TÉCNICAS DE LOS SISTEMAS INTELIGENTES

Práctica2: Planificación Clásica.
Sesion8. El lenguaje PDDL

Temporización Prácticas 2 y 3

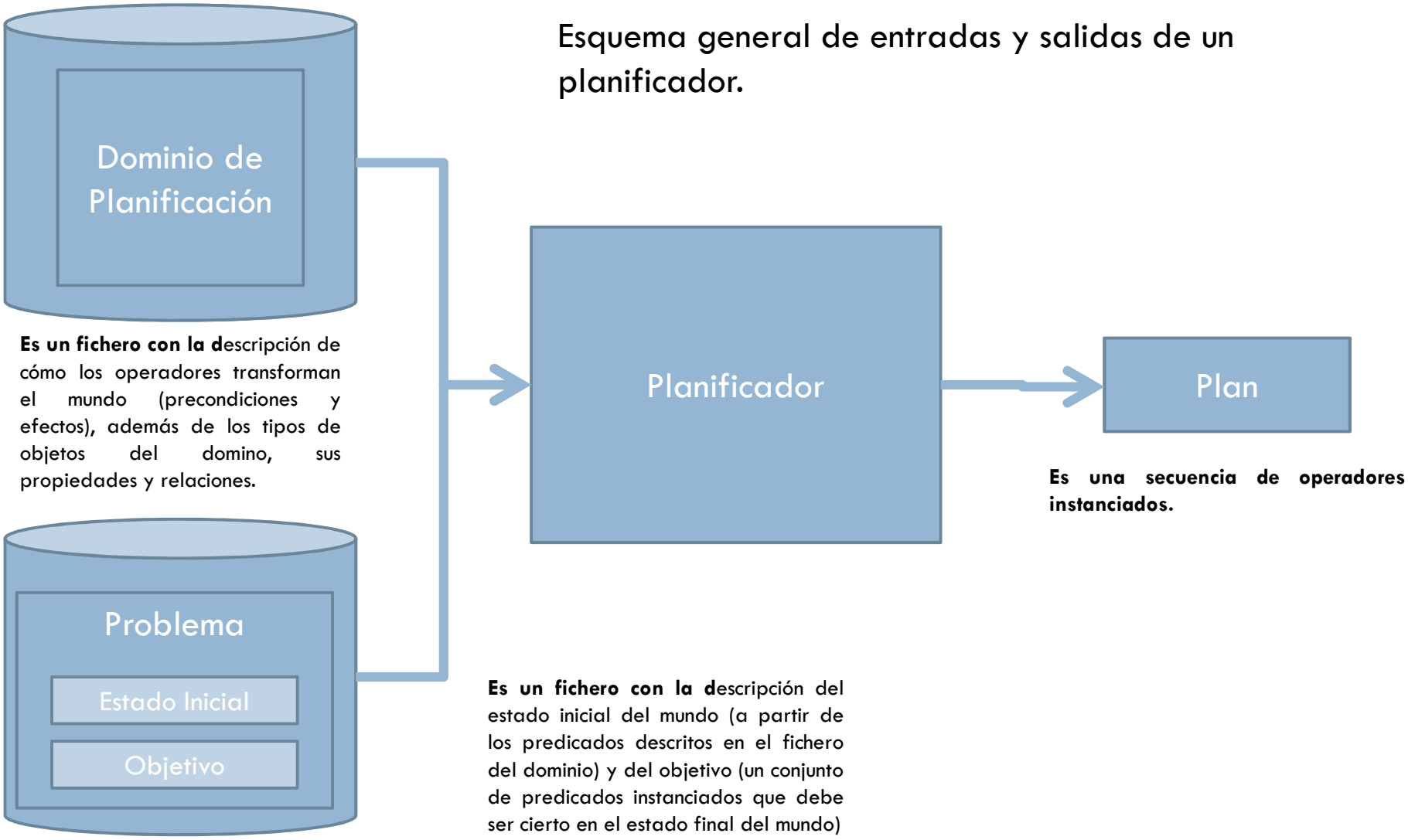
Plan Sesiones

- *Sesión 8: 8, 11, 12 abril:*
  - Introducción a la Planificación Clásica con PDDL
- *Sesión 9: …, 25 , 26 abril:*
  - Seguimiento.
- *Sesión 10: 29 abril, 2 ,3 mayo*
  - Seguimiento
- *Sesión 11: 6, .., 10 mayo (9 mayo día de la Escuela)*
  - Seguimiento
- **Entrega P2:** 12 de mayo a las 14:00
- *Sesión12: 13, 16, 17 mayo*
  - Seminario 3: Planificación HTN con HPDL.
- *Sesión13: 20, 23, 24 mayo*
  - Seguimiento
- *Sesión14: 27,30, 31 mayo*
  - Seguimiento.
- **Entrega P3**: 1 de Junio hasta las 14:00

Objetivos

- Sesión 8 a 11
  - Conocer el lenguaje PDDL como el estándar para la descripción de dominios y problemas de planificación.
  - Definir dominios y problemas de planificación.
  - Usar el planificador FF (Fast Forward) para resolver problemas de planificación descritos en PDDL.
  - Entrega Práctica 2 (12 Mayo)
- Sesión 12 a 14
  - Conocer el lenguaje HPDL como un lenguaje para describir problemas de planificación jerárquica de tareas (HTN: Hierarchical Task Networks)
  - Definir dominios y problemas de planificación jerárquica
  - Usar el planificador lactive Planner (desarrollado por nuestro grupo de investigación y comercializado por lactive) para resolver problemas de planificación jerárquica.
  - Entrega Práctica 3 (1 de Junio)

Esquema general de entradas y salidas de un planificador.



