Motivación de PDDL

Planteamos el problema general en el que nos encontramos: se define un mundo (domain) donde va a haber una serie de objetos que tienen una clase o tipo (type) y estas entidades van a tener una serie de relaciones entre sí, o posibles estados (predicates) y además el mundo se puede influir mediante acciones (actions), que modificaran los estados de los objetos.

Si os dais cuenta este es un mundo genérico, estamos estableciendo las reglas generales por las que se rige pero no hemos especificado nada; así hemos creado una base para la que se puedan definir diferentes problemas sobre el mismo mundo. Este problema en concreto (problem) usará el mundo genérico que hemos creado (domain) y tendrá los objetos concretos de cada tipo según se lo especifiquemos (objects) así como el estado inicial del mundo, es decir los predicados iniciales (init); y finalmente tendrá un objetivo, lo que queremos que se resuelva (goal).

Al final lo que vamos a tener, es una semejanza a problemas de lógica, tenemos una serie de objetos y predicados iniciales de los cuales usando las acciones que están permitidas en el mundo vamos a ir derivando de esos predicados otros nuevos de manera que consiga llegar al predicado objetivo/goal.

Nota: si un predicado existe se dice que se cumple (que es verdad), si no existe entonces no se cumple (es falso) por tanto tendremos que definir bien las condiciones iniciales.

Domain/Problem

Como ya he comentado en la introducción esta será nuestra estructura para resolver problemas. Primero haremos un archivo que sea del tipo "Ejercicio1.pddl" que contendrá nuestro dominio/mundo genérico donde podremos crear tantos problemas diferentes como queramos sobre este mundo genérico "Problema1-1.pddl" "Problema1-2.pddl"...

Domain

La estructura general es la siguiente:

```
(define (domain Ejercicio1)
    (:requirements :strips :adl :equality :typing)

    (:types)
```

```
    (:constants)

    (:predicates)

    (:functions)

    (:action)

    (:action)
)
```

Definimos un dominio con nombre Ejercicio1 `domain Ejercicio1` y hacemos imports de funcionalidades diversas de PDDL (`:requirements`). A continuación se pondrán los distintos tipos en `:types` , las constantes (objetos constantes) en `constants` , los predicados/estados en `:predicates` y finalmente las acciones que se pueden poner tantas como se quieran `:action` .

Veamos en profundidad cada sección.

Types

Listamos los diferentes tipos (clases para entendernos) de los que se podrán crear objetos. Veamos un ejemplo:

```
(:types
    Zona
    Orientacion
    Oscar Manzana Rosa Algoritmo Oro - Objeto
    Princesa Principe Bruja Profesor Leonardo - Npc
    Npc Jugador - Persona
    Persona Objeto - Posicionable
)
```

Por **cada línea** se crea un tipo, en este caso `Zona` y `Orientacion` son dos tipos diferentes. Además para indicar si una clase es clase de otra se pueden listar todos los tipos que se quieran y al final un guión `-` y la superclase que queremos. Por ejemplo, `Oscar Manzana Rosa Algoritmo Oro` son 5 tipos distintos que además son de tipo `Objeto` ; `Princesa Principe Bruja Profesor Leonardo` son otros 5 tipos distintos que también son tipo `Npc` .

Constants

Este apartado es opcional pero puede ser de utilidad si se tienen objetos constantes en nuestro mundo y no dependen del problema concreto, por ejemplo:

```
(:constants
    N S O E - Orientacion
)
```

En mi caso creo `N S O E` diciendo que son de tipo `Orientacion` y ya puedo usarlas para cosas que haga en el dominio independientemente del problema.

