Proyecto Fin de Grado
Ingeniería Robótica Electrónica y Mecatrónica

Planificación automática basada en redes jerárquicas de tareas para resolver problemas en contextos multi-robot

Autor: Alejandro Robles Bernal

Tutor: J. Iván Maza Alcañiz

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018

ANDALUCÍA TECH

Proyecto Fin de Grado
Ingeniería Robótica Electrónica y Mecatrónica

Planificación automática basada en redes jerárquicas de tareas para resolver problemas en contextos multi-robot

Autor:
Alejandro Robles Bernal

Tutor:
J. Iván Maza Alcañiz
Profesor Titular

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018

Proyecto Fin de Grado: Planificación automática basada en redes jerárquicas de tareas para resolver problemas en contextos multi-robot

Autor: Alejandro Robles Bernal
Tutor: J. Iván Maza Alcañiz

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Resumen

Los planificadores por redes jerárquicas de tareas o Hierarchical Task Network (HTN) son una propuesta popular para construir planes con un conjunto de tareas para el control de sistemas inteligentes. Es utilizado en campos como la robótica (robot Curiosity en Marte) control de sistemas como el Astronomical Image Processing System (AIPS) o en videojuegos (Killzone 2) entre otros.
En el ámbito de la robótica como el mundo real es extremadamente complejo, la representación del dominio del robot consiste en una serie de instrucciones de alto nivel sobre cómo el robot debe realizar las tareas, pero dándole la oportunidad de elegir las acciones de bajo nivel y su orden.
En este proyecto definiremos los HTN y la aplicación de un popular ejemplo utilizando una versión del programa Simple Hierarchical Ordered Planner 2 (SHOP2) para Python llamada Pyhop.

Índice

1 Introducción

En este capítulo se verá una breve introducción a la planificación de tareas, que necesitaremos para una mejor comprensión del programa desarrollado.

1.1 Estructura del trabajo

Comenzamos en el Capítulo 2 presentando una breve descripción de la planificación de tareas y utilizándola para motivar a nuestra decisión de usar la planificación de HTN. Luego definimos con mayor precisión el formalismo HTN utilizado en el resto del tfg. Este formalismo no solo presenta una nueva representación gráfica del algoritmo del HTN si no que también identifica los diferentes tipos de retrocesos involucrados en la búsqueda de una solución.
En el capitulo 3 hablaremos del HTN que vamos a utilizar, llamado Pyhop, una versión para Python del algoritmo SHOP2. Explicaremos sus usos y ventajas frente a otros HTN.
Por último en el capítulo 4 mostraremos el ejercicio realizado mediante Pyhop, que será una versión del típico Dock Worker Robot (DWR)

1.2 Trasfondo de los planificadores de tareas

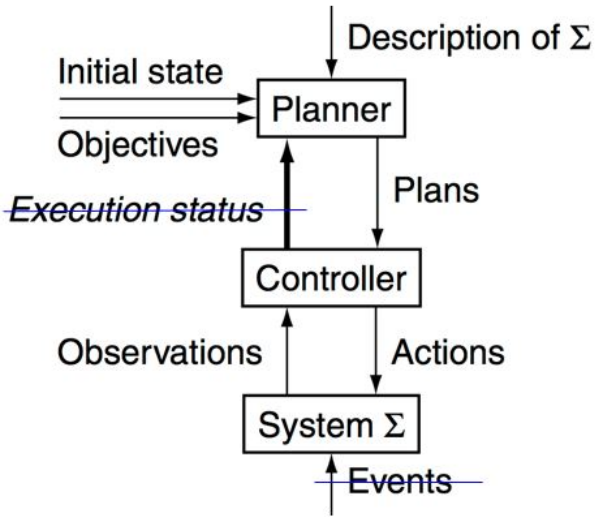
El libro (Ghallab et al., 2004) empieza con la siguiente introducción:"Planificación es la componente de razonada de una acción. Es un proceso de deliberación abstracto y explícito que elige y organiza acciones previendo su resultado esperado. Esta deliberación tiene como meta completar satisfactoriamente una serie de objetivos preestablecidos. La planificación automatizada es un área de Inteligencia Artificial, o en inglés, Artificial Intelligence (AI), que estudia este proceso de deliberación computacionalmente".
Es posible controlar un sistema robótico usando un mecanismo de comportamiento, Brooks (1987) y Arkin (1998) argumentan que la planificación no es necesaria para que un robot actúe. El principio es crear tareas para realizar con éxito la planificación de acuerdo con los objetivos de alto nivel. Y usando un sistema de comportamiento, el robot puede actuar para alcanzar estos objetivos. El problema con tal mecanismo es la previsión, los robots deciden solo la siguiente acción y no planifican el futuro. Le permite al robot abordar ciertos objetivos específicos, pero lo que queremos, es que nuestros robots alcancen objetivos más complejos. Por otro lado, un planificador, utilizando un proceso de decisión, puede responder a objetivos tan complejos.

1.2.1 Diferentes enfoques en la planificación simbólica

En las últimas décadas, el campo de la AI ha sido muy activo y ha producido una gran cantidad de enfoques para la planificación automatizada. Muchos de ellos se basan en un mecanismo que construye un plan de solución razonando sobre las acciones disponibles y sus diferentes secuencias posibles. Hay muchas maneras diferentes de formalizar tal mecanismo,abordando el planificador desde el dominio, podemos distinguir tres tipos:

- Específico según el dominio: Planificador hecho para un solo dominio, no funciona bien, si es que funciona, en otros dominios de planificación. La mayoría de los planificadores en tiempo real funciona con este tipo.

- Independiente del dominio: En un principio debería funcionar en todos los dominios, pero en la práctica, esto no es posible, por lo que se crean una serie de asunciones que restringirán los dominios, la planificación clásica entra dentro de este apartado.



- Configurable según el dominio: Son tipos de planificadores independientes del dominio, pero con una serie de entradas con información sobre como resolver problemas en el respectivo dominio. Esto permite que el motor del planificador tenga menos restricciones al tipo anterior.

1.2.2 Planificación clásica

Vamos a hacer un pequeño inciso para hablar de la planificación clásica, como hemos comentado, se incluye dentro de los planificadores independientes. Debemos recordar que para un planificador, incluso los problemas más simples pueden tener grandes espacios de búsqueda, por lo que debemos preguntarnos qué dificultades nos podemos encontrar en la creación de un planificador. Empezaremos por definiciones básicas utilizadas en la planificación clásica:

- Tareas: Una tarea representa una actividad para realizar. Sintácticamente, una tarea consiste en un símbolo de la tarea, seguido de una lista de argumentos. Una tarea puede ser primitiva o compuesta.
  - La tarea primitiva debe ser realizada por un operador de planificación.
  - Una tarea abstracta o no primitiva, es aquella debe descomponerse en tareas primitivas usando un método; cualquier método cuya cabeza se unifica con el símbolo de la tarea y sus argumentos puede ser potencialmente aplicable para descomponer la tarea.
- Operadores: En la planificación clásica, un operador es algo que debe ser ejecutado por un agente, e indica como se debe de realizar una tarea. También pueden ser llamados acciones.
- Métodos: Los métodos dan una serie de procedimientos para descomponer una tarea en un conjunto parcialmente ordenado de subtareas, ya sean primitivas, o no.
  La versión más simple de un método tiene tres partes:
  - La tarea para la cual se usará el método.
  - La condición previa o requisito previo que el estado actual debe satisfacer para que el método sea aplicable.
  - Las subtareas que debe lograrse para llevar a cabo esa tarea.
- Dominio: El dominio es el conjunto de tareas y métodos que puede usar el planificador para encontrar una solución al problema actual.

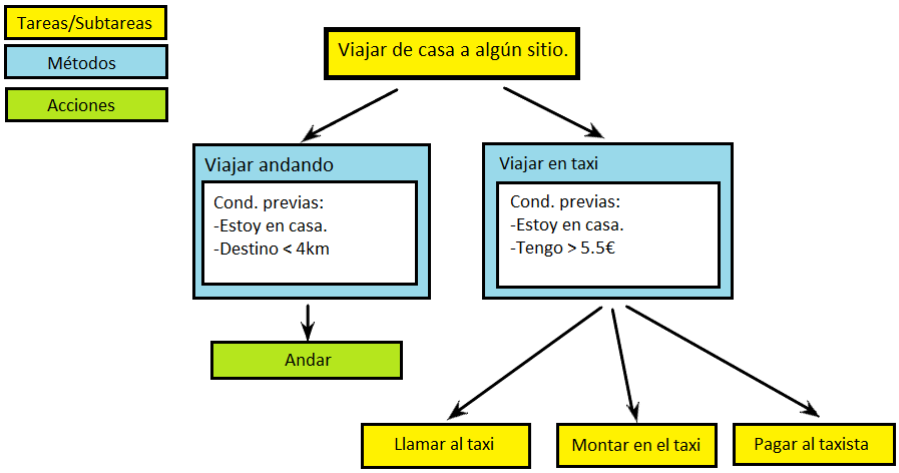


Figura 1.1 Ejemplo básico de una HTN.

Sobre las dificultades que nos encontramos en los planificadores clásicos, la principal nos la encontramos con los problemas del lenguaje, ya que debemos traducir el problema a un lenguaje llamado language-recognition, que básicamente significa que debemos formular problemas que se puedan resolver con sí o no.

1.2.3 Planificador espacio-estado

El más obvio -y el primero históricamente- es la planificación espacio-estado.
Estos planificadores son en su mayoría procedimientos de búsquedas,en los que cada nodo es un estado en el mundo conocido. En un planificador espacio-estado como Stanford Research Institute Problem Solver (STRIPS), Fikes y Nilsson (1971), la búsqueda utiliza una representación exhaustiva de los estados completos para planificar. Estos planificadores funcionan utilizando:

- Búsqueda hacia atrás: El proceso comienza con el objetivo e intenta avanzar hacia el estado inicial. La ramificación es menor comparada con la búsqueda hacia adelante, la técnica mas utilizada en esta búsqueda se llama **Lifting**, que se basa en el instanciamiento parcial de los operadores, pese a ello, la ramificación sigue siendo muy grande.
- Búsqueda hacia adelante: El proceso es el opuesto a la anterior opción, el cual garantiza una solución si cualquiera de sus trazadores no deterministas devuelve un plan, lo cual siempre pasará si existe una solución, pero su factor de ramificación es muy alto, lo que puede repercutir en el rendimiento. Alguna de las implementaciones de esta búsqueda son:
  - Breadth-first search: Empieza en la raíz y se explora los nodos vecinos, se repite el proceso hasta recorrer todo el grafo.

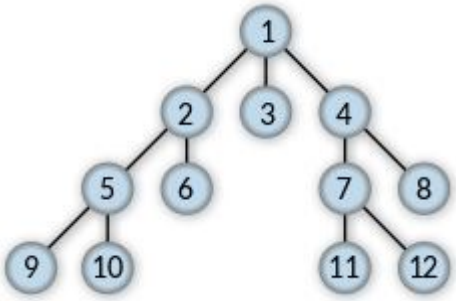


Figura 1.2 Breadth-first search.

  - Depth-first search: Expande los nodos que encuentra de forma recurrente en un camino recto hasta encontrar la primera solución válida.

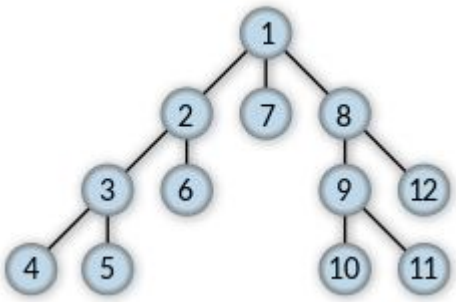


Figura 1.3 Depth-first search.

- Best-first search: Explora el grafo expandiendo el nodo más prometedor elegido según una determinada regla que suele depender de la meta.
- Greedy search: Similar a la anterior, pero se utiliza la heurística para calcular el siguiente nodo.

Pese a que la primera y la tercera opción son robustas y completas, requieren de mucha memoria, por lo que se utilizan las dos restantes.

Tanto el encadenamiento hacia atrás como el enfoque similar de encadenamiento directo se benefició de la búsqueda heurística[1], un ejemplo notable fue el planificador de FF Hoffmann y Nebel (2001).

1.2.4 Planificador plan-espacio

Otro enfoque común es la planificación plan-espacio, cuya paternidad se atribuye a Sacerdoti y su planificador de NOAH Sacerdoti (1974). El principio es construir iterativamente un plan sin depender de una representación explícita del estado actual. Una solución que se encuentra en la planificación plan-espacio se representa como un conjunto de acciones y vínculos causales, por lo tanto, es más flexible y permite un mejor control durante la ejecución.
El problema en este caso se resuelve cuando deja de haber defectos en el plan. Los defectos más comunes son:

- Metas abiertas: Una acción tiene una condición previa que no hemos establecido. Para resolver esta meta, buscamos una acción con la que podamos establecer la condición, tras ello creamos un enlace mediante variables.

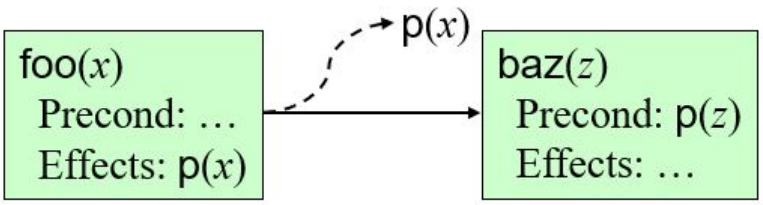


Figura 1.4 Solución de las metas abiertas.

- Amenazas: Son interacciones con condiciones borradas por otra acción. Este fallo se resuelve imponiendo una restricción a la acción que nos borra la condición

[1] Es una técnica en AI que emplea heurística en sus movimientos. Se utiliza porque ayuda a reducir en número de alternativas de un número exponencial a uno polinómico, este método no es siempre efectivo.

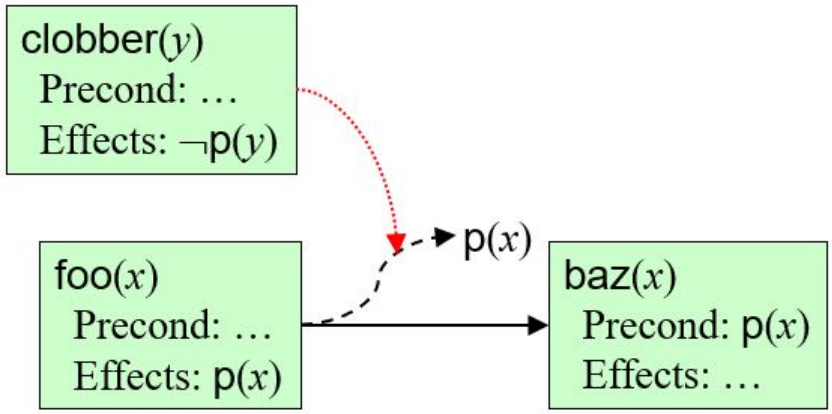


Figura 1.5 Solución de las amenazas.

Usualmente se utiliza el algoritmo Personal Software Process (PSP) el cual es robusto y completo, ya que devuelve un plan parcialmente ordenado, además puede ejecutar acciones en paralelo si el ambiente lo permite.
El último método que presentamos aquí será el que utilicemos en el resto del proyecto, y será explicado con más detalles.

2 Red jerárquica de tareas

2.1 Funcionamiento de una HTN

La idea general de la planificación por HTN consiste en una reducción del problema, en vez de metas, tenemos tareas, y métodos que descomponen las tareas a completar. En cada iteración, el planificador descompone una tarea no primitiva usando un método, y el proceso continúa hasta que solo haya acciones.
Los planificadores HTN pueden ser tanto específicos del dominio, como configurables. Además podemos elegir entre dos opciones respecto al movimiento del planificador, podemos elegir si queremos que vaya de iazquierda a derecha, o de arriba a abajo.
Como nuestra implementación en la planificación de HTN usa organización total[1], limitamos el formalismo

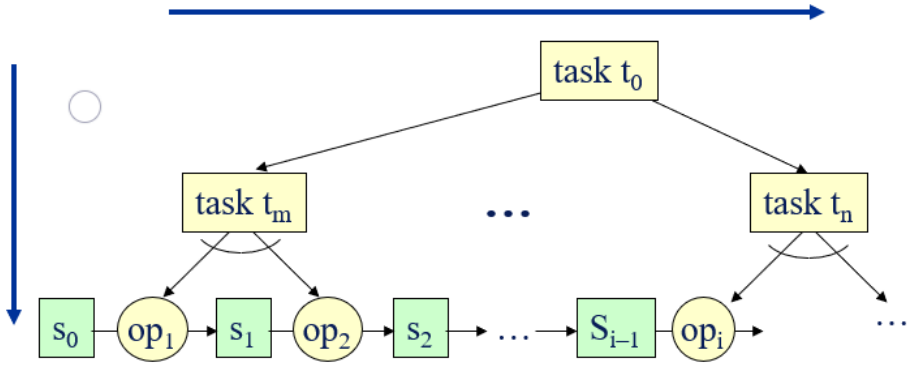


Figura 2.1 Opciones de dirección.

a este caso.
Un problema de planificación HTN está compuesto por 3 partes:

$$P = (d, s_0, D) \tag{2.1}$$

Donde **d** es la meta a realizar, expresada como una tarea, **s0** es el estado inicial y **D** es el dominio del planificador HTN. En un planificador HTN, el dominio se compone por:

$$D = (A, M) \tag{2.2}$$

Donde **A** representa las acciones que utilizará la red, siendo una acción un conjunto finito de operadores, y **M** es el conjunto de métodos usados en la red.
Formalmente, una acción está definida por 4 partes:

$$a = (nombre(a), tarea(a), condiciones - previas(a), efectos(a)) \tag{2.3}$$

[1] El proceso está totalmente acotado y delimitado

Donde **nombre(a)** es el nombre y los parámetros de la acción, **tarea(a)** la tarea primitiva lograda por este operator, **condiciones-previas(a)** un conjunto de pre-requisitos que debe ser verdaderos para que la acción sea aplicable, **efectos(a)** los efectos de la acción.
Un método se compone de 5 partes:

$$m = (nombre(m), tarea(m), condiciones - previas(m), subtareas(m), restricciones(m)) \tag{2.4}$$

Nombre(m) corresponde al nombre y parámetros que utilizará el método, **tarea(m)** es un resumen de la tarea a la que se puede aplicar este método, **condiciones-previas(m)** especifica cuándo el método es aplicable, **subtareas(m)** es el conjunto de tareas que usa para definir la tarea(m), y **restricciones(m)** es el conjunto de restricciones previas[2] entre cada tarea que forman las subtareas(m).
El proceso de planificación funciona seleccionando los métodos aplicables de **M** y aplicándolos a tareas abstractas en **d** de una forma similar a la utilizada en el algoritmo depth-first search[3]. En cada iteración, este proceso dará como resultado la descomposición de una tarea no primitiva mediante la elección de un método aplicable (siempre que las condiciones previas del método se cumplan). El proceso continúa hasta que **d** sólo le quedan acciones. En cualquier etapa de la planificación si no se puede aplicar ningún método para la tarea actual, el planificador retrocederá y buscará un método alternativo para dicha tarea. Además, en cualquier etapa de la planificación podemos calcular el estado simbólico **s** actual, aplicando los efectos de todas las acciones en la descomposición d (siempre aplicando los requisitos previos de cada acción).

