**TEMA 2: TECNICAS DE PLANIFICACION Y BUSQUEDA**

En este tema vamos a ver algunas de las técnicas de planificación partiendo de la programación PDDL que se ha visto en el tema anterior.

# Contenidos

1. SSS & PSP

1.1. Introducción

1.2. Clasificación de algoritmos: búsqueda en espacio de estados(SSS)

1.2.1 SSS mediante progresión

1.2.2 SSS mediante regresión

1.3. Clasificación de algoritmos: búsqueda en espacio de planes(PSP)

1.4. Técnicas de planificación:

2. Técnicas: Planificadores de Orden Total(TOP)

2.1. Definición

2.2. Plan prodigioso

3. Técnicas: Planificadores de Orden Parcial(POP)

3.1. Definición

3.2. Árbol POP

3.3. UCPOP

3.4. Planificadores POP

4. Técnicas: Planificación de Grafos(GP)

5. Técnicas: Red de Tareas Jerárquicas(HTN)

5.1. Definición

5.2. Ejemplo

5.3. HTN vs STRIPS

5.4. Planificadores HTN

6. Técnicas: Planificadores SAT

6.1. Introducción

6.2. Notación

6.3. Ejemplo

6.4. Planificadores SAT

7. Planificadores HSP

7.1. Introducción

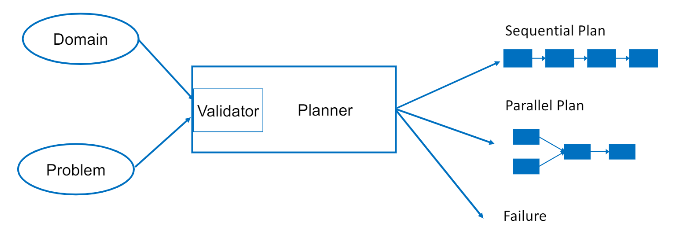
7.2. Heurísticas

7.3. FF

# 1. SSS & PSP

## 1.1. Introducción

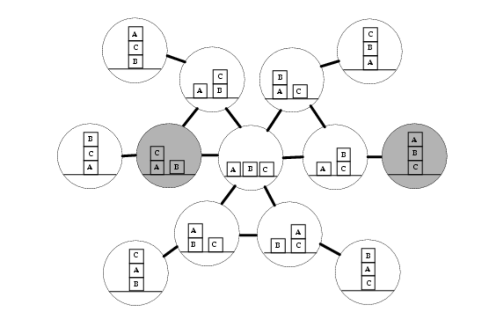
Casi todos los procedimientos de planificación son procedimientos de búsqueda. Como ya vimos en el tema anterior, cuando deseamos resolver un problema de planificación, disponemos de un fichero dominio.pddl y de un fichero problema.pddl. Estos ficheros antes d ser ejecutados, pasan por un validador que comprueba que no haya ningún tipo de error en el código. Una vez que sabemos que no hay errores, el planificador se encarga de generar una solución. Dicha solución o plan, puede ser de tres tipos, secuencial, paralela, y error.



Cuando el planificador nos proporciona un plan secuencial, las tareas se realizan una detrás de otra, mientras que, si nos proporciona un plan paralelo, varias tareas pueden hacerse al mismo tiempo. Por último, el planificador puede que no encuentre una solución o bien porque nuestro problema está mal descrito o bien porque el planificador no soporta la versión de pddl que estamos empleando.

## 1.2. Clasificación de algoritmos: búsqueda en espacio de estados

En la búsqueda en espacio de estados, cada nodo representa un estado del mundo y el plan es una camino a través del dicho espacio. En el plan de estados, cada nodo es un conjunto de operadores parcialmente instanciados con algunas restricciones. Se trata de imponer cada vez más y más restricciones hasta encontrar un plan.

La madera más sencilla de construir un planificador es convertirlo en un problema de búsqueda a través de un espacio de estados. Donde como ya se ha indicado antes, cada nodo en el plan representa un estado del mundo y los arcos que conectan cada nodo son acciones. Una vez que el problema se ha convertido, podemos aplicar cualquier algoritmo estudiado.