Predicates

Los predicados o estados del mundo, van a representar los distintos estados o relaciones que se dan lugar en nuestro mundo, por ejemplo:

```
(:predicates
    (estanConectadas ?z1 - Zona ?z2 - Zona ?o - Orientacion)
    (orientadoHacia ?j - Jugador ?o - Orientacion)
    (estaEn ?p - Posicionable ?z - Zona)
    (tiene ?p - Persona ?o - Objeto)
)
```

`(estanConectadas ?z1 - Zona ?z2 - Zona ?o - Orientacion)`: si tengo dos salas en un juego y quiero expresar que estan conectadas respecto de una orientación (norte, sur, este, oeste) necesito una relación entre ambas junto a una dirección. Así creo esta relación que necesita de 2 Zonas y orientación, por ello tomamos 3 parámetros de manera que que el nombre de cada parámetro empieza siempre con una `?` seguido del nombre que queramos y después un `-` junto al tipo del parámetro, en este caso ponemos `?z1 - Zona` para decir que hay un parámetro de nombre `?z1` y de tipo `Zona`.

Nota: puede haber predicados sin parámetros.

Functions

Las funciones son iguales que los predicados solo que devuelven números en vez de Falso o Verdadero, por ejemplo:

```
(:functions
  (distanciaZona ?z1 - Zona ?z2 - Zona)
  (distanciaTotal ?j - Player)
)
```

Para cambiar los valores de estos predicados se usan `increase` y `decrease` en las acciones, por ej:

```
  (increase (distanciaTotal ?j) (distanciaZona ?z1 ?z2))
  (decrease (distanciaTotal ?j) (distanciaZona ?z1 ?z2))
```

Nota: hay que incluir en los requirements `:fluents`

Action

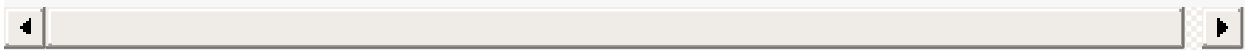
Las acciones que modifican el mundo, y se pueden tener tantas como se quieran, estas constan de la siguiente estructura:

```
(:action nombreAccion
    :parameters ()
    :precondition ()
    :effect ()
)
```

Le ponemos un nombre a la acción el que sea `nombreAccion` y como si fuese una función, necesita una serie de parámetros que se indicarán en `:parameters`, además para que podamos aplicar esta acción tienen que cumplirse antes una serie de condiciones `:precondition`. Finalmente el efecto que produce aplicar esta acción se incluye en `:effect`.

Veamos un ejemplo en concreto:

```
(:action moverseA
    :parameters
      (?j - Jugador ?oJ - Orientacion ?z1 - Zona ?z2 - Zona ?oZ - Orientacion)
    :precondition
      (and (orientadoHacia ?j ?oJ) (estanConectadas ?z1 ?z2 ?oZ) (estaEn ?j ?z1))
    :effect (and (estaEn ?j ?z2) (not (estaEn ?j ?z1)))
)
```



Nota importante: he dejado un salto de linea para que quepa en el pdf, pero originalmente no están esos saltos.

Para los parámetros procedemos igual que cuando poniamos los predicados en `:predicates`, lista de parámetros con `?nombre - tipo`.

H -> z1[] z

En `:precondition` vamos a listar las diferentes condiciones que queremos que se cumplan, como en los lenguajes normales podemos usar `and` o `or` con el mismo efecto lógico conocido; en nuestro caso usamos `and` indicando que se cumplan todos los predicados que siguen a continuación del `and`. Es decir para poder efectuar la acción `moverseA` tiene que cumplirse `orientadoHacia ?j ?oJ`, `estanConectadas ?z1 ?z2 ?oZ` y `estaEn ?j ?z1`; si alguna no existe entonces no se puede ejecutar la acción.

Finalmente en `:effect` incluimos lo que pasa de hacerse la acción, como estamos trabajando con predicados lo que se puede hacer es crear o destruir predicados de manera que con `and` vamos a decir que se van a añadir la siguiente lista de predicados, es decir se añaden `(estaEn ?j ?z2)` y `(not (estaEn ?j z1))` pero lo que pasa ahora es que si precede un `not` antes de un predicado estamos diciendo que se destruya ese predicado (ahora es falso).