Entrando en detalles, d es una red de tareas: d = (U,C) donde **U** es un conjunto de nodos de la tarea y **C** es un conjunto de restricciones previas. El proceso de planificación consiste en la descomposición de un nodo de la tarea $\mathbf{u} \in \mathbf{U}$ usando un método **m**. que es una instancia del método M, la tarea (m) = t_u (t_u es la tarea del nodo **u**) y cada condición previa(**m**) se mantiene en el estado simbólico actual **s**. El método **m** descompone **u** en subtareas con:

$$\delta(d,u,m) = ((U-u) \cup subtareas(m), C' \cup condprev(m)) \tag{2.5}$$

Tenga en cuenta que C′ es una copia modificada de C en d: si subtareas(m)=u_1,u_2, entonces la restricción u < v (v es otra tarea) se reemplaza con u_1 < v y u_2 < v.
A veces durante el proceso de planificación, se pueden obtener varios métodos **M** de pueden realizar la tarea t_u, para estos casos el planificador guarda un punto de retroceso llamado Backtrack Point (BP) como una posible alternativa para cada método $\mathbf{m} \in \mathbf{M}$. Hay dos etiquetas asociadas a BP: **tipo**() = descomposición y tarea(BP) = **m**. La función "tipo" proporciona el tipo del punto de retroceso (descomposición, enlace, orden...) y la función "tarea" asocia una tarea primitiva o no primitiva[4] a BP. El planificador realiza una primera búsqueda en profundidad, toma decisiones y crea los puntos de retroceso correspondientes, si falla regresa a un punto de retroceso anterior.
Además, puede ocurrir que un método tenga subtareas con parámetros que no estén fundamentados. En en este caso se produce una vinculación variable: se crea un punto de retroceso BP y para cada valor posible de cada parámetro se agrega una alternativa para un posible retroceso con un **tipo**(BP) = enlace y **task**(BP) = **m**. En realidad, cada alternativa almacena un $\mathbf{m}'$, una copia de **m** tal que todos los parámetros de las subtareas están registrados. En una HTN de orden total, aunque el plan de solución está totalmente ordenado, el dominio **D** permite métodos contener subtareas parcialmente ordenadas: algunas subtareas de métodos pueden tener solo un conjunto limitado restricciones previas. Para obtener todas las órdenes totales, linealizamos las ordenes parciales. Por ejemplo, si tenemos: **m** = [..., subtareas(m) = u_1, u_2, u_3, restricc(m) = $\varnothing$], obtenemos las siguientes órdenes: $u_1 \prec u_2 \prec u_3$, $u_1 \prec u_3 \prec u_2$,..., hasta un total de seis órdenes. Para cada una de estas órdenes el planificador creará una alternativa con un punto de retroceso (BP) con tipo(BP) = orden y tarea(BP) = **m**. De nuevo, cada alternativa guarda m′, una copia de m con todas las restricciones para obtener subtareas totalmente ordenadas.
Con este proceso, el plan parcial π' se puede calcular en cualquier etapa de la planificación, luego, aplicando todas las acciones que contiene (y respetando los enlaces entre ellas) podemos calcular el estado actual **s** completo. Gracias a este cálculo, es posible tener una afirmación evaluable para todas las condiciones previas, una función que realiza cálculos elaborados sobre el estado actual **s**, mientras se considera como una literal:

[2] Las restricciónes previas: **u** < **v** significa que todas las subtareas de **u** deben realizarse antes de que pueda comenzar cualquier subtarea de **v** (aplicable a las acciones también). Es importante tener en cuenta que no estamos utilizando todas las restricciones posibles de la clásica definición de HTN.

[3] El funcionamiento de este algoritmo se basa en empezar la búsqueda por nodo inicial, para seguir explorando hasta el final de la rama, después se volverá a la ramificación anterior y continuará por un nueva bifurcación.

[4] Una tarea puede ser primitiva si es directamente ejecutable (una acción) o no primitiva si tiene que ser descompuesta en subtareas

debe ser verdadero (junto con las otras literales) para que el planificador considere las condiciones previas como válidas.
Este formalismo debe hacer que sea obvio que el experto de dominio proporciona conocimiento al planificador en la forma de la jerarquía de tareas. De hecho, la jerarquía de tareas permite al planificador razonar solo en un conjunto limitado de tareas, además, el experto puede proporcionar un orden para las tareas que también evita cualquier orden que reduce aún más el número de intentos antes de encontrar el plan de solución.

2.2 Ilustración del algoritmo

En esta sección ilustrará algunos de los pasos internos utilizados por el algoritmo que utilizan los HTN. Los procedimientos utilizados se representarán al mismo tiempo que las ilustraciones. Se presentarán los elementos más importantes enfocados en un mismo dominio ficticio para enlazar todos los diferentes casos.

2.2.1 Método de selección

El primer mecanismo HTN presentado es la selección del método, cuando una tarea abstracta debe descomponerse y hay varios métodos disponibles. La figura 2.1 muestra un extracto del dominio y la forma en que crece el árbol de descomposición (cómo se cambia el plan parcial). En el "**PARTIAL PLAN**" parte (lado derecho) vemos dos pasos numerados a la derecha y separados con la línea punteada:

1. El estado inicial con la tarea a realizar, llamado **T1**.
2. La descomposición de esta tarea cuando método ha sido elegido.

En este ejemplo en particular, asumimos que de entre todas las condiciones, sólo "**Precond1**" y "**precond2**" se pueden resolver en el estado inicial, por lo tanto, solo los métodos correspondientes a estas pueden ser aplicados.La HTN elige aleatoriamente un método y lo usa para descomponer T1. Como solo hay otro método posible, se guarda en caso de que se necesite un retroceso. Por lo que se crea un punto de retroceso de tipo **decompo**, y una sola alternativa.

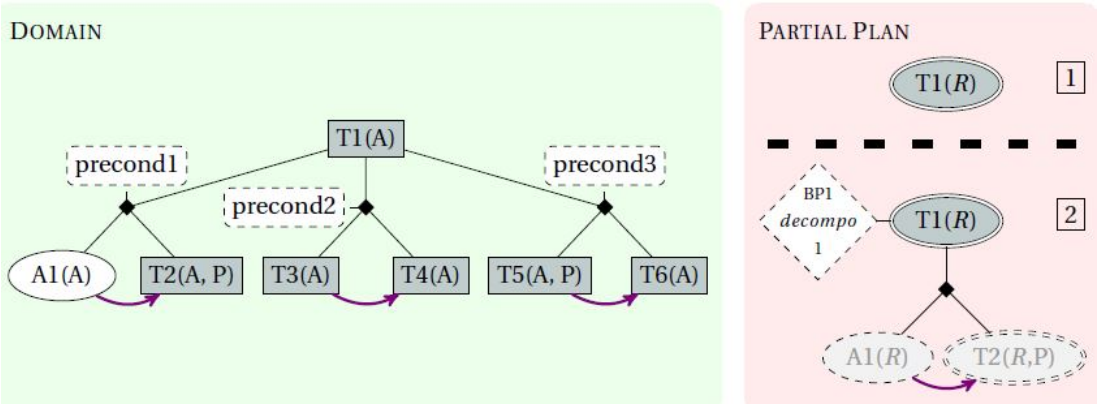


Figura 2.2 Ilustración del mecanismo de selección del método HTN.

En el paso 1, supongamos que solo las condiciones "**precond1**" y "**precond2**" están satisfechas, luego una se elige (al azar) y la otra es mantenido como una alternativa del punto de retroceso (de tipo descomposición). Tenga en cuenta que la variable A ha sido vinculada a **R**, mientras que la variable **P** (en **T2**) sigue siendo libre, por lo que es necesaria una unión variable.
Sobre la norma decidimos que las tareas del dominio están representadas con un rectángulo gris. Cada método que coincide con esta tarea (por ejemplo, tarea(m) = NombreTarea) se representa como un diamante negro, sus requisitos previos en un pequeño rectángulo discontinuo, y su descomposición está representada debajo del mismo (el nombre del método no está representado en la figura). Para representar la restricciones previas, **condprev(m)**, utilizamos enlaces causales que se muestran como las flechas violetas (por ejemplo, en la figura **A1(A)** debe ocurrir antes de **T2(A,P)**). Por lo general, el panel "DOMAIN" (lado izquierdo) solo contiene un extracto del dominio completo: la parte que es relevante en el paso de planificación representado. A medida que se construye el plan parcial, el panel derecho representará los estados del diagrama de flujo.

Las acciones se colorearán con un color para cada agente, aunque las condiciones previas para la acción **A1(R)** no se han probado (se acaban de agregar) por lo que está marcado como "listo" (texto gris y borde discontinuo, como para la tarea **T2(R,T)**). Una variable a tierra (parámetro) se representará en cursiva (por ejemplo, R) mientras que las variables libres se mantienen con la fuente estándar (por ejemplo, P). La elección de usar elipsis grises para las tareas (como para **T1(R)**) en el plan parcial proviene de la necesidad de poder diferenciar el dominio del plan parcial en un vistazo. Finalmente cuando HTN tiene que hacer una elección, se crea un punto de retroceso con un conjunto de alternativas, para representarlos. Por lo tanto, usamos un cuadro adjunto a la tarea correspondiente con la siguiente información:

1. BPi el identificador del punto de retroceso (útil para comprender qué punto de retroceso se utilizará primero cuando retrocede).
2. El tipo de puntos de retroceso.
3. La cantidad de alternativas de retroceso disponibles.

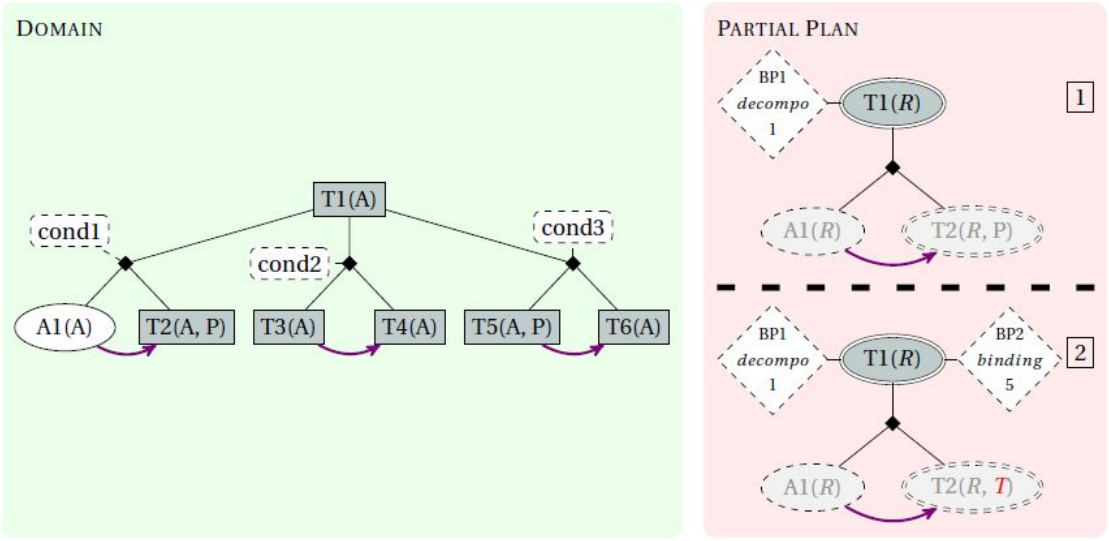


Figura 2.3 La figura muestra cómo HTN maneja un enlace variable: a T2 le faltaba un parámetro 'P'.

En el ejemplo decidimos que hay 6 valores posibles, por lo tanto, uno es elegido[5] y cinco alternativas alternativas se crean en BP2. Hay que tener en cuenta que las tareas **A1(R)** y **T2(R,T)** siguen en gris para representar el hecho aún no se han verificado: la unión ocurre antes de efectuar las tareas.

2.2.2 Enlace de variables

Cuando la HTN tiene que vincular una variable, puede ocurrir que haya varios valores posibles que coincidan con tipo de parámetro, así que de nuevo como HTN debe tomar una decisión, se crean algunos puntos de retroceso, véase la Figura 2.2. El tipo de parámetro necesario se representa en su ubicación: **T2(A,P)** necesita un valor de tipo Á'y otro de tipo 'P', en nuestro ejemplo elegimos respectivamente 'R'y 'T'. Mientras R es dada por la tarea a realizar, T podría haber tomado otro valor (asumimos otros cinco valores posibles). También se puede usar, en lugar del tipo, un nombre significativo como parámetro libre si el tipo es obvio, como por ejemplo **Goto(From, To)** obviamente necesita ubicaciones para los parámetros.

2.2.3 De orden parcial a orden total

Antes mencionamos que estamos usando una HTN de orden total y necesitamos linealizar todas las posibles órdenes de tareas al agregar algunas tareas al plan parcial. La figura 2.3 presenta este proceso de linealización. Es importante tener en cuenta que antes de descomponer **T2(R,T)** se realiza una prueba sobre las condiciones previas para **A1(R)**, de hecho, un vínculo causal implica que para aplicar **T2(R, T)** la acción **A1(R)** debe ser realizable. Además, cuando se encuentra una orden, todas las tareas nuevas deben ser probadas.

[5] La elección de usar 'T'como el valor del parámetro es destacada en rojo, como lo serán todos los cambios que no sean tan fáciles de ver.

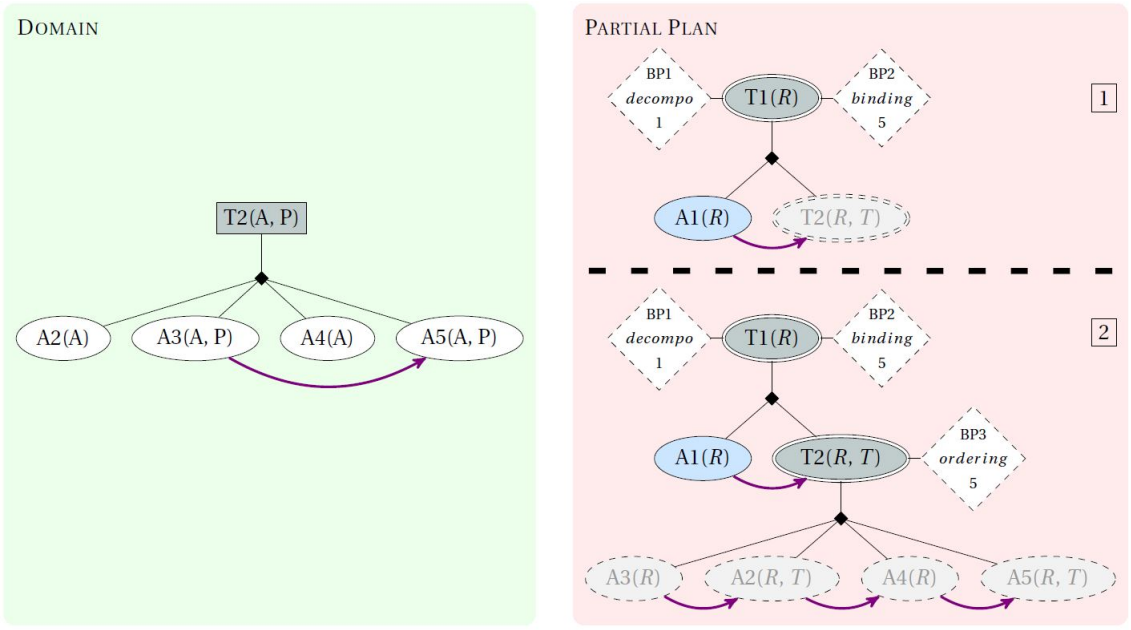


Figura 2.4 En este paso del ejemplo, la tarea T2(R, T) debe descomponerse (A1 (R) ha sido probado y es válido).

En el dominio, solo hay un método disponible, por lo que se elige directamente (y no retrocede) punto es necesario). Sin embargo, el método no da un orden definido para la acción A2, A3, A4 y A5, solo que A3 debe aparecer antes que A5, por lo que hay un total de seis posibles órdenes. Uno de los pedidos es usado y el proceso de planificación puede continuar: pruebe que las acciones en este orden son aplicables en la corriente estado (después de aplicar A1 (R)).

2.2.4 Mecanismo de retroceso

El último mecanismo HTN presentado es el retroceso, la razón por la que podemos necesitar un retroceso son:

1. Una acción en el plan parcial no se puede aplicar en el estado actual (es decir, sus requisitos previos no se cumplen).
2. Una tarea abstracta no tiene ningún método aplicable (es decir, los métodos correspondientes no tienen condiciones válidas).
3. Se ha encontrado un plan y la búsqueda se reinician para encontrar uno nuevo.

En la Figura 2.4, las condiciones previas de la acción **A4(R)** no son válidas, por lo que se desencadena un retroceso. Como HTN utiliza una búsqueda en profundidad, irá al último punto de retroceso insertado y usará una de sus alternativas para proceder. Si todos los pedidos no son válidos, por lo tanto, agotan BP3, el planificador retrocedería al anterior punto de retroceso cambiando el valor utilizado en el parámetro (retrocede al momento en el que damos un valor a la variable).

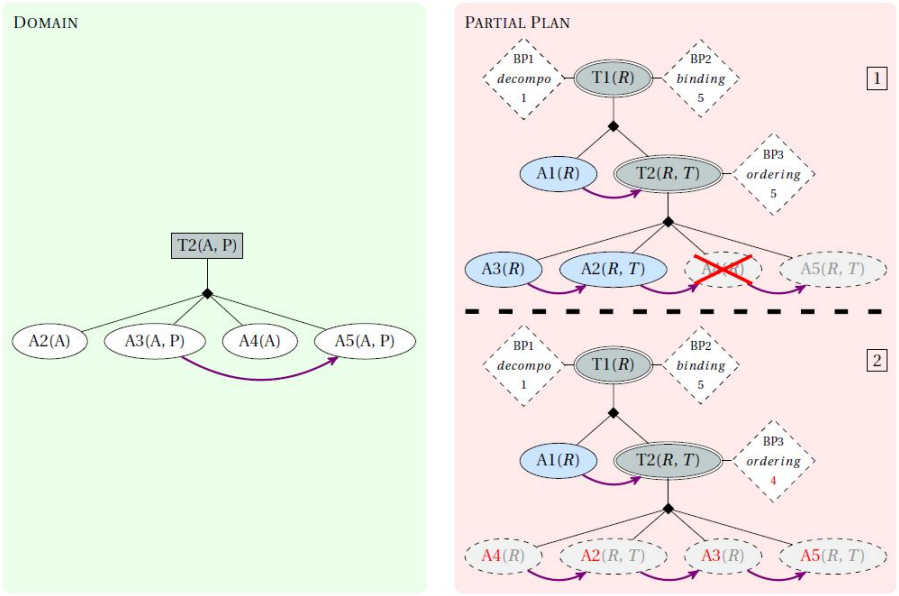


Figura 2.5 Las condiciones previas de la acción A4(R) no son válidas, por lo que se activa un retroceso. El último punto de retroceso es BP3 (solo quedan cuatro alternativas más) por lo que se intenta un nuevo orden de tareas y el proceso de planificación continúa..

3 Shop2 y Pyhop

3.1 Introducción

SHOP2 (Nau, Muñoz-Ávila, Cao, Lotem, y Mitchell, 2001),es un sistema de planificación de dominio independiente basado en planificación (HTN), el cual, ganó la competencia mundial de planificación de 2002 el premio al mejor rendimiento.En este capítulo describiremos brevemente el funcionamiento del planificador y algunas de las características que le permitió sobresalir sobre los demás planificadores.
Al igual que su predecesor, SHOP (Nau, Cao, y Muñoz-Avila, 1999), SHOP2 genera los pasos a seguir de cada plan en el mismo orden en el que estos se ejecutarán posteriormente, es decir, utiliza un tipo de búsqueda hacia adelante (Forward search). Esto reduce la complejidad de razonamiento, eliminando una gran cantidad de incógnitas del mundo en el que se desarrolla el sistema, haciendo fácil de incorporar gran potencia al sistema.
Al igual que SHOP, SHOP2 puede hacer deducciones axiomáticas, cálculos mixtos simbólicos / numéricos y llamadas a programas externos, pero sus capacidades van más allá del SHOP:

1. Permite que las tareas y subtareas se ordenen parcialmente, por lo tanto, los planes pueden intercalar subtareas de diferentes tareas. Esto a menudo hace que sea posible especificar el conocimiento del dominio de una manera más intuitiva de lo que era posible en SHOP.
2. Incorpora muchas funciones de Planning Domain Definition Language (PDDL),[1] como cuantificadores y efectos condicionales.
3. Si hay formas alternativas de satisfacer la condición previa de un método, SHOP2 puede ordenar las alternativas de acuerdo con un criterio especificado en la definición del método. Esto proporciona una forma conveniente para que el autor del dominio de planificación le indique a SHOP2 qué partes del espacio de búsqueda debe explorar primero. En principio, tal técnica podría ser utilizada con cualquier planificador de búsqueda hacia adelante.
4. Para que SHOP2 pueda manejar los dominios de planificación temporal, tenemos una forma de traducir los operadores PDDL temporales en operadores SHOP2 que mantienen la información de contabilidad para múltiples líneas temporales dentro de un mismo estado. Aunque, esta técnica podría usarse con cualquier planificador no temporal que tenga suficiente potencia.

Finalmente el sistema que utilizaremos en el ejemplo será en Pyhop, una versión de SHOP2 modificada para su uso en Python. A efectos prácticos ambas versiones son iguales, siendo la versión en Python algo mas potente.

[1] El Planning Domain Definition Language (PDDL) es un intento de estandarizar los lenguajes de planificación de inteligencias artificiales

3.2 Características de SHOP2

3.2.1 Elementos básicos para la descripción del dominio

Operadores

Los operadores son muy similares a los operadores PDDL: cada operador o tiene una cabecera **head(o)** que consiste en el nombre del operador y una lista de parámetros, una expresión de la condición previa **pre(o)** que indica lo que debe ser cierto en el estado actual para el operador a ser aplicable, y una lista de eliminación **del(o)** y una lista para añadir **add(o)**, dando los efectos negativos y positivos del operador. Al igual que en PDDL, las condiciones previas y los efectos pueden incluir conectores lógicos y cuantificadores. Los operadores también pueden hacer cálculos numéricos y asignaciones a variables locales. Al igual que en PDDL, no hay dos operadores que puedan tener el mismo nombre; por lo tanto, para cada tarea primitiva, todas las acciones aplicables son instancias del mismo operador.
Cada operador también tiene una expresión de costo opcional (el valor predeterminado es 1). Esta expresión puede ser arbitrariamente complicada y puede usar cualquiera de las variables que aparecen en la cabeza del operador y condiciones previas. El costo de un plan es la suma de los costos de las instancias del operador.

Código 3.1 Versión simplificada de un método.

```
(:method
  ; head
    (transport-person ?p ?c2)
  ; precondition
    (and
      (at ?p ?c1)
      (aircraft ?a)
      (at ?a ?c3)
      (different ?c1 ?c3))
  ; subtasks
    (:ordered
      (move-aircraft ?a ?c1)
      (board ?p ?a ?c1)
      (move-aircraft ?a ?c2)
      (debark ?p ?a ?c2)))
```

Axiomas

La condición previa de cada método u operador puede incluir entre otros, conjunciones, disyunciones, negaciones, cuantificadores universales y existenciales, implicaciones, cálculos numéricos y llamadas a funciones externas. Además, los axiomas pueden usarse para inferir en condiciones previas que no se afirman explícitamente en el estado actual. Los axiomas son versiones generalizadas de las cláusulas Horn[2], escritas en una sintaxis tipo Lisp: por ejemplo, **(: head tail)**, que indica que **head** es verdadero si **tail** es verdadero. La cola de la cláusula puede contener cualquier cosa que pueda aparecer en la condición previa de un operador o método. Como ejemplo, en el siguiente código:

Código 3.2 Versión simplificada de un axioma.

```
(:-
  ; head
    (enough-fuel ?plane ?current-position ?destination ?speed)
  ; tail
    (and (distance ?current-position ?destination ?dist)
    (fuel ?plane ?fuel-level)
    (fuel-burn ?speed ?rate)
    (eval (>= ?fuel-level (* ?rate ?dist)))))
```

[2] Una fórmula logica es una cláusulas Horn si es una cláusula con un literal positivo como máximo

el axioma mostrado dice que un avión tiene suficiente combustible para llegar a destino si se cumplen las siguientes condiciones: la distancia a recorrer es **?dist**, el nivel de combustible es **?fuel-level** de combustible, la velocidad de combustión es s **?rate**, y si **?fuel-level** no es menor que el producto de **?rate** y **?distance**. La última de estas condiciones se maneja usando una llamada de función externa, como se describe a continuación.

Código 3.3 Versión simplificada de un procedimiento de planificación de SHOP2.

```
procedure SHOP2(s, T, D)
  P = plan vacío
  T0 ← {t ∈ T : ninguna otra tarea en T está obligada a preceder t}
  bucle
    if T = ∅ entonces return P
    no determinísticamente elegir cualquier t ∈ T0
    if t es una tarea primitiva entonces
      A ← {(a, θ) : a es una instancia de un operador en D, θ es una
          sustitución que unifica {head(a), t}, y s satisface las condiciones
          previas de a}
    if A = ∅ devuelve un fallo
    no determinísticamente elegir un par (a, θ) ∈ A
    modifica s borrando del(a)y añadiendo add(a)
    enlaza a con P
    modifica T borrando t y aplicando θ
    T0 ← {t ∈ T : ninguna otra tarea en T está obligada a preceder t}
  else
    M ← {(m, θ) :m es una instancia de un método en D, θ unifica {head (m),
        t}, pre(m) es verdadero en s, y m y θ son tan generales como sea
            posible}
    if M = ∅ devuelve un fallo
    no determinísticamente elegir un par (m, θ) ∈ M
    modifica T borrando t, añadiendo sub(m), restringiendo cada tarea en sub(m)
      en sub (m) para preceder las tareas que t precedió y aplicar θ
    if sub(m) 6 = ∅ entonces
      T0 ← {t ∈ sub(m) : ninguna tarea en T está obligada a preceder t}
    else T0 ← {t ∈ T : ninguna tarea en T está obligada a preceder t}
  repetir
end SHOP2
```

3.2.2 Algoritmo de SHOP2

El código anterior muestra una versión simplificada del procedimiento de planificación SHOP2. Entre otros, los argumentos que podemos encontrar incluyen el estado inicial **s**, un conjunto parcialmente ordenado de tareas **T** y una descripción de dominio **D**.
Como mencionamos anteriormente, SHOP2 planifica las tareas en el mismo orden en que serán ejecutadas. Para hacer esto, no determinísticamente, se elige una tarea $\mathbf{t} \in \mathbf{T}$ que no tenga predecesores; **t** es la primera tarea en la que SHOP2 comenzará a trabajar. En este punto, hay dos casos:

- El primer caso es si **t** es primitivo, es decir, si **t** se puede lograr directamente usando una acción. En este caso, SHOP2 encuentra una acción que coincide con **t** y cuyas condiciones previas se cumplen en **s**, y aplica **a** a **s** (si tal acción no existe, entonces esta rama de la búsqueda falla).
- El segundo caso es donde **t** es compuesto, es decir, se debe aplicar un método para descomponer **t** en subtareas. En este caso, SHOP2 elige de forma no determinista un método **m** que descompondrá **t** en subtareas (si no existe dicho método, entonces esta rama de la búsqueda falla).

Si existe un plan de la solución que involucra **m**, entonces las acciones en **P** serán los nodos de un árbol de descomposición **DP** como se muestra en la Figura 3.1.

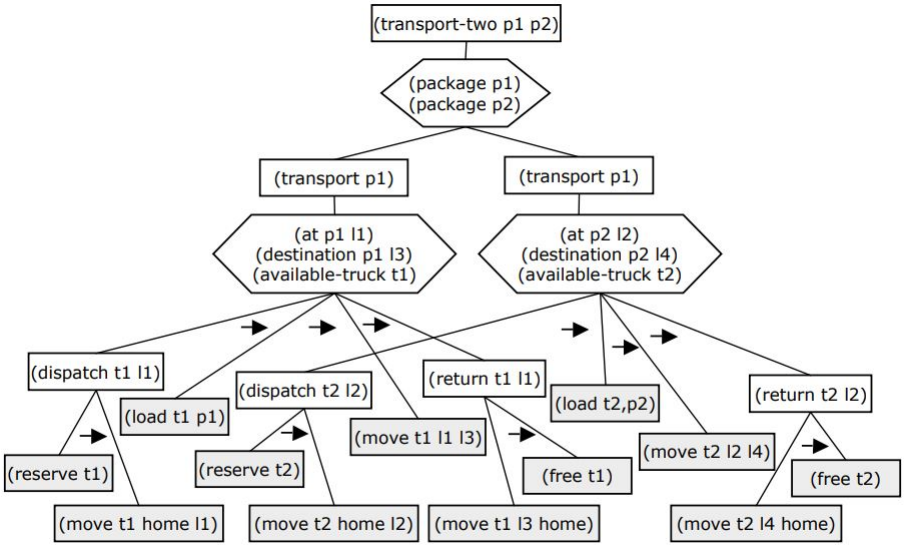


Figura 3.1 Ejemplo básico de un plan para transportar p1 y p2 desde un estado inicial (package p1), (at p1 l1), (destination p1 l3), (available-truck t1), (at t1 home), (package p2), (at p2 l2), (destination p2 l4), (available-truck t2), (at t2 home).

La fórmula de las condiciones previas **pre(m)** debe ser verdadera en el estado que precede inmediatamente a la primera acción **a** en **DP** que es un descendiente de **m**. Para asegurar que **pre(m)** sea verdadero en el estado correcto, SHOP2 necesita generar la rama más a la izquierda de **D** hasta abajo y evaluar **pre(m)** en el estado justo antes de **a**. Las últimas tres líneas del ciclo aseguran que esto sucederá, indicando a SHOP2 que si el método actual **m** tiene alguna subtarea, SHOP2 debe generar una de esas subtareas antes de generar cualquier otra subtarea en la red de tareas.
Por ejemplo, SHOP2 podría comenzar a generar el plan en la Figura 3.1 descomponiendo primero (transport-two p1 p2) en (transport p1) y (transport p2), y luego eligiendo de manera no determinista descomponer (transport p1) en {(dispatch t1 l1) , (pickup t1 p1), (move t1 l1 l3)}. Una vez hecho esto, SHOP2 debería descomponer (dispatch t1 l11) antes de descomponer (transport p2), para garantizar que (dispatch t1 l1) y (reserve t1) se produzca en el mismo estado del mundo en el que (dispatche t1) fue evaluado. El operador para (reserve t1) hace que t1 no esté disponible, lo que garantiza que cuando (transport p2) se descompone más adelante, la descomposición utilizará el camión t2 en lugar de t1.

3.3 Características adicionales

SHOP2 tiene varias características adicionales además de las básicas descritas anteriormente. Esta la sección describe los más importantes.

3.3.1 Clasificación de los enlaces de variables

Cuando SHOP2 evalúa la condición previa de un método, obtiene una lista de todos los posibles conjuntos de enlaces de variables que satisfacen la expresión en el estado actual. Cada conjunto de estos enlaces puede conducir a una rama diferente en el árbol de búsqueda de SHOP2. Esta elección no determinista se implementa en SHOP2 a través de una retroalimentación en profundidad. Para que SHOP2 encuentre una buena solución y la encuentre rápidamente, es importante decidir qué conjunto de enlaces debe probar primero.
Para este propósito, SHOP2 tiene un constructo "**sort-by**" que ordena la lista de enlaces de variables por un criterio específico. Esto es especialmente útil cuando el problema de planificación es un problema de optimización, por ejemplo, un problema en el que el objetivo es encontrar un plan que tenga el menor costo posible. Con la construcción **sort-by**, podemos escribir una función heurística para estimar el costo anticipado de cada conjunto de enlaces variables, y ordenar los conjuntos de enlaces variables de acuerdo con sus valores de función heurística para que SHOP2 intente primero el más prometedor.

Código 3.4 Ejemplo del uso de la función sort-by.

```
(:method
  ; head
    (transport-person ?p ?c2)
  ; precondition
    (:sort-by ?cost #'<
      (and (at ?p ?c1)
        (aircraft ?a)
        (at ?a ?c3)
        (different ?c1 ?c3)
        (cost-of ?a ?c3 ?c1 ?cost)))
  ; subtasks
    ((move-aircraft ?a ?c1)
      (board ?p ?a ?c1)
      (move-aircraft ?a ?c2)
      (debark ?p ?a ?c2)))
```

3.3.2 Optimizacion Branch-and-Bound

SHOP2 permite la opción de usar la optimización de ramas y enlaces para buscar el plan con el mínimo coste. Esta opción generalmente resulta en pasar tiempo de planificación adicional para buscar planes de calidad superior. Al usar la opción Branch-and-Bound, también se puede especificar un límite de tiempo para la búsqueda. Si la búsqueda lleva más tiempo que el establecido, SHOP2 finaliza la búsqueda y devuelve el mejor plan que ha encontrado hasta el momento.

3.4 Traducción de operadores PDDL

El procedimiento de planificación de SHOP2 puede ser sólido y completo en un gran conjunto de problemas de planificación, en el sentido de que si un conjunto de métodos y operadores es capaz de generar una solución para algún problema, se garantiza que el procedimiento de planificación generará una respuesta correcta (Nau et al., 2001). Sin embargo, aunque tal prueba nos dice que el algoritmo de planificación debería funcionar correctamente si la descripción del dominio es correcta, no nos dice si la descripción de nuestro dominio representa el mismo dominio de planificación de un conjunto de operadores de planificación PDDL.
Las descripciones de dominio producidas por el traductor el programa no es suficiente para una planificación eficiente con SHOP2: deben modificarse a mano para incluir el conocimiento del dominio. Sin embargo, el programa del traductor puede al menos proporcionar un punto de partida correcto.

3.5 Debugging

SHOP2 también incluye varias instalaciones de debugging. El más importante de estos es un mecanismo de seguimiento: uno puede decirle a SHOP2 que rastree cualquier conjunto de operadores, métodos y axiomas. Por ejemplo, en el Código 3.5, hemos dado nombres (a saber, Case1 y Case2) a las dos cláusulas diferentes de un método. Podemos decirle a SHOP2 que rastree cualquiera de estas cláusulas o ambas; SHOP2 imprimirá mensajes cada vez que entre y salga de una cláusula que se está rastreando.
Dependiendo de las opciones de seguimiento particulares que uno seleccione, los mensajes pueden incluir elementos tales como la lista de argumentos, el estado actual del mundo e información sobre si el operador, método o axioma tiene éxito o falla.

Código 3.5 Ejemplo del uso de la función debug.

```
(:method
    ; head
      (transport-person ?person ?destination)
  Case1 ; a label for use in debugging
    ; preconditions
      (and (at ?person ?current-position)
```

```
      (same ?current-position ?destination))
  ; subtasks
    ()
Case2 ; a label for use in debugging
  ; preconditions
    (and (at ?person ?current-position)
      (plane ?p))
  ; subtasks
    ((transport-with-plane ?person ?p ?destination)))
```

4 Ejemplo de un dominio DWR

4.1 Introducción del DWR

Este capítulo presenta el ejercicio realizado en este tfg. El dominio utilizado para esta HTN sera el de un robot portuario o DWR en sus siglas en inglés.
En nuestro caso el DWR constará de dos zonas en cuyo interior encontraremos una grúa y dos lugares para colocar los contenedores que se deberán transportar. Además, contamos con un camión cuya función será la de transportar un contenedor de una zona a otra.

La HTN será realizada en Pyhop y consta de 4 archivos:

- Dock worker operators: Este archivo contiene los operadores del DWR.
- Dock worker methods: De manera similar al anterior, en este guardamos los métodos uitlizados en el DWR.
- pyhop: Contiene la versión de Python de SHOP2.
- Ddock Worker robot: Archivo principal.

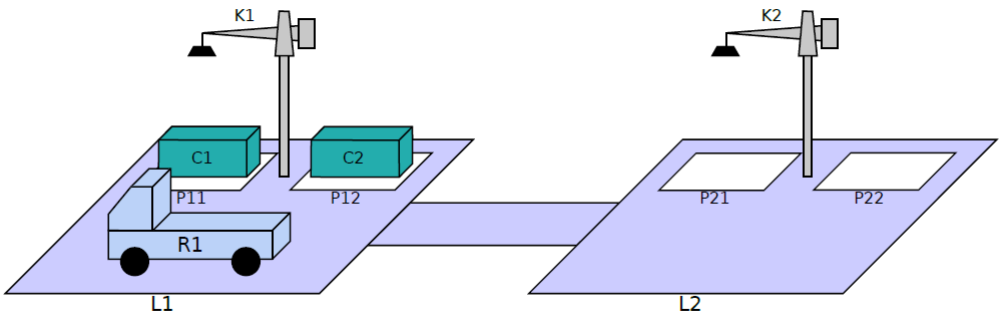


Figura 4.1 Ilustración de un robot portuario.

4.2 Código del DWR

4.2.1 Operadores

El código consta de 5 operadores:

- move: Mueve el camión de una zona a otra, las condiciones previas son: el tipo de **r** debe ser **ROBOT**, y se debe encontrar en la zona inicial.
- load: Utilizando una grúa, carga un contenedor al camión, las condiciones previas: el agente **k** debe ser tipo **Grúa** en la misma posición que el contenedor, debe sujetar el contenedor, tiene que ser tipo **ROBOT**, el camión en el que se va a cargar el contenedor debe de estar en la misma zona y debe de estar vacío. Tras esto, actualizamos la posición del camión.
- unload: Utilizando una grúa, descarga un contenedor del camión. Condiciones previas: el agente debe de ser tipo **Grúa**, tanto el camión como la grúa deben estar en la misma zona, el camión debe de tener un contenedor y la grúa debe de estar libre. Se actualiza la el estado de la grúa a vacío, el camión con el contenedor y la posición de éste.
- take: La grúa coge un contenedor de una de las posiciones del suelo.Condiciones previas: el agente debe ser tipo **Grúa**, la grúa y el contenedor deben de estar en la misma zona, la grúa debe de estar libre y el contenedor no debe de tener nada encima. Se actualiza la posición del contenedor, la posición anterior de éste y el estado de la grúa como **ocupado por la carga**.
- put: La grúa deja un contenedor en una de las posiciones del suelo.Condiciones previas: el agente debe ser tipo **Grúa**, la grúa y la futura posición del contenedor deben de estar en la misma zona, la grúa debe sujetar el contenedor y la futura posición de éste debe de estar libre. Se actualizan los valores de manera inversa a la acción anterior.

Código 4.1 Operadores del DWR.

```
import pyhop


def move(state, r, f, t):
    if state.type[r] == 'ROBOT' and state.pos[r] == f:
        state.pos[r] = t
        if state.carry[r] is not None:
            state.inside[state.carry[r]] = state.pos[r]
        return state
    else:
        return False


def load(state, k, r, c, l):
    if state.type[k] == 'CRANE' and state.pos[k] == l and state.carry[k] == c
        and
         state.type[r] == 'ROBOT' and\state.pos[r] == l and state.carry[r] is
             None:
        state.carry[k] = None
        state.carry[r] = c
        state.pos[c] = r
        return state
    else:
        return False


def unload(state, k, r, c, l):
```

```
    if state.type[k] == 'CRANE' and state.pos[k] == l and state.carry[k] is
        None \
            and state.type[r] == 'ROBOT' and state.pos[r] == l and state.carry[r
                ] == c:
        state.carry[k] = c
        state.carry[r] = None
        state.pos[c] = k
        return state
    else:
        return False


def take(state, k, c, l):
    if state.type[k] == 'CRANE' and state.pos[k] == l and state.carry[k] is
        None and\
                state.clear[c] is True and state.inside[c] == state.pos[k]:
        state.carry[k] = c
        state.carry[state.pos[c]] = None
        state.clear[state.pos[c]] = True
        state.pos[c] = k
        return state
    else:
        return False


def put(state, k, c, p, l):
    if state.type[k] == 'CRANE' and state.pos[k] == l and state.carry[k] == c
        and
        state.pos[p] == l:
        state.carry[k] = None
        if state.clear[p] is True:
            state.carry[p] = c
            state.pos[c] = p
            state.clear[p] = False
        elif state.clear[state.carry[p]] is True:
            state.carry[state.carry[p]] = c
            state.pos[c] = state.carry[p]
            state.clear[state.carry[p]] = False
        return state
    else:
        return False
print("""
*************************************
Dock operators have been imported successfully.
*************************************
""")
pyhop.declare_operators(move, load, unload, take, put)
```

4.2.2 Métodos del DWR

Los métodos utilizados en el ejercicio son:

- Setup1: Una grúa coge un contenedor y lo carga al camión. Se escogerá el contenedor que se encuentre en la posición deseada. Nos cercioramos de que en esa posición hay un contenedor sin nada encima y que el agente que vamos a utilizar es una grúa. Si todo es correcto, ejecutamos las acciones **take** y **load**.
- Finish1: Una grúa descarga un contenedor y lo deja en la posición deseada. Primero se comprueba que el contenedor se encuentra en la zona deseada, si es así comprobamos que tenemos los agentes correctos y ejecutamos las acciones **unload** y **put**.
- Transfer1: Este método mueve un contenedor de una zona a otra. Para ello comprobamos que el contenedor se encuentra en la zona inicial y comprobamos que contamos con el agente correcto, en este caso, el camión. si el camión se encuentra en la zona inicial ejecutamos las acciones **Setup**, **move** y **Finish**, en caso que el camión se encuentre en la otra zona ejecutamos primero una acción **move** adicional para traerlo de vuelta.
- Transfer2: Transfiere 2 contenedores. Para ellos vamos a usar un bucle **for** con el vector que contiene el número de contenedores, se indican los valores iniciales y finales y se ejecuta la acción **Transfer1** con cada contenedor.

Código 4.2 Métodos del DWR.

```
import pyhop

def setup1(state, c, l, r):
    if state.pos.get('K1') == l:
        k = 'K1'
    else:
        k = 'K2'
    if state.inside[c] == l and state.clear[c] is True and state.type[k] == '
        CRANE' and\
            state.carry[k] is None and state.pos[r] == l and state.type[r] == '
                ROBOT':
        return [('take', k, c, l), ('load', k, r, c, l)]
    else:
        return False


pyhop.declare_methods('Setup', setup1)


def finish1(state, c, l, r, p):
    if state.pos.get('K1') == l:
        k = 'K1'
    else:
        k = 'K2'
    if state.type[k] == 'CRANE' and state.type[r] == 'ROBOT' and state.pos[k]
        == l and
        state.carry[k] is None and state.pos[r] == l and state.carry[r] == c:
        return[('unload', k, r, c, l), ('put', k, c, p, l)]
    else:
        return False


pyhop.declare_methods('Finish', finish1)
```

```
def transfer1(state, c, l1, l2, r, p):
    if state.inside[c] == l1 and state.type[r] == 'ROBOT' and state.pos[r] == l
        1:
        return[('Setup', c, l1, r), ('move', r, l1, l2), ('Finish', c, l2, r, p)
            ]
    elif state.inside[c] == l1 and state.type[r] == 'ROBOT' and state.pos[r] ==
         l2:
        return[('move', r, l2, l1), ('Setup', c, l1, r), ('move', r, l1, l2),
            ('Finish', c, l2, r, p)]
    else:
        return False


pyhop.declare_methods('Transfer_one_container', transfer1)


def transfer2(state, goal):
    Result=[]
    r = 'R'
    for c in state.inside.keys():
        p = goal.pos[c]
        l1 = state.inside[c]
        l2 = goal.inside[c]
        if state.inside[c] == l2:
            continue
        elif state.inside[c] == l1:
            Result.append(('Transfer_one_container', c, l1, l2, r, p))
    return Result
pyhop.declare_methods('Transfer_two_containers', transfer2)


print("""
*************************************
Dock methods imported successfully.
*************************************
""")
```

4.2.3 Archivo principal del DWR

En este archivo importaremos los operadores y métodos anteriores, además definiremos el estado inicial del dominio y la meta que esperamos que alcance el robot. Enviaremos toda esta información al archivo que contiene el planificador Pyhop, que nos devolverá un vector con la solución que ha encontrado el robot. En el siguiente paso crearemos el diagrama de flujo dividiendo las solución proporcionada por Pyhop en acciones con las variables utilizadas, se creará una variable tipo **String** y añadiremos los agentes y posiciones utilizados, esta variable se guardará en un vector, creamos los nodos y las uniones del diagrama leyendo este vector mediante un bucle **for**. Por último, mostramos el resultado en el terminal y abrimos un PDF con el diagrama creado por Graphviz.

Código 4.3 DWR Main Page.

```
#Importando librerias
from __future__ import print_function
from pyhop import pyhop, State, Goal
from graphviz import Digraph
import os
#Importando los operadores y los métodos
import Dock_Worker_methods
import Dock_Worker_operators

#Seleccionamos el path de graphviz e imprimimos los operadores y los métodos
os.environ["PATH"] += os.pathsep + 'C:/ColocarPath/Graphviz/bin'
print('')
print_methods()
print('')
print_operators()

#Fijamos el estado inicial del sistema
state1 = State('state1')
state1.pos = {'R': 'L1', 'K1': 'L1', 'K2': 'L2', 'P11': 'L1', 'P12': 'L1', 'P
    21': 'L2', 'P22': 'L2', 'C1': 'P11',
             'C2': 'P12'}
state1.clear = {'C1': True, 'C2': True, 'P11': False, 'P12': False, 'P21': True
    , 'P22': True}
state1.type = {'R': 'ROBOT', 'K1': 'CRANE', 'K2': 'CRANE'}
state1.carry = {'R': None, 'K1': None, 'K2': None}
state1.inside = {'C1': 'L1','C2': 'L2'}
print_state(state1)
print('')

#Fijamos la meta que esperamos que alcance el robot
goal = Goal('goal')
goal.pos = {'R': 'L2', 'K1': 'L1', 'K2': 'L2', 'P11': 'L1', 'P12': 'L1', 'P21':
     'L2', 'P22': 'L2', 'C1': 'P21',
           'C2': 'P22'}
goal.clear = {'C1': True, 'C2': True, 'P11': False, 'P12': False, 'P21': True,
    'P22': True}
goal.type = {'R': 'ROBOT', 'K1': 'CRANE', 'K2': 'CRANE'}
goal.carry = {'R': None, 'K1': None, 'K2': None}
goal.inside = {'C1': 'L2', 'C2': 'L2'}
print_goal(goal)

#Buscamos con Pyhop las acciones necesarias para llegar a la meta
Result = pyhop(state1, [('Transfer_two_containers', goal)], verbose=1)
```

```
#En esta parte Crearemos el diagrama de flujo utilizado por el robot
dot = Digraph(comment='HTN Graph')
Name = []
N=1
#Creamos los nodos que se van a utilizar
if Result == False:
    print("""
          ***************************************
          Error estimating the correct methods
          ***************************************
          """)
else:
    for i in Result:
        A = i[0]+' '
        B = i[1]+' '
        C = i[2]+' '
        D = i[3]
        S = A + B + C + D
        Num = 'A' + str(N)
        dot.node(Num, S)
        N=N+1

    N = N - 1

#Creamos las uniones entre nodos del diagrama
    for i in range(1, N):
        S1 = 'A' + str(i)
        M = i + 1
        S2 = 'A' + str(M)
        dot.edge(S1, S2, constraint='true')

    print(dot.source)
    dot.render('test-output/round-table.gv', view=True)
```

A continuación se incluyen imágenes del terminal de Python y el PDF generado en la ejecución del programa:

```
**************************************
Dock methods imported successfully.
**************************************


**************************************
Dock operators have been imported successfully.
**************************************




** pyhop, verbose=1: **
   state = state1
   tasks = [('Transfer_two_containers', <pyhop.Goal object at 0x000001BA51EE85C0>)]
** result = [('take', 'K1', 'C1', 'L1'), ('load', 'K1', 'R', 'C1', 'L1'), ('move', 'R', 'L1', 'L2'), ('unload', 'K2', 'R',
'C1', 'L2'), ('put', 'K2', 'C1', 'P21', 'L2')]

// HTN Graph
digraph {
        A1 [label="take K1 C1 L1"]
        A2 [label="load K1 R C1"]
        A3 [label="move R L1 L2"]
        A4 [label="unload K2 R C1"]
        A5 [label="put K2 C1 P21"]
        A1 -> A2 [constraint=true]
        A2 -> A3 [constraint=true]
        A3 -> A4 [constraint=true]
        A4 -> A5 [constraint=true]
}
```

Figura 4.2 Terminal Python con la ejecución de la HTN.

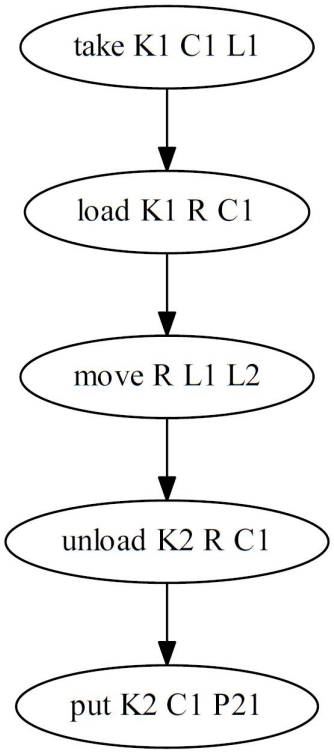


Figura 4.3 PDF generado por Graphviz.

5 Conclusiones y líneas de desarrollo futuras.

Para conocer los orígenes de la inteligencia artificial, nos tenemos que remontar a mas de 50 años atrás. En 1956 el primer laboratorio para el estudio de la inteligencia artificial se fundaría en la universidad de Dartmouth con la esperezan de inventar ordenadores tan inteligentes como los humanos, eventualmente, por las limitaciones del hardware se abandonó el proyecto.
A medida que ha paso el tiempo y los ordenadores han ganado potencia se ha empezado a comprobar el uso que este campo puede tener para el desarrollo de la tecnología y del ser humano.
Durante esta década se ha empezado a observar como esta rama de la computación ha explotado en importancia gracias a su rango de posibles usos, sobre todo en la capacidad del manejo de una cantidad inmensa de datos, de aprendizaje y de predicción de sucesos entre otros.
Se ha visto como grandes compañías, no solo tecnológicas, sino compañías del ámbito económico como bancos y auditorias o del sector servicios están desarrollando sus propios sistemas de AI.
En los siguientes gráficos podemos observar una mayor inversión en AI, un numero mayor de startups especializadas y la necesidad de especialistas en el campo.

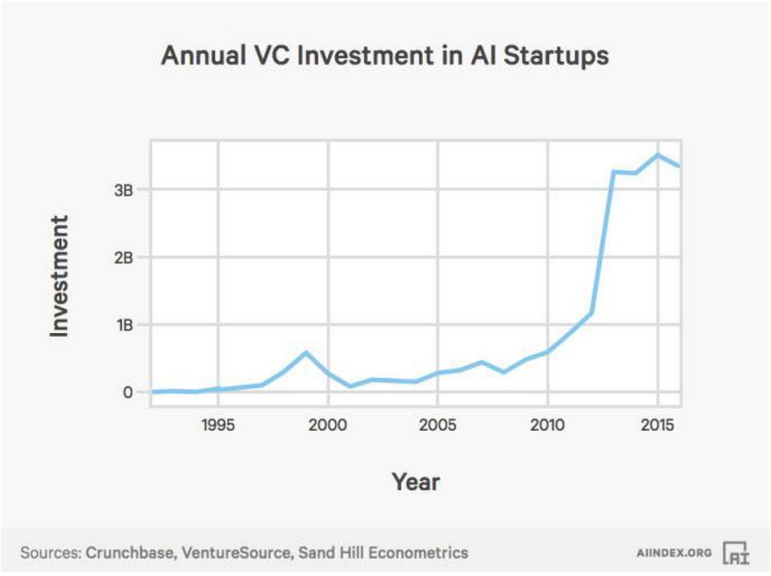


Figura 5.1 Inversion (en mil millones) en AI Startups.

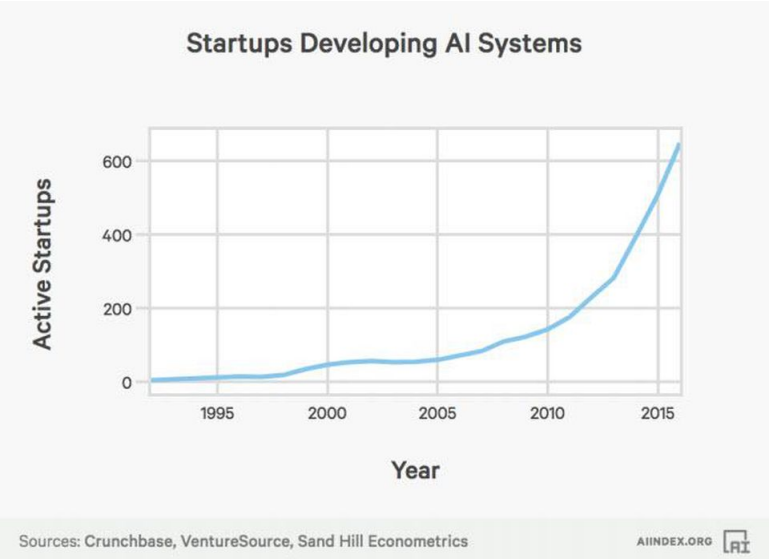


Figura 5.2 Numero de Startups desarrollando sistemas AI.

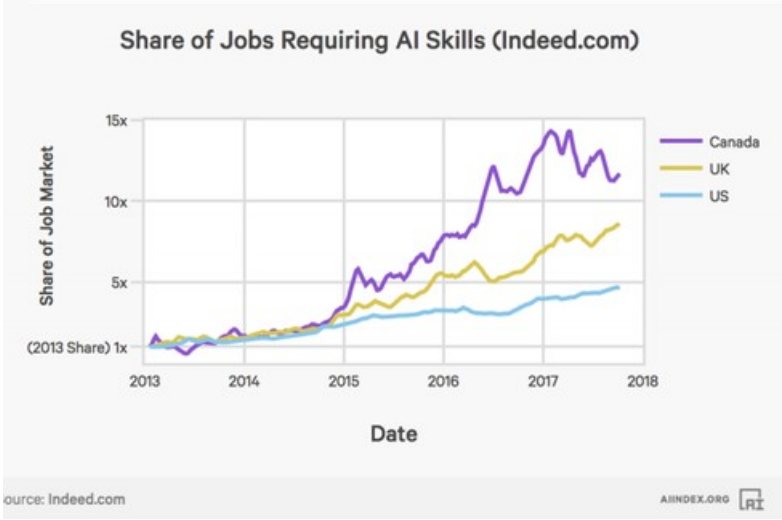


Figura 5.3 Cuota de trabajos que requieren habilidades con AI en el mercado.

Conceptos como **Machine Learning** y **Big Data** son realidades a día de hoy y su potencial no ha hecho más que comenzar.
En este momento nos preguntamos dónde se encuentran las HTN.
Realizando una observación de los estudios relacionados con la planificación de HTN a partir de un período de casi 40 años podemos ver que existe una escasa comprensión sobre el estado teórico, técnico y práctico actual de la planificación de HTN. El financiamiento de estos estudios es disperso durante largos periodos de tiempo, además, pese a existir HTN con implicaciones prácticas y teóricas interesantes, solo unos pocos han recibido atención mediática.

Además, los planificadores tienen una serie de deficiencias que se hacen más aparentes cuando son aplicadas en el mundo real, ya sea por la falta de previsión por parte del programador para crear todos las acciones y métodos necesarios, lo que da lugar a situaciones no previstas; o por el propio sistema de búsqueda, que es incapaz de razonar el resultado real de las acciones sobre el mundo físico, estos problemas pueden pasar en un principio desapercibidos, pero cuya acumulación puede lugar a grandes errores. También tenemos que hablar de las deficiencias de los sistemas de búsqueda, según el que se utilice, pueden llegar a dar un plan correcto, pero no el más eficiente. Una solución para este problema sería el uso de distintos tipos de búsqueda en un mismo planificador, por ejemplo, utilizar en un principio una búsqueda con un planificador plan-espacio (más eficiente cuantas más metas tenga el planificador) para después utilizar el planificador estado-espacio (más útil para la generación y comparación de metas) que es menos eficiente, pero mas seguro. El problema radicaría en un aumento exponencial del tiempo de cálculo del plan final.
Pese a todo, sabemos que las HTN funcionan mas que satisfactoriamente, un ejemplo de ello sería el robot Curiosity lleva más de 7 años sobre la superficie de marte, resistiendo tormentas de arena de semanas de duración y radiación solar letales, entre otros fenómenos que hacen imposible la vida en el planeta. Además, su representación jerárquica de problemas permite una mejor compresión de éste, y con una buena programación podemos obtener tiempos de planificación bastante razonables.
Como se ha comentado antes, este campo de la robótica esta poco explorado, por lo que las lineas de desarrollo son variadas. Se podría investigar nuevos algoritmos y motores de búsqueda, la capacidad de manejar eficientemente acciones y métodos complejosde las HTN se podría utilizar para controlar el uso de redes neuronales especializadas en distintas acciones como agentes para realizar trabajos con un enfoque más amplio, trabajos que las redes neuronales super-especializadas de hoy en día por si solas no son capaces de manejar.
La creación de un lenguaje descriptivo específico para las HTN también permitiría obtener metas complejas que no son posibles de realizar ahora mismo, como por ejemplo la declaración de un estado deseado, o condiciones adicionales como el mantenimiento de una serie de propiedades en una situación determinada, o simplemente queremos monitorizar o modificar el entorno. Por último, aunque parezca contrario a la planificación automática la creación de una interfaz cómoda para el usuario podría estimular al mismo utilizar más planificadores HTN para resolver problemas en el mundo real.

Índice de Figuras

Índice de Códigos

Siglas

AI Artificial Intelligence. 1

AIPS Astronomical Image Processing System. I

BP Backtrack Point. 5

DWR Dock Worker Robot. 1, 17

HTN Hierarchical Task Network. I, 1, 4–9, 11, 17, 25

PDDL Planning Domain Definition Language. 11, 12, 15

PSP Personal Software Process. 2

SHOP2 Simple Hierarchical Ordered Planner 2. I, 1, 11, 13–15

STRIPS Stanford Research Institute Problem Solver. 2