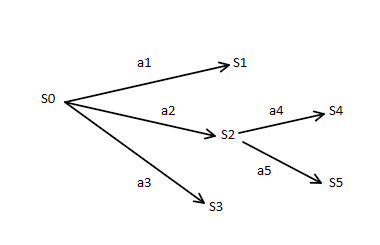
De manera incremental, podemos ir generando un conjunto de estados accesible desde el estado inicial, mediante una secuencia de acciones, mediante **progresión**. Los estados alcanzados en la secuencia pueden ser calculados comprobando si la meta ha sido alcanzada o si las precondiciones se han satisfecho.

Pero en vez de mirar hacia delante desde el estado inicial, podemos volver hacia atrás desde las metas encontrando así una solución, mediante **regresion**.

Una vez que disponemos de nuestro algoritmo, debemos tener en cuenta varias consideraciones. En primer lugar, debemos saber si dicho algoritmo es robusto, es decir retorna una solución valida. En segundo lugar, si es completo, es decir si la meta que se propone puede ser alcanzada.

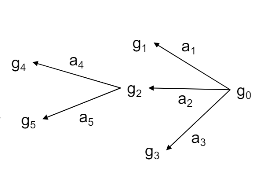
## 1.2.1. Búsqueda en espacio de estados mediante progresión

Para cualquier plan devuelto por alguna ruta no determinista, este plan está garantizado que será una solución. La búsqueda mediante progresión será completa si existe un camino que llegue a la solución desde el estado inicial. Algunas implementaciones de la búsqueda en espacio de estados mediante progresión son: BEA, BEEP, A\*…

BEA y A\* son algoritmos óptimos y completos, pero no se suelen utilizar mucho en la práctica puesto que requieren mucha memoria. La memoria requerida es exponencial al tamaño de la solución. En la practica el algoritmo más utilizado es BEEP debido a que la memoria requerida es lineal al tamaño de la solución, en general es óptimo, pero no completo.

La búsqueda en espacio de estados mediante progresión puede tener un factor alto de ramificación, esto es peligroso puesto que implementaciones deterministas pueden malgastar mucho tiempo en acciones irrelevantes. Es necesario una buena función heurística y/o un proceso de poda.

## 1.2.1. Búsqueda en espacio de estados mediante regresion

En la búsqueda mediante progresión, comenzamos desde el estado inicial y media te una serie de estados tratamos de llegar a la meta. En la búsqueda mediante regresion, comenzamos desde la meta y vamos hacia atrás en busca del estado inicial.

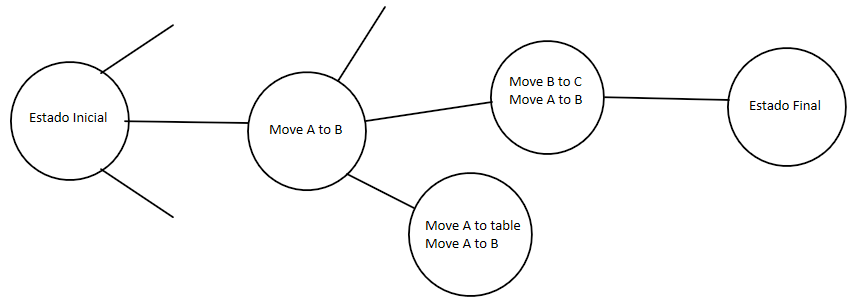
La búsqueda por regresion también puede tener un factor alto de ramificación, al igual que antes, implementaciones deterministas pueden malgastar mucho tiempo. STRIPS usa esta búsqueda.

## Lifting

Podemos reducir ese factor de ramificación de la búsqueda hacia detrás si inicialmente instanciamos los operadores. El factor de ramificación seguirá siendo alto, pero mucho menos que antes. Puede que existe un plan que no tenga solución, en el que no podamos llegar al estado inicial.

## 1.3. Clasificación de algoritmos: búsqueda en espacio de planes

En 1974 se construyó el planificador NOAH, la búsqueda conducía a través de un espacio de planes. En dicha búsqueda cada nodo en el plan/árbol representa un plan parcial y cada arista representa operaciones de dicho plan. Además, en el plan puede haber un conjunto de restricciones.

