UNIVERSIDAD DE GRANADA

Técnicas de los Sistemas Inteligentes
grado en ingeniería informática

Práctica 3

Planificación Clásica (PDDL)

Autor
José María Sánchez Guerrero

Rama
Computación y Sistemas Inteligentes || Grupo 2 - Pablo Mesejo

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Curso 2019-2020

Índice

Introducción

En esta práctica tendremos que realizar varias tareas en el mundo de *StarCraft*, confeccionando un dominio de planificación clásico mediante el lenguaje PDDL. Estos ficