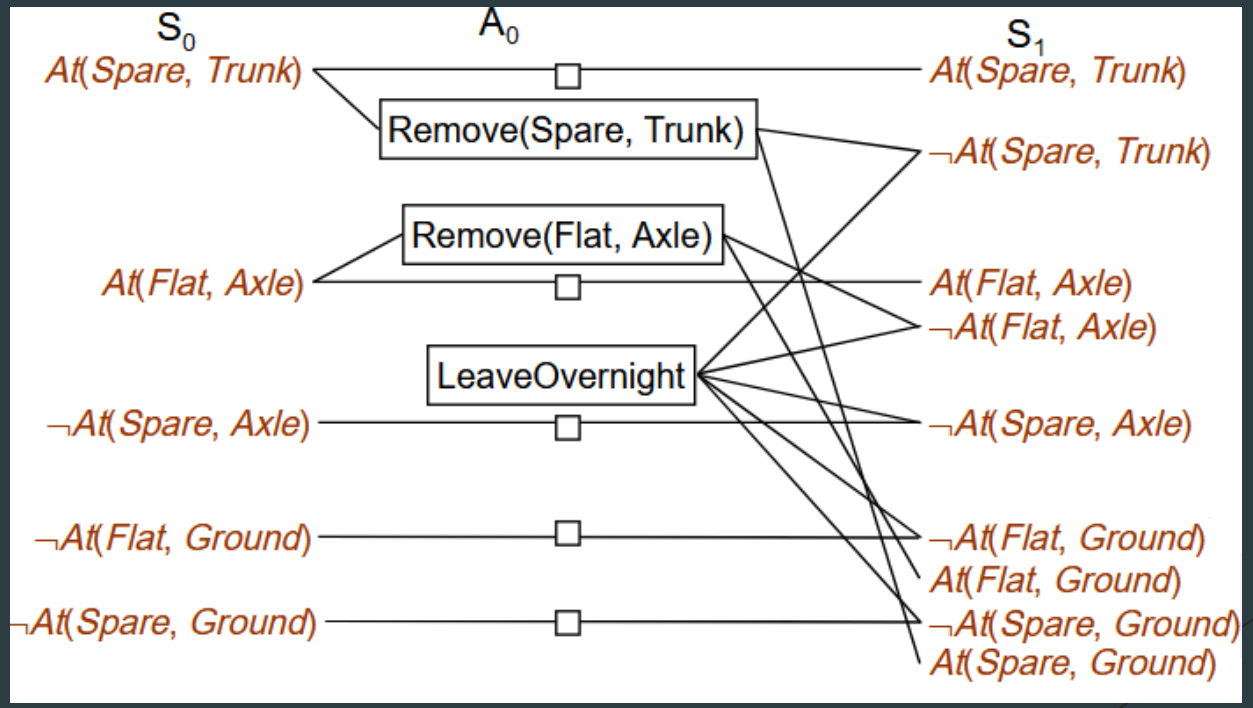
- Además, añadir las acciones de persistencia y las relaciones de mutex (no mostradas)



Ejemplo de GraphPlan

- Añadir los efectos en el nivel S_1.

graph ← EXPAND-GRAPH (A_0)

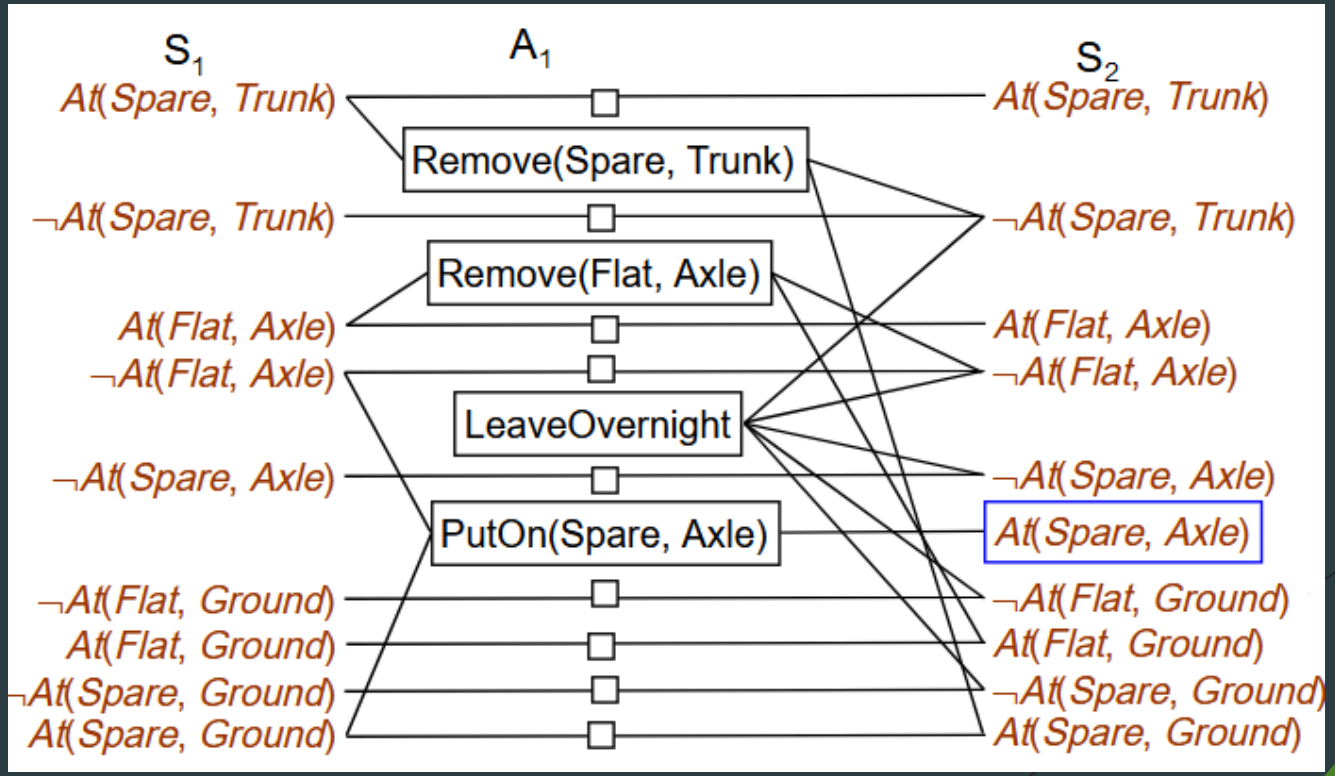


Ejemplo de GraphPlan

- EXPAND-GRAPH también tiene en cuenta las relaciones mutex.
  - Efectos inconsistentes.
    - Ejemplo: Remove(Spare, Trunk) y LeaveOverNight.
  - Interferencia.
    - Ejemplo: Remove(Flat, Axle) y LeaveOverNight
  - Necesidades en conflicto.
    - Ejemplo: PutOn(Spare, Axle) y Remove(Flat, Axle)
  - Soporte inconsistente.
    - Ejemplo en S2: At(Spare, Axle) y At(Flat, Axle)

Ejemplo de GraphPlan

- Iterar hasta que el objetivo At(Spare, Axle) pueda conseguirse
- En S2 el literal objetivo existe y no es mutex con ningún otro.
  - La solución podría existir y se utiliza EXTRACT-SOLUTION para intentar conseguirla.



Algoritmo GraphPlan

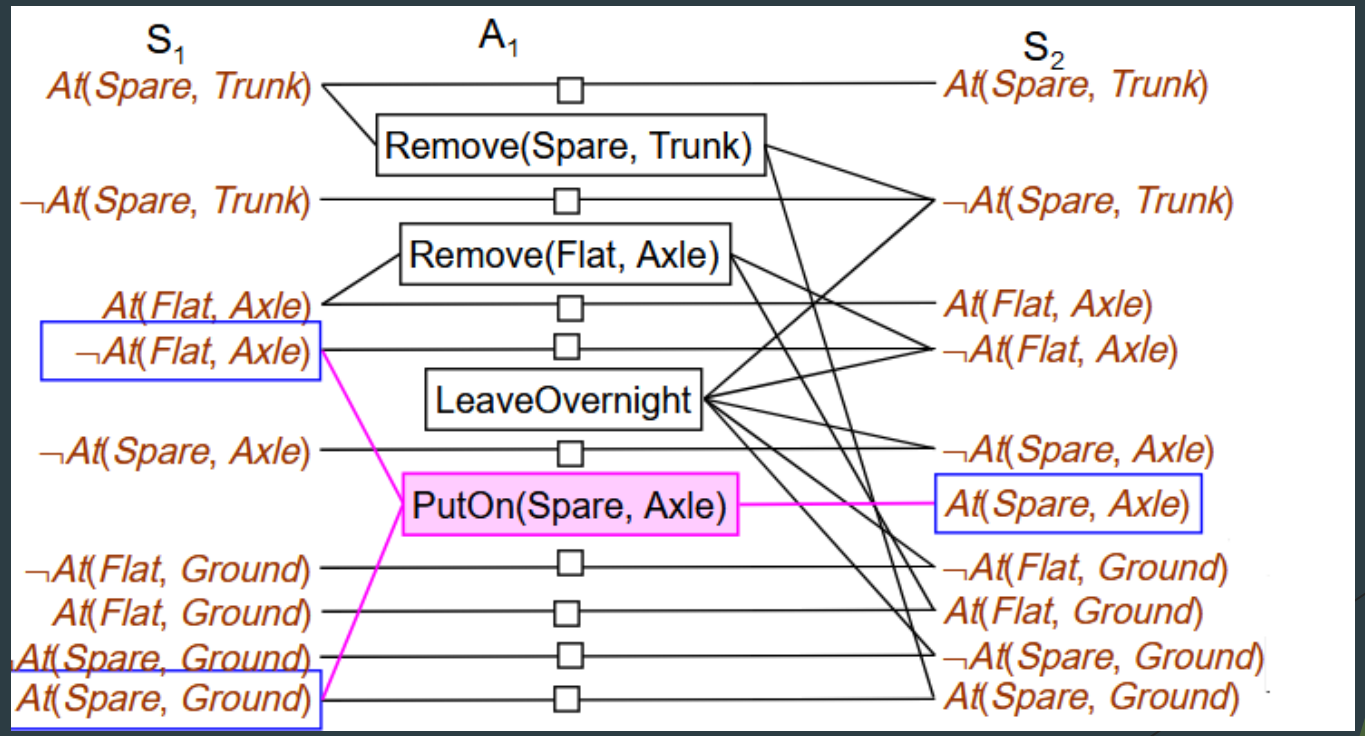
- EXTRACT-SOLUTION se puede formular como un problema de satisfacción de restricciones CSP booleano:
  - Las variables son las acciones en cada nivel
  - Los valores de cada variable están dentro o fuera del plan,
  - Las restricciones son las exclusiones mutuas y la necesidad de satisfacer cada objetivo y precondición.
- Alternativamente, EXTRACT-SOLUTION se puede formular como un problema de búsqueda regresiva.
  - Cada estado en la búsqueda contiene un puntero a un nivel en el grafo de planificación y un conjunto de objetivos insatisfechos.

Algoritmo GraphPlan

- EXTRACT-SOLUTION:
  - Estado inicial.
    - Último nivel del grafo de planificación y los objetivos del problema de planificación.
  - Acciones.
    - Seleccionar cualquier conjunto de acciones no conflictivas que cumpla los objetivos del estado.
      - No conflictivas significa que no haya mutex entre ellas.
      - La unión de precondiciones se convierte en el objetivo del siguiente estado.
  - Objetivo.
    - Alcanzar el nivel S_0 de forma que todos los objetivos sean satisfechos.
  - Coste.
    - Valor de 1 por cada acción realizada.

Algoritmo GraphPlan

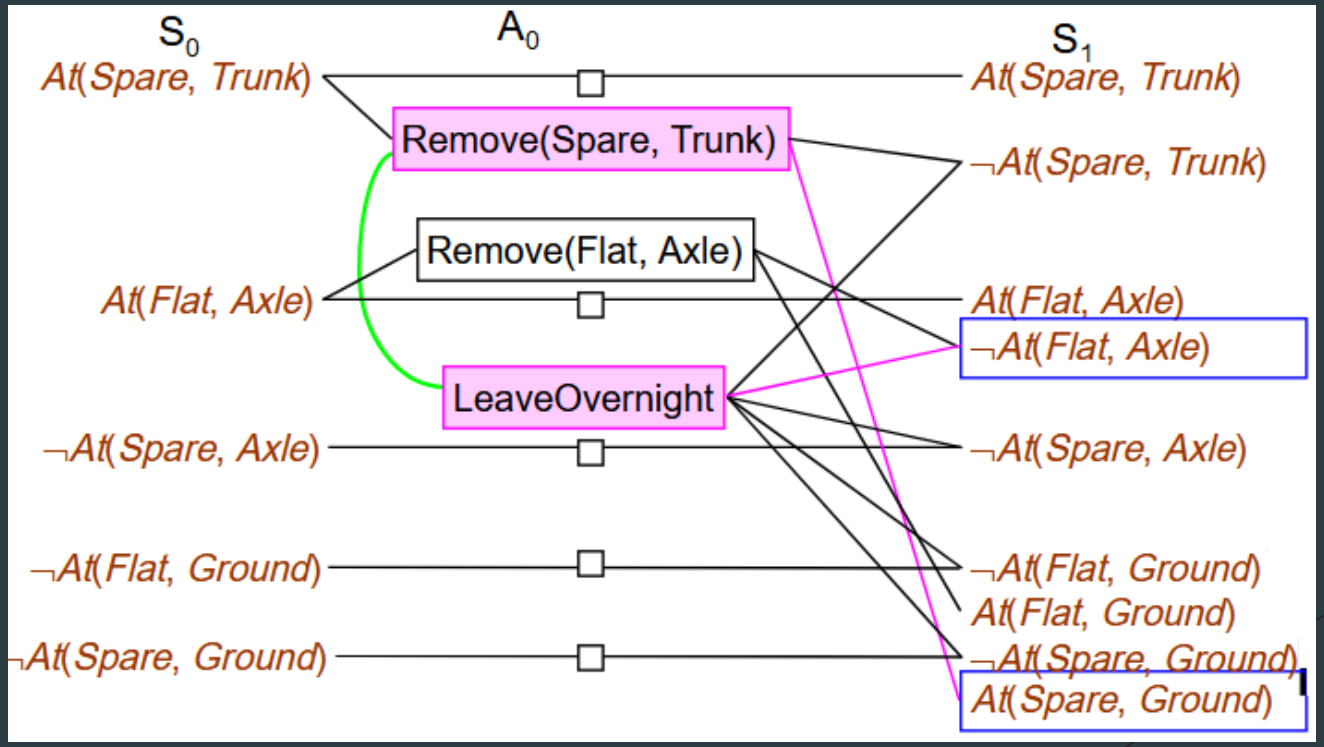
- La única acción para conseguir el objetivo At(Spare, Axle) es PutOn



Algoritmo GraphPlan

- Dos opciones para conseguir ¬At(Flat, Axle) y At(Spare, Ground)

  1) Remove(Spare, Trunk) y LeaveOvernight, pero son mutex.



Algoritmo GraphPlan

- Está demostrado que la etapa EXPAND-GRAPH termina eventualmente, y que no se quedará en un bucle infinito.
  - Los grafos de planificación crecen o decrecen monotónicamente.
    - Los literales crecen monotónicamente. Una vez un literal aparece en un nivel, aparecerá en todos los niveles subsiguientes debido a las acciones de persistencia.
    - Las acciones crecen monotónicamente. Una vez una acción aparece en un nivel, aparecerá en todos los niveles subsiguientes como consecuencia del crecimiento monotónico de los literales. Si las precondiciones de una acción aparecen en un nivel, aparecerán en niveles subsiguientes, así como la acción.
    - Las exclusiones mutuas decrecen monotónicamente. Si dos acciones o literales son mutex en un nivel, también lo son en los niveles previos. Sin embargo, una de las condiciones de mutex de literales implica que todas las acciones para conseguirla sean mutex con todas las acciones para conseguir el otro. Como las acciones crecen monotónicamente, por inducción las exclusiones mutuas deben reducirse.
  - Debido a estas propiedades y a que existe un número finito de acciones y literales, cualquier grafo de planificación acaba llegando a eventualmente a un estado de level off.