Es un fichero con la descripción de cómo los operadores transforman el mundo (precondiciones y efectos), además de los tipos de objetos del domino, sus propiedades y relaciones.

Es un fichero con la descripción del estado inicial del mundo (a partir de los predicados descritos en el fichero del dominio) y del objetivo (un conjunto de predicados instanciados que debe ser cierto en el estado final del mundo)

Es una secuencia de operadores instanciados.

- PDDL
  - Planning Domain Definition Language
  - https://en.wikipedia.org/wiki/Planning_Domain_Definition_Language
- Lenguaje estándar para la representación de dominios de planificación clásicos
  - Conocimiento completo : todas las propiedades y relaciones entre los objetos o bien se conocen inicialmente o bien pueden conocerse durante la planificación
    - Hipótesis del mundo cerrado: los hechos no especificados en el estado inicial son falsos.
  - Acciones deterministas
    - Los efectos de las acciones son conocidos a priori
  - Cambios en el mundo sólo producidos por la ejecución de acciones.
    - No se consideran eventos exógenos.
- Referencias sobre PDDL
  - Writing Planning Domains and Problems (breve introducción) http://users.cecs.anu.edu.au/~patrik/pddlman/writing.html
  - Online PDDL Reference (recomendado!!) https://nergmada.github.io/pddl-reference/
  - Otros recursos didácticos sobre PDDL y planificación http://education.planning.domains/

Componentes de PDDL

- **Objetos**: cosas del mundo que nos interesa representar
- **Predicados**: propiedades o relaciones de objetos de interés (pueden ser ciertos o falsos)
- **Acciones/Operadores**: Maneras de cambiar el estado del mundo.
- **Estado inicial**: El estado del mundo a partir del que empezamos a planificar.
- **Objetivo**: Cosas que queremos que sean ciertas cuando se ejecute el plan.

- Una tarea de planificación en PDDL se especifica en dos ficheros:
  - Un fichero de dominio:
    - especificación de objetos, predicados y acciones
  - Un fichero de problema
    - Objetos (constantes), estado inicial y especificación del objetivo.

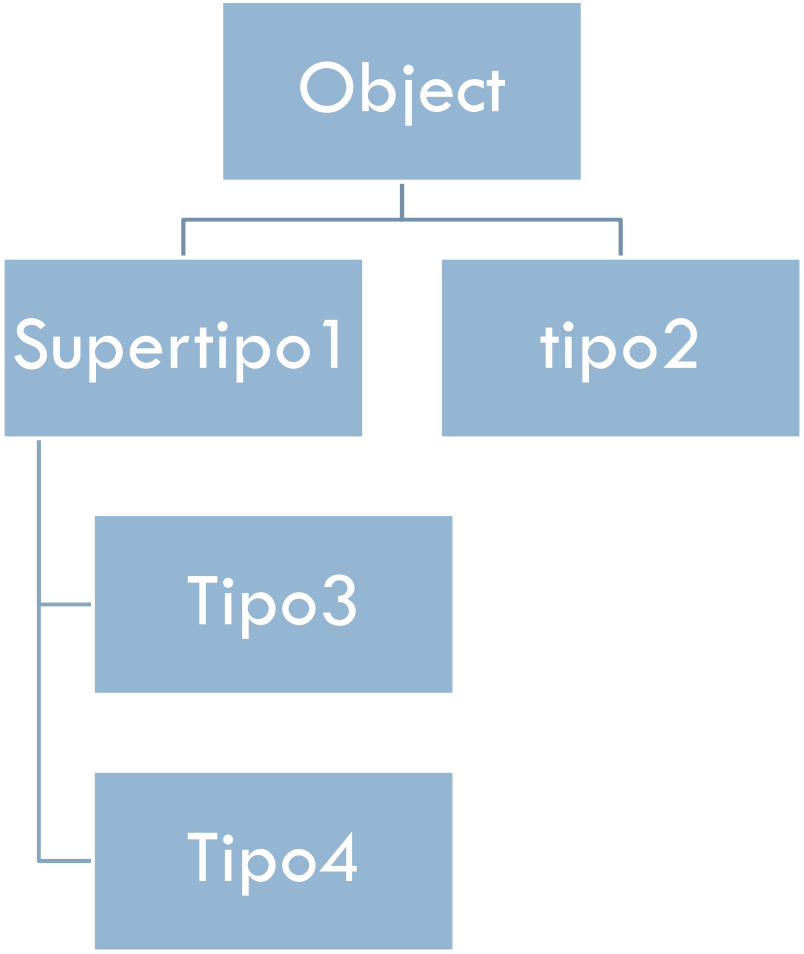
Ficheros de dominio

```
(define (domain <domain name>)
   (:requirements <requirements spec>)
   (:types <types spec>)
   (:predicates <predicates spec>)
   (:action <action spec>)
   (:action <action spec>)
   ….
)
```

- Requirements:
  - :strips
    - El subconjunto más básico de PDDL, solo la representación introducida en STRIPS
  - :adl
    - Disyunciones y cuantificadores en precondiciones y objetivos
    - Efectos condicionales y universalmente cuantificados.
  - :typing
    - El dominio usa tipos.

Ejemplo

```
(:types
   supertipo1 tipo2 - object
   tipo3 tipo4 - supertipo1)
```



Observar:

En cada línea se define un supertipo y sus tipos.

Ejemplo: el mundo de bloques

```
(define (domain BLOCKS)
  (:requirements :strips :typing)
  (:types block)
  (:predicates
        (on ?x - block ?y - block)
        (ontable ?x - block)
        (clear ?x - block)
        (handempty)
        (holding ?x - block)
        )
```

Ejemplo - Acciones

```
(:action pick-up
        :parameters (?x - block)
        :precondition (and (clear ?x) (ontable ?
                           (handempty))
        :effect
        (and (not (ontable ?x))
          (not (clear ?x))
          (not (handempty))
          (holding ?x)))

(:action put-down
        :parameters (?x - block)
        :precondition (holding ?x)
        :effect
        (and (not (holding ?x))
          (clear ?x)
          (handempty)
          (ontable ?x)))
```

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

Fichero de problema

```
(define (problem <problem id>)
  (:domain <domain name>)
  (:objets <object spec>)
  (:init <initial facts spec>)
  (:goal <goal spec>)
)
```

Ejemplo - Problema

```
(define (problem BLOCKS-4-0)

(:domain BLOCKS)

(:objects D B A C - block)

(:INIT  (CLEAR C)
   (CLEAR A)
   (CLEAR B)
   (CLEAR D)
   (ONTABLE C)
   (ONTABLE A)
   (ONTABLE B)
   (ONTABLE D)
   (HANDEMPTY))

(:goal (AND (ON D C) (ON C B) (ON B A)))
)
```

Editores

- Cualquier editor de texto ….
- Plugin para Atom y Sublime https://github.com/Pold87/myPDDL
- PDDL Online Editor (de la web http://planning.domains/)
  - http://editor.planning.domains/
  - Recomendable para la fase de creación de dominios, para problemas ligeros, pero no para problemas más duros (que requieren más de 10 segs. para resolverse.
  - En este caso usar un planificador en local (por ejemplo FF). Siguiente transparencia …
  - Ojo!, editor online no guarda sesión. No abandonar la página antes de guardar todo el trabajo hecho.
- Experimental: Extensión de Visual Studio Code para PDDL
  - https://marketplace.visualstudio.com/items?itemName=jan-dolejsi.pddl

Primera ejecución de un planificador

- Descargar y descomprimir Metric-FF desde PRADO.
  - Por ejemplo, crear /home/<user>/Practica2/<directorio_nuevo> = <directorioFF>
- Compilar Metric-FF
  - Ejecutar make en <directorioFF>
  - Si no compila descargar las librerías para el analizador léxico “lex” y el sintáctico “bison”.
- Hacer la ejecución en local (siguiente slide) o, por ejemplo, a través del editor online de http://planning.domains

Ejecutar Metric-FF

- EN LOCAL

1. Copiar blocks_domain.pddl y blocks_problem.pddl en un directorio, por ejemplo, <dir> =/home/Practica2/DomYProbs
2. Cd <directorioFF>
3. ff –p <dir> -o blocks_domain.pddl –f blocks_problem.pddl
    1. -p indica el directorio donde tenemos almacenados los dominios y problemas

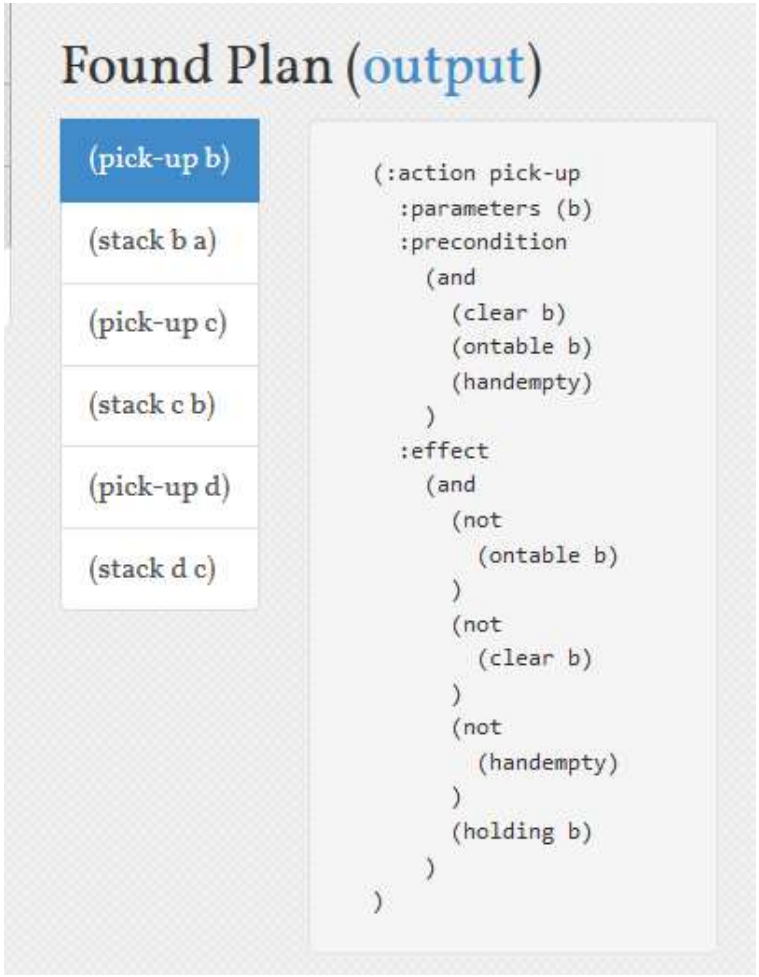
- Observar:
    1. Entrada de un planificador:
        1. Dominio de planificación (declaración de operadores, predicados y tipos)
        2. Problema de planificación (declaración estado inicial y objetivo)
    2. Salida: Un plan (secuencia ordenada de acciones, e.d, de operadores instanciados).

Editor On-Line

- En http://editor.planning.domains
  - Subir ficheros de problema y dominio (opción File/Load)
  - Ejecutar el planificador (opción Solve)
  - Observar la salida (Pestaña "Plan(I)").

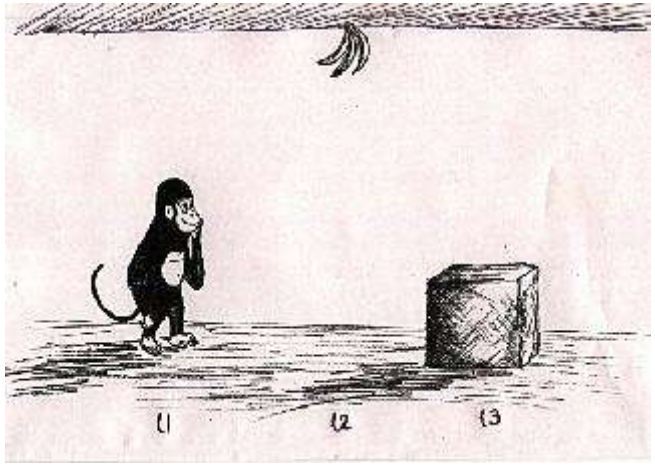
Observar:

1. Entrada de un planificador:
   1. Dominio de planificación (declaración de operadores, predicados y tipos)
   2. Problema de planificación (declaración estado inicial y objetivo)
2. Salida: Un plan (secuencia ordenada de acciones, e.d, de operadores instanciados).



El mono y los plátanos

- Un mono en un laboratorio tiene lejos de su alcance un racimo de plátanos.
- Una caja permite alcanzar los plátanos si éste se sube en ella.
- El mono está en una posición desde la que no alcanza las bananas. Las bananas y la caja están en posiciones distintas también.
- El mono puede **desplazarse** de una ubicación a otra, **empujar** la caja, **subir a la caja** y **coger los plátanos.**
- ¿Qué tiene que hacer el mono para coger los plátanos.



Algunas consideraciones

- Representación
  1. ¿Cómo representamos el estado del mundo?
  2. ¿Cómo representamos acciones?
  3. ¿Hace nuestra representación fácil
     - la comprobación de precondiciones
     - representar el estado del mundo después de ejecutar cada acción
     - reconocer/alcanzar el estado final?

- A continuación conoceremos una "guía de buenas prácticas" para representar dominios y problemas de planificación.

- Ontología (Objetos, propiedades y relaciones) del problema del Mono y los Plátanos
  - Tipos de objetos
  - Relaciones entre objetos:
  - Escoger predicados adecuados y argumentos:

Representando el mundo

- Ontología del problema del Mono y los Plátanos
  - Tipos de Objetos:
    - Tipos: monkey, box, banana, location (**se definen en el dominio)**
      - El tipo "object" es el tipo que representa "cualquier tipo".
    - Objetos: monkey1, box1, banana1, p1, p2, p3(**se definen en el problema)**
  - Relaciones entre objetos:
    - Un mono está, o no, en el suelo
    - Un mono tiene cogidos, o no, los plátanos
    - Un mono está, o no, subido a una caja en una ubicación
    - Cualquier objeto (de tipo monkey, box o banana) está en alguna ubicación
      - Podemos definir un nuevo tipo "locatable" que inicluye a "monkey, box, banana" para distinguirlos del tipo "location"
  - Escoger predicados adecuados y argumentos:
    - `(onfloor ?m – monkey)`
    - `(hasbananas ?m - monkey)`
    - `(onbox ?m – monkey ?b – box ?x – location)`
    - `(at ?m - locatable ?x - location)`

Estado inicial y final

Cuando hayamos definido los predicados, podemos usarlos para definir estados iniciales y finales (el problema).

- Objetos que participan en un problema:
  - p1,p2,p3 son de tipo **location**
  - monkey1 es de tipo **monkey**
  - bananas1 es de tipo **banana**
  - box1 es de tipo **box**
- **Estado inicial:**
  - (at monkey1 p1)
  - (on-floor monkey1)
  - (at box1 p2)
  - (at bananas1 p3)
  - **hipótesis del mundo cerrado: cualquier predicado que no aparezca en el estado inicial es falso.**
- **Estado final:**
  - (hasbananas monkey1)
  - (at monkey1 p2)
  - (onbox monkey1 p2)
  - (at box1 p2)

Objetivo

- El objetivo no tiene que corresponderse con un estado final completo.
- Basta con especificar qué predicados queremos que se hagan verdad en el estado final. Por ejemplo:
  - Goal: hasbananas(monkey1)

Representando operadores

- Nombre y parámetros.
  - Los parámetros de las precondiciones deben aparecer en los parámetros de la acción.
- Precondiciones:
  - qué predicados tienen que ser ciertos para que se aplique el operador (acción)
- Efectos : qué predicados representan el cambio del mundo representado por la acción
  - STRIPS:
    - Adición: qué predicados se añaden al mundo
    - Supresión: qué predicados son eliminados del mundo
  - En pddl una única expresión lógica,
    - Si queremos especificar que se suprime el predicado P entonces escribiremos not(P)

El problema del marco

- ¿Qué aspectos del mundo cambian realmente cuando ejecutamos una acción?
  - Cuando representamos acciones asumimos que los únicos efectos (en la realidad) de nuestro operador son los especificados.
  - En el mundo real es una restricción muy fuerte, puesto que no podemos saber con absoluta seguridad cuales son los efectos de la acción
  - Esto es conocido como el Problema del Marco:
    - ¿qué es lo que se mantiene (marco) y qué es lo que cambia?
- En los sistemas de planificación reales es difícil encontrar una solución de compromiso, y los dominios de planificación deben de adaptarse a medida que se va conociendo la ejecución en el mundo real.
  - La escritura de un dominio de planificación es un proceso
    - Iterativo: se realizan varias veces el proceso escritura-ejecución
    - Incremental: en cada iteración se va añadiendo más conocimiento (precondiciones, efectos, nuevas acciones).

Representación de acciones

- El mono puede **desplazarse** de una ubicación a otra, **empujar** la caja, **subir a la caja** y **coger los plátanos**

- Acción “Ir de una ubicación a otra”
- Acción “Empujar la caja”
- Acción “Subir a la caja”
- Acción “Coger los plátanos”

Representar una acción (I)

- Una acción representa un cambio de estado (de un estado previo a un estado sucesor).
- Los **efectos** de la acción representan el estado sucesor. En general es recomendable
  - plantearse la pregunta "¿qué se pretende conseguir con la ejecución de la acción?"
  - y considerar que una acción va a ser parte de un plan (ordenada antes o después de otras acciones y con las que va a tener relaciones causales)
- Ejemplo:
  - "Coger los plátanos" **NOMBRE : GRAB-BANANAS**
  - Efectos: ¿qué se pretende?. Que el mono tenga los plátanos cogidos.
  - hasbananas(monkey1) -> hasbananas(?m)
  - Efectos: `(and (hasbananas ?m))`

- Representar una acción (II)
  - Las **precondiciones** de la acción representan las condiciones que deben cumplirse para ejecutar correctamente la acción.
  - Ejemplo:
    - Precondición (estado previo): "el mono tiene que estar subido en la caja en alguna posición"
    - ¿Deberíamos representar que el mono y la caja estén en la misma posición?.
      - De nuevo, el problema del marco. Puede no ser necesario representarlo, si puedo escribir otra acción que me garantice que el mono y la caja ya están en la misma posición cuando el mono se suba a la caja. Por lo pronto, asumimos que no es necesario representarlo y vemos más adelante si es necesario.
    - Precondición: (and (onbox ?m ?y))
  - Parámetros: Los argumentos de las precondiciones tienen que ser parámetros de la acción (o constantes si hemos definido constantes en el dominio). Asumimos siempre que el dominio PDDL tiene tipos.
  - GRAB-BANANAS(?m – monkey ?y – location)

- Coger los plátanos

```
(:action GRAB-BANANAS
           :parameters (?m - monkey ?y - location)
           :precondition (and (onbox ?m  ?y))
           :effect (hasbananas ?m))
```

- Verificar **sintaxis** y **semántica** de la acción.
  - **Análisis sintáctico**
    - No hay herramientas para tal fin como en los lenguajes de programación estándar.
    - El proceso de análisis sintáctico se basa en la ejecución de un planificador (que en general tiene implementado un parser de pddl)
  - El **análisis semántico** (comprobar que la acción es correcta) se basa en un proceso manual:
    - Definir un problema y dominio pequeños, para comprobar que la acción se planifica correctamente, es decir, se genera un plan a partir de un problema y un dominio que incorpora esa acción.
    - El problema: se recomienda que contenga como estado inicial solo instancias de las precondiciones de la acción.
    - El dominio: conteniendo solo la acción a verificar.
    - Hacer una ejecución de prueba con el planificador.

Problema

```
(define (problem monkey-test1)
    (:domain monkey-domain)
  (:objects p1 p2 p3 p4 - location
            monkey1 - monkey
            box1 - box
            bananas1 - banana)
  (:init  (onbox monkey1 p2) )
  (:goal (AND (hasbananas monkey1))))
```

Dominio

```
(define (domain monkey-domain)
  (:requirements :strips :equality :typing)
  (:types  monkey banana box - locatable
           location)

  (:predicates
                    (on-floor ?x - monkey )
                    (at ?m - locatable ?x - location)
                    (onbox ?x - monkey ?y - location)
                    (hasbananas ?x - monkey)

                    )
  (:action GRAB-BANANAS
                :parameters (?m - monkey ?y - location)
                :precondition (and (onbox ?m  ?y))
                :effect (hasbananas ?m))
)
```

- Ejemplo con http://editor.planning.domains
- Crear un fichero de dominio y un fichero de problema con el problema y dominio anteriores.
- Comprobar que la acción GRAB-BANANAS está correctamente escrita solucionando el problema.

- Proceso incremental
  - Definir una nueva acción.
  - Verificar la sintaxis y semántica de la acción
  - Para cada una del resto de las acciones
    - Definir acción
    - Verificar sintaxis y semántica
    - Plantear un problema en el que se genere un plan que tenga al menos una instancia de las acciones ya definidas

- Problema propuesto (para hacer en casa o en el descanso)
  - ¿Acción “Subir a la caja”?
  - ¿Acción “Empujar la caja”?
  - ¿Acción “Ir de una ubicación a otra”?

Opciones avanzadas de PDDL

- Precondiciones con disyunciones
  - La expresión lógica de la precondición de una acción no es solo una conjunción (and), además puede incluir el operador de disyunción “or”
- Efectos condicionales
  - Efectos que se hacen ciertos dependiendo de condiciones de contexto.
  - El siguiente ejemplo es de un dominio para resolver problemas simples de logística. Los efectos condicionales se pueden usar para habilitar una solución al “problema de la maleta”: si un objeto contenedor (un camión, p.ej.) se mueve a una posición destino, todos los objetos que contiene también cambian a la posición destino.

Efectos condicionales

```
(define (domain logistics)
        (:requirements :adl :typing )
        (:types physobj location city - object
                obj vehicle - physobj
                truck airplane - vehicle
                airport - location)
        (:predicates          (at ?x - physobj ?l - city)
                              (in ?x - obj ?t - vehicle)
                              (connected ?city1 ?city2 - city))

        (:action drive-truck
        :parameters (?truck - truck ?city1 ?city2 - city)
        :precondition (and  (at ?truck ?city1)
                            (connected ?city1 ?city2))
        :effect (and  (at ?truck ?city2)
                      (not (at ?truck ?city1))
                      (forall (?x - obj)
                              (when (and (in ?x ?truck))
                                    (and (not (at ?x ?city1))
                                         (at ?x ?city2))
                              )
                              )
                       )
    )
```

Relaciones de conectividad en PDDL

```
(define (problem logistics-test1)
        (:domain logistics)
;; make sure these are constants or objects:
;; citya cityb cityc cityd city
(:objects citya cityb cityc cityd - city
          truck1 - truck
          pack1 pack2 pack3 - obj

          )
(:init

(connected citya cityb)
(connected cityb citya)
(connected cityb cityc)
(connected cityc cityb)
(connected citya cityd)
(connected cityd citya)
(connected cityd cityc)
(connected cityc cityd)

(at truck1 citya)
(at pack1 citya)
(at pack2 citya)
(at pack3 citya)
(in  pack1 truck1)
(in  pack2 truck1)
(in  pack3 truck1)
)

(:goal (and (at pack1 cityd)
            (at pack2 cityd)
            (at pack3 cityd) )

)
```

- En problemas que implican transporte hay que representar la relación de adyacencia (conexión) entre los distintos puntos del grafo de ubicaciones.
- El editor online ofrece un plugin que facilita la codificación rápida de relaciones de adyacencia en cuadrículas o grafos.
- ¿Qué representa este problema?
- En http://editor.planning.domains
  - Instalar plugin Misc PDDL Generators
  - Pinchar luego la opción Insert>Network.
  - Escribir las relaciones de conectividad.

Representación de recursos y tiempo

- PDDL ha evolucionado en distintas versiones:
  - «Planning Domain Definition Language», *Wikipedia, the free encyclopedia*. 13-feb-2015. Referencia a todas las versiones y lenguajes de planificación más empleados.
  - PDDL 2.1: uso de recursos y tiempo (y otras extensiones).
  - PDDL 3: uso de preferencias (y otras extensiones).
- PDDL 2.1 permite representar el uso de recursos
  - Ejemplo: el consumo de fuel de un camión: cada vez que se mueve el camión, una cantidad de fuel usado por el camión se actualiza.
  - Ejemplo: representar costes en un grafo (la distancia entre ciudades)

Representación de recursos

```
(define (domain logistics)
 (:requirements :adl :typing :fluents )
 (:types physobj location city - object
         obj vehicle - physobj
         truck airplane - vehicle
          airport - location)
  (:predicates (at ?x - physobj ?l - city)
               (in ?x - obj ?t - vehicle)
               (connected ?city1 ?city2 - city))
   (:functions
      (fuel ?t - truck)
      (consumo-viaje ?t - truck)
      (total-fuel ?t -truck))

    (:action drive-truck
    :parameters (?truck - truck ?city1 ?city2 - city)
    :precondition (and  (at ?truck ?city1)
                        (connected ?city1 ?city2)
                        (>= (- (fuel ?truck) (consumo-viaje ?truck)) 0))
    :effect (and  (at ?truck ?city2)
            (not (at ?truck ?city1))
            (increase (total-fuel ?truck) (consumo-viaje ?truck))
            (decrease (fuel ?truck) (consumo-viaje ?truck))
            (forall (?x - obj)
             (when (and (in ?x ?truck))
                   (and (not (at ?x ?city1))
                 (at ?x ?city2)) ) )))
```

PDDL 2.1 puede representar valores numéricos mediante ***functions o fluents***, pueden interpretarse como predicados que devuelven un valor numérico en lugar de un valor de verdad. Una función o fluent también juega el papel de una variable y por tanto puede asignarse (operador **assign**), incrementarse (operador **increase**) o decrementarse (operador **decrease**).

Usando fluents/funciones pueden expresarse precondiciones con expresiones numéricas.

Representación de recursos

```
(define (problem logistics-test1)
        (:domain logistics)
;; make sure these are constants or objects:
;; citya cityb cityc cityd city
(:objects citya cityb cityc cityd - city
          truck1 - truck
          pack1 pack2 pack3 - obj

          )
(:init

(connected citya cityb)
(connected cityb citya)
(connected cityb cityc)
(connected cityc cityb)
(connected citya cityd)
(connected cityd citya)
(connected cityd cityc)
(connected cityc cityd)

(at truck1 citya)
(at pack1 citya)
(at pack2 citya)
(at pack3 citya)
(in  pack1 truck1)
(in  pack2 truck1)
(in  pack3 truck1)

(= (fuel truck1) 100)
(= (consumo-viaje truck1) 20)
(= (total-fuel truck1) 0)
)

(:goal (and (at pack1 cityd)
            (at pack2 cityd)
            (at pack3 cityd) ))
```

Las funciones se inicializan en el estado inicial con valores numéricos, siguiendo la sintaxis
(= <funcion> valor).

La cantidad de fuel inicial para el camión truck1 es 100.

```
(= (fuel truck1) 100)
```

El consumo que realiza el camión en cada viaje es 20.

```
(= (consumo-viaje truck1) 20)
```

El consumo total de fuel del camión truck1 se almacena en la función (total-fuel truck1). Inicialmente es 0.

```
(define (domain logistics)
    (:requirements :adl :typing :fluents )
    (:types physobj location city - object
        obj vehicle - physobj
        truck airplane - vehicle
        airport - location)
    (:predicates
       (at ?x - physobj ?l - city)
       (in ?x - obj ?t - vehicle)
       (connected ?city1 ?city2 - city))
    (:functions
        (fuel ?t - truck)
        (consumo-viaje ?t - truck)
        (total-fuel ?t -truck)
        (distancia ?x ?y - city)
        (distancia-total))

    (:action drive-truck
    :parameters (?truck - truck ?city1 ?city2 - city)
    :precondition (and  (at ?truck ?city1)
                        (connected ?city1 ?city2)
                        (>= (- (fuel ?truck) (consumo-viaje ?truck)) 0))
    :effect (and  (at ?truck ?city2)
           (not (at ?truck ?city1))
           (increase (total-fuel ?truck) (consumo-viaje ?truck))
           (decrease (fuel ?truck) (consumo-viaje ?truck))
           (increase (distancia-total) (distancia ?city1 ?city2))
           (forall (?x - obj)
            (when (and (in ?x ?truck))
                  (and (not (at ?x ?city1))
                (at ?x ?city2))
                   )
            )
            )
    )
    )
```

Podemos representar costes en los arcos de un grafo y resolver problemas de optimización. Por ejemplo, usando la función ***distancia*** para representar la distancia entre dos ciudades (ver en la siguiente transparencia el problema) y la función ***distancia-total*** para representar el coste del camino recorrido.

```
(define (problem logistics-test1)
        (:domain logistics)
...

          )
(:init

(connected citya cityb)
(connected cityb citya)
(connected cityb cityc)
(connected cityc cityb)
(connected citya cityd)
(connected cityd citya)
(connected cityd cityc)
(connected cityc cityd)

(= (distancia citya cityb) 20)
(= (distancia cityb citya) 20)
(= (distancia cityb cityc) 10)
(= (distancia cityc cityb) 10)
(= (distancia citya cityd) 5)
(= (distancia cityd citya) 5)
(= (distancia cityd cityc) 15)
(= (distancia cityc cityd) 15)

...

(= (fuel truck1) 100)
(= (consumo-viaje truck1) 20)
(= (total-fuel truck1) 0)
(= (distancia-total) 0)
)

(:goal (and (at pack1 cityc)
            (at pack2 cityc)
            (at pack3 cityc) ))
(:metric minimize (distancia-total))

)
```

Podemos representar costes en los arcos de un grafo y resolver problemas de optimización. Por ejemplo, usando la función ***distancia*** para representar la distancia entre dos ciudades (ver en la siguiente transparencia el problema) y la función ***distancia-total*** para representar el coste del camino recorrido.

La sentencia

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

informa al planificador sobre qué función tiene que minimizar

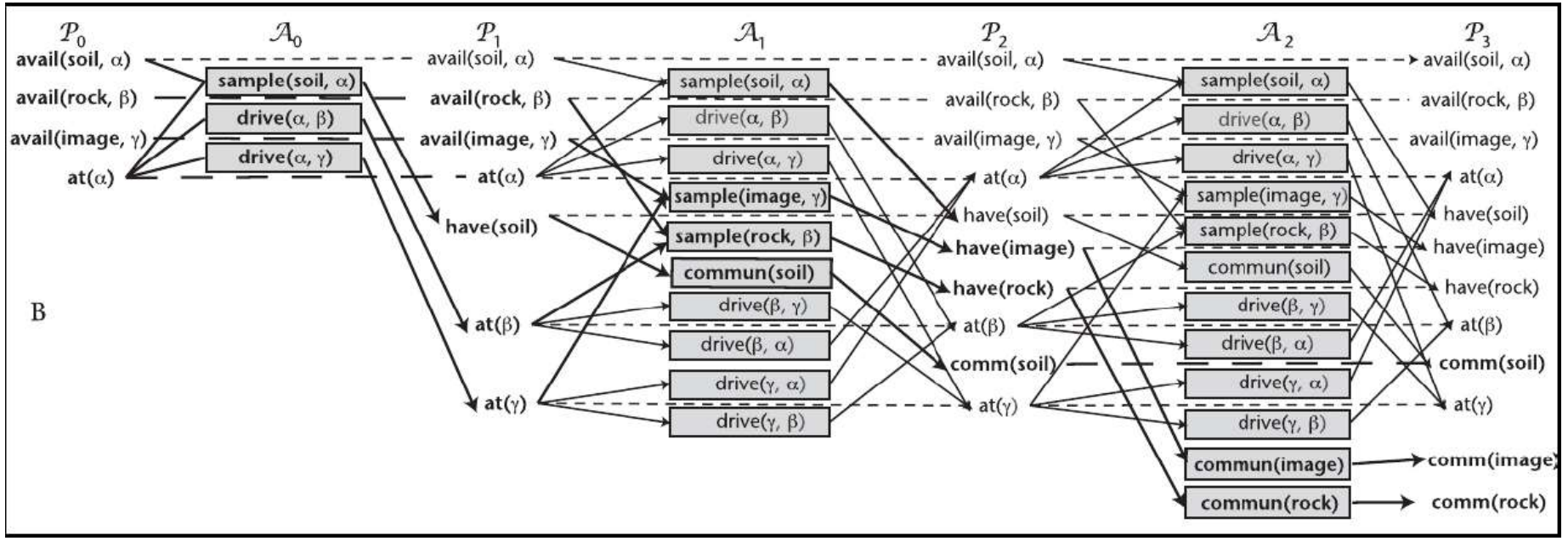
Optimización en FF

- Opciones de línea de comando de FF sobre configuración de la heurística y optimización
  - - O : activa la opción de optimizar. Por defecto FF no optimiza aunque en el fichero del problema esté la sentencia (:metric minimize <funcion>).
    - Si está activada la optimización, por defecto usará como métrica a optimizar la longitud del plan, a no ser que especifiquemos en el problema otra función con **:metric**.
  - Aun así, FF busca soluciones subóptimas si no se configura adecuadamente la heurísitca.
    - Por defecto utiliza una búsqueda primero mejor con la heurística f(n) =1.0 * g(n) + 5.0*h(n)
    - Los pesos de la heurística se pueden configurar con las opciones:
    - - g <peso de g(n)>
    - - h <peso de h(n)>.
- En definitiva, para que FF optimice y busque la solución óptima, el comando tiene que ser:
- `ff – p <directorio> -o <dominio> -f <problema> -O –g 1 –h 1`

Planificador FF (Fast Forward)

- Hoffmann, J., & Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. Journal of Artificial Intelligence Research, 14, 253–302.
- Significó una revolución en las técnicas de planificación y es uno de los más usados y referenciados en la actualidad.
- Utiliza una búsqueda en espacio de estados "estándar" progresiva
  - Genera sucesores a partir del estado inicial, aplicando los operadores (acciones) permitidos en cada estado
  - Operador permitido en un estado:
    - Todas sus precondiciones son satisfechas por el estado.
  - Sucesores:
    - Estados representados como listas de hechos (listas de literales, listas de predicados instanciados).
  - Condición de parada
    - Todos los literales del objetivo están incluidos en el estado seleccionado.

Grafo del plan



- Expande el "**grafo relajado del plan**"
  - Aplicar de forma progresiva (forward) todas las posibles instanciaciones de acciones
    - sin listas de eliminación (e.d. sin efectos negados)
  - desde el estado inicial hasta que todos los objetivos están cubiertos.

Algoritmo general

- Etapa previa: generar el grafo del plan relajado para extraer una heurística admisible:
  - Obtener un plan por regresión (buscando en ese grafo desde los objetivos hacia atrás).
  - El tamaño de ese plan es un valor heurístico que nunca sobreestima la longitud total del plan (siempre desconocida a priori).
- Aplicar Escalada por la máxima pendiente "forzada", desde el estado inicial.
  - "Forzada": Una variante de escalada en la que, si no mejoran los sucesores (meseta, mínimo), se aplica BÚSQUEDA EN ANCHURA hasta que se encuentra un mejor sucesor.
  - El camino hasta ese sucesor es añadido al plan y la búsqueda por escalada continúa.
- Funciona bien porque los problemas de planificación de referencia tienen mesetas y mínimos "pequeños".