Problem

La estructura general de un archivo problema es la siguiente:

```
(define (problem Problema1)
    (:domain Ejercicio1)

    (:objects)

    (:init)

    (:goal)

    (:metric)
)
```

Definimos un problema de nombre `Problema1`, que usa el dominio `:domain Ejercicio1` (tienen que coincidir los nombres), con una serie de objetos/entidades en `:objects`, la lista de predicados iniciales en `:init` y finalmente el predicado objetivo al que se quiere llegar `:goal`.

Objects

Hacemos igual que para los tipos:

```
(:objects
```

```
    jugador1 - Jugador
    princesa1 - Princesa
    principe1 - Principe
    leonardo1 - Leonardo
    bruja1 - Bruja
    profesor1 - Profesor
    oscar1 - Oscar
    manzana1 - Manzana
    rosa1 - Rosa
    algoritmo1 - Algoritmo
    oro1 - Oro
    z1 z2 z3 - Zona
)
```

Lista de objetos, que por ejemplo hay 3 objetos de tipo `Oscar` que son `oscar1 oscar2 oscar3`, y así con todos.

Init

Lista inicial de predicados:

```
(:init
    ;; Estado inicial del jugador
    (orientadoHacia jugador1 N)
    (estaEn jugador1 z13)

    ;; Estado inicial de los npcs
    (estaEn princesa1 z1)
    (estaEn principe1 z5)
    (estaEn leonardo1 z21)
    (estaEn bruja1 z25)
    (estaEn profesor1 z12)
)
```

Usando los predicados que teniamos en el archivo de `Domain` en `:predicates`, usamos los objetos concretos que hemos indicado en `:objects` para crear los predicados iniciales que se indican así, un predicado por línea (además se pueden usar las constantes de `:constants`).

Para los fluents la estructura es:

```
(= (predicado) valorInicial)
```

Por ejemplo:

```
(= (distanciaTotal jugador1) 0)
```

Goal

Finalmente el predicado objetivo:

```
    (:goal (and (tiene princesa1 oro1)
                (tiene principe1 rosa1)
                (tiene principe1 rosa1)
                (tiene bruja1 manzana1)
                (tiene profesor1 algoritmo1)
                (tiene leonardo1 oscar1)))
```

En este caso queremos que se cumplan los 6 predicados concretos (con objetos que hemos definido en el problema concreto).

Metric

Además se le puede indicar que se quiera minimizar una función como:

```
    (:metric minimize (distanciaTotal jugador1))
```

Estructuras adicionales

Comentarios: se hacen por líneas poniendo `;;` al principio.

Estructura de bucle `forall`, esta estructura de control sirve para tomar todos los objetos que existan de un tipo y aplicarles algo.

Estructura condicional `when`: esta estructura condicional sirve para que si se cumple cierta condición entonces se aplique algo.

Veamos estas dos estructuras con un ejemplo:

```
(forall (?p - Perro)
```

```
    (when (and (esGoodBoi ?p))
    ;; Siempre se cumple (claramente)
      (and (soyFelizCon ?p))))
```

Para cada objeto del tipo `Perro` se mira se si cumple la condicion `(esGoodBoi ?p)` y si se cumple (que sí, jeje) entonces se añade `(soyFelizCon ?p)`.

Nota: `when` solo se puede usar en `effect`.

Ejecutando

Se da por hecho que habeis descargado el programa MetricFF de los archivos de la práctica2 que hay en PRADO, pues teniendo los archivos de domain y problem se ejecutan así:

`./ff -p <directorio> -o <nombreArchivoDominio> -f <nombreArchivoProblema> -O -g 1 -h 1`

`-O` es para optimización, y `-g 1 -h 1` indica el parámetro para la métrica que se usa.

Comentarios finales

Esta guía no contiene toda la documentación sobre PDDL, para más información consultad en Internet (aunque no hay mucho); además puede contener errores así que perdonad de antemano si hay algo más. Faltan cosillas que puede que iré ñadiendo según las vaya usando en los ejercicios.

Friendly advise: cuidado con los paréntesis