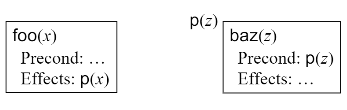
El nodo inicial del plan es NULL y el nodo final representa el plan solución, la meta. Mientras que la búsqueda en espacio de estados retorna el camino desde el estado inicial hasta la meta, en la búsqueda en espacio de planes, la meta es la solución.

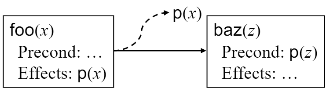
En cuanto a las restricciones, existen distintos tipos que son los siguientes:

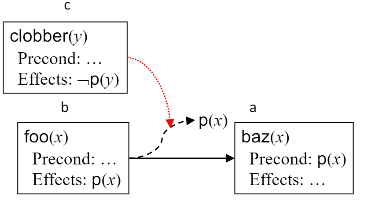
Restricciones de precedencia: A debe preceder a B.

Restricciones de unión: pueden ser restricciones de desigualdad(≠) o de igualdad(=)

Enlaces causales: usar las acción A para establecer las precondiciones que necesita la acción B.

Existen ciertos problemas, algunos de ellos son los **flaws** y los **threat**. Los flaws son casos donde parecer metas abiertas, es decir, acciones que tienes precondiciones que no han sido establecidas. Uno de esos caso son lo que se denomina como flaws. Una manera de resolver este problema es usar una determinada acción *b* que podemos insertar nosotros en el plan, para establecer una acción *p*.



Un threat o amenaza, es una interacción para eliminar una condición. Por ejemplo, una determinada acción *a* establecer una precondición de la acción *b*. Otra acción *c* es capaz de eliminar *p*. Para resolver este problema, creamos una restricción que impida a *c* eliminar *p*, ¿Cómo?, hacemos que *b* preceda a *c*, y que *c* preceda a *a*.

## 1.3. Técnicas de planificación

A continuación, y con más profundidad en los siguientes apartados, se explicarán una serie de técnicas de planificación. Alguna de ellas son las siguientes:

TO: La solución es una secuencia totalmente ordenada de acciones(estados/planes).

POP: Búsqueda en espacio de planes. Implementa lo que se denomina como “leats Commited approach”.

HTN: redes de tareas y restricciones.

Grafo-base(GP): la estructura de búsqueda es un Grafo de Planificación.

SAT: toma un problema como entrada, determina su tamaño y genera una serie de cláusulas preposicionales.

HSP: transforma problemas de planificación en problemas de heurística.

# 2. Planificadores de Orden Total

## 2.1. Definición

Como ya hemos dicho anteriormente, la solución de todo planificador es una secuencias de acciones que están ordenadas. Un plan es válido si las precondiciones de cada operador se satisfacen antes de la ejecución de dicho operador. A diferencia de algoritmos que veremos más adelantes, introducir restricciones so es necesario. Dado un determinado problema, la mayoría de planificadores empiezan con un plan vacío y lo van modificando a medida que se encuentra la solución.

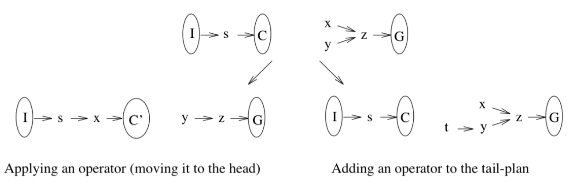
Un plan puede ser modificado insertando nuevos operadores, introduciendo restricciones o instanciando variables en la descripción de un determinado operador.

Durante la ejecución del plan, mientras que se determina la solución, el plan se considera incompleto. Cada plan incompleto puede ser visto como un nodo en la búsqueda del espacio de planes del algoritmo. Modificar un plan incompleto consiste con expandir un nodo.

El factor de ramificación se determina por el número de posibles modificaciones del plan.

La distinción de planificadores de orden total debe estar separada de la distinción de los planificadores SSS y PSP. Como ya hemos vito en el apartado anterior, en SSS cada estado de la búsqueda corresponde con un estado en el mundo, mientras que en PSP cada estado de la búsqueda corresponde con un plan. Generalmente TO se corresponde con SSS.

## 2.2. Prodigy planner

Un plan incompleto consiste de dos partes, una cabeza y una cola. La cola del plan se construye mediante un algoritmo de regresion, que comienza en la meta y añade operadores uno por uno para conseguir que dichos operadores satisfagan sus precondiciones, los operadores que no se satisfacen sus precondiciones se denominan metas pendientes .La cabeza del plan aplica el operador en la cola del plan.

# 3. Planificadores de Orden Parcial

## 3.1. Definición

Los planificadores de orden parcial se basan en la planificación en espacio de planes. Un plan parcial se define como P = (A, O, L, OC, UL), es decir, está compuesto por acciones, restricciones ordenadas, enlaces causales, condiciones abiertas(flaws), enlaces inseguros.

Los planificadores de orden parcial realizan la búsqueda mediante regresion, donde cada nodo está compuesto por acciones parcialmente instanciadas y un conjunto de restricciones. El proceso para si la solución se encuentra.

El algoritmo de planificación implementa leats commited technique. Solo las acciones esenciales son guardadas puesto que aquellas que no son necesarias no se comitean. La estructura del enlace causal es la responsable de almacenarlas en tres campos, productos, consumidor y proposición.

Algunos ejemplos de planificadores de orden parcial son UCPOP, Cassandra, ZENO, VHPOP.

## 3.2. Árbol POP

El plan inicial es creado por la descripción del estado inicial y la descripción de la meta. Dicho plan inicial se crea en dos pseudo pasos:

Comienzo:

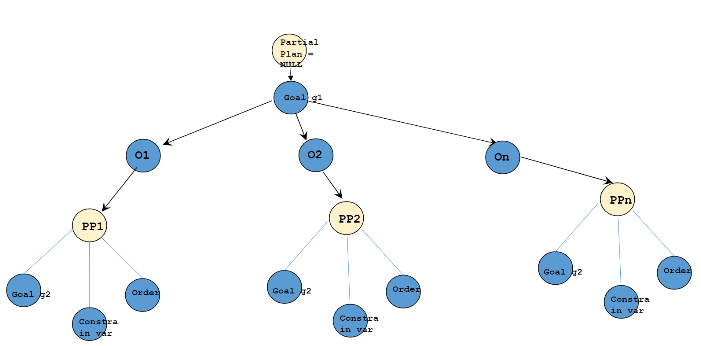
Precondición: Ninguna

Efecto: Todas los literales positivos se definen en el estado inicial

Final:

Precondición: Los literales definen conjuntamente la meta a alcanzar

Efecto: Ninguno



## 3.3. Planificadores de Orden Parcial

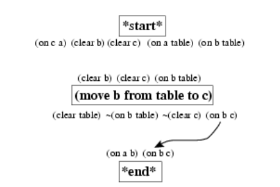
UCPOP

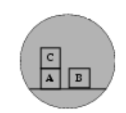
Cassandra

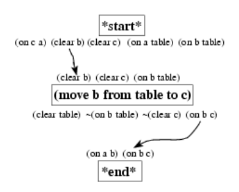
ZENO

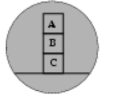
VHPOP

## 3.4. Ejemplo: UCPOP

Partimos del ejemplo de blocksworld donde disponemos de tres cajas iniciales y tenemos que disponerlas de una determina forma.



Como podemos observar, se ha añadido un enlace causal entre la acción “mover b de la mesa a c” y la meta. Después de ello, unimos podemos unir el estado inicial con dicha acción y encontrar un camino.



Una vez añadido el ultimo enlace causal de la acción (clear b), el plan contiene dos enlaces causales {(clear c) (on B table) (on A B)}

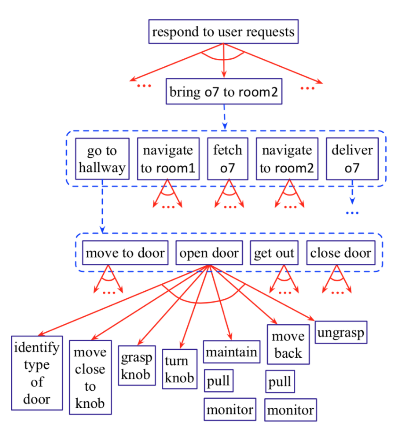
# 4. GraphPlan

# 5. Red de Tareas Jerárquicas

## 5.1. Definición

Mas conocido como reducción de tareas o Hierarquical Task Network, HTN es un algoritmo en el que tanto problemas como operaciones se disponen en una red o conjunto de tareas llamadas acciones, que corresponden con las transición entre estados. Las tareas de mas alto nivel se reducen a áreas de nivel bajo. El algoritmo se encarga de ir expandiendo de forma iterativa tareas y resolver conflictos hasta que se encuentre un plan libre de conflictos.

## 5.2. Ejemplo

Vamos a imaginar que queremos llevar el objeto 7 a la habitación 2. Puede que queramos especificar la forma apropiada de realizar la tarea de la siguiente forma:

“Tenemos un robot r que busque el objeto 7 y que vaya a la habitación 2 y deje el objeto 7”. Un buen conjunto de métodos puede permitir que un planificador HTN funcione bien en un determinado punto de referencia.

Tenemos una tarea principal, y la subdividimos en subtareas mas simples. Cada una de las subtareas se divide en más subtareas que a su vez se dividen en subtareas. De esta forma al final disponemos de tareas simples que podemos tratar.

## 5.3. HTN vs STRIPS

El objetivo de STRIPS es encontrar un conjunto de acciones ordenadas desde el estado inicial hasta la meta. Encuentra los operadores apropiados que disponen de los efectos indicados para cumplir las precondiciones, a estos operadores se les denomina submetas.

Por otro lado HTN busca redes de tareas que incluyan otras cosas además de metas. El plan se descompone en tareas y resolución de conflictos. Los métodos deben contener todos los posibles formas de obtener la tarea, cosa que es más tediosa.

## 5.4. Planificadores HTN

NONLIN

O-PLAN

DESIVER

SIPE

NMRA

SHOP

TALPlanner

# 6. Planificadores SAT

## 6.1. Introducción

Los planificadores SAT surgen a partir de la siguiente idea. Dado un problema proposicional de satisfacción como el siguiente:

*(P v Q) ^ (~Q v R v S) ^ (~R v ~P)*

Se quiere saber si existe una modelo que haga que la formula sea cierta. Este fue el primer problema NP-completo. Los algoritmos eran conocidos por resolver conjuntos de problemas en un tiempo polinomial. Por lo tanto se desea trasladar problemas clásicos de planificación en problemas de satisfacción y resolverlos de esa manera.

## 6.2. Notación

## 6.2.1. Proposiciones

Para problemas se satisfacción de restricciones necesitamos emplear la lógica proposicional para ser capaces de codificar proposiciones. Por ejemplo, para un conjunto teórico de planificación, los codificaríamos como se muestra a continuación:

* Predicado: at(r1, loc1)
* Proposición: at-r1-loc1

Para problemas de satisfacción de restricciones, hacemos el mismo proceso, pero no nos molestamos en reescribir lo mismo, simplemente usamos at(r1, loc1) para la proposición.

## 6.2.1. Fluents

Una proposición en in predicado determinado es verdaderas para cierto estado en particular. Por ejemplo at(r1, loc1, i) es un fluent que en verdadero si at(r1, loc1) está en i. Se usará Li para denotar el fluent de cada literal L en un estado determinado. L = at(r1, loc1), Li = at(r1, loc1, i).

## 6.3. Ejemplo

Disponemos de un robot r1 y dos localizaciones adyacentes l1 y l2. Además se dispone de un operador move que permite al robot desplazarse de una localización otra. El estado inicial se encuentra el robot at(r1, l1), y el estado meta que queremos alcanzar es at(r1, l2). La formula que describe el estado inicial se puede codificar como una conjunción de proposiciones de la siguiente manera:

*at(r1, l1)*

*at(r1, l1, 0) ^ at(r1, l2, 0)*

La fórmula que describe el estado meta es la siguiente:

*at(r1, l2)*

*at(r1, l2, 1) ^ at(r1, l1, 1)*

Para cada acción a en A y cada i = 1, 2, 3, …, n la formula a describe que cambia desde el momento inicial hasta n-1.

Una vez se ha definido el estado inicial y el estado final, hay que definir el operador mover. El operador GoTo(r, l, l’), tiene una serie de precondiciones y una serie de efectos.

Operador: *GoTo(r, l, l’)*

Precondiciones: *at(r, l)*

Efectos: *at(r, l’), ~at(r, l)*

La precondición es true en t = 0 y los efectos serán true en t = 1. Ahora se codifica el operador:

*GoTo(r1, l1, l2, 0) => at(r1, l1, 0) ^ at(r1, l2, 1) ^ ~at(r1, l1, 1)*

*GoTo(r1, l2, l1, 0) => at(r1, l2, 0) ^ at(r1, l1, 1) ^ ~at(r1, l2, 1)*

*GoTo(r1, l1, l1, 0) => at(r1, l1, 0) ^ at(r1, l1, 1) ^ ~at(r1, l2, 1)*

*GoTo(r1, l2, l2, 0) => at(r1, l2, 0) ^ at(r1, l2, 1) ^ ~at(r1, l1, 1)*

Vamos a suponer que encontramos una asignación de valores que satisfacen el predicado. Esto significa que P tiene una solución de tamaño n. Por lo tanto para encontrar valores de verdad que satisfagan nuestra fórmula solo debemos emplear algoritmos de satisfacción de restricciones. Por ejemplo el algoritmo DPLL primero necesita transformar la formula a forma normal conjuntiva y después dividir cada conjunción en clausulas de la siguiente forma:

*D ^ (~D v A v ~B) ^ (~D v ~A v ~B) ^ (~D v ~A v B) ^ A*

*{{D}, {~D, A, ~B}, {~D, ~A, ~B}, {~D, ~A, B}, {A}}*

En algunos casos especiales puede que la formula dada sea un conjunto vacío o que el conjunto vacío forme parte de la formula, en ese caso siempre será true.

## 6.4. Planificadores SAT

SATPLAN: construye un grafo y lo transforma en clausulas, luego usa SAT para encontrar la solución, si no la encuentra el tamaño aumenta.

BLACKBOX: combina GraphPlan y SATPLAN

LPSAT: emplea un algoritmo de backtracking aleatorio con determinados formalismos que combinan la lógica proposicional con un conjunto de restricciones.

LPG

El definitiva la filosofía del algoritmo SAT es trasladar un problema de planificación en una conjunción de disyunciones(FNC), genera un conju8nto de cláusulas que determinen la formula es satisfacible o no. Algunos algoritmos que trabajan con proposiciones lógicas son DPLL, WALSAT, LMTS-style…

# 7. Planificadores HSP

## 7.1. Introducción

Ningún algoritmo de búsqueda hacia delante, ni ningún algoritmo de backtracking es eficiente sin una buena función heurística. Una heurística es un metodo que decide entre distintas alternativas, la mas efectiva que encuentre la meta.

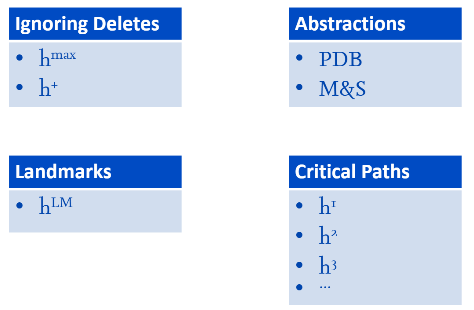
HSP se encarga de transformar problemas de planificación en problemas de heurística extrayendo funciones heurísticas en vez de introducirlas a mano. Algunos de los problemas de HSP es que el num3ro de nodos a explorar es demasiado elevado, además de que n cada paso se calcula la heurística.

## 7.2. Heurísticas

Ignoring deletes: simplifica el mundo asumiendo que algo que nunca fue True, será True en el futuro.

Landmark: algo que debe ser True en algún momento del plan.

Abstraction: ignora parte de la descripción del problema.

Critical Pahts: Se asemeja con subproblemas con un tamaño modificado. Esta parametrizado.

Si el algoritmo siempre encuentra una solución óptima, entonces la función heurística será admisible. Puede ser de dominio especifico o de dominio independiente.

## 7.2.1. Max-cost and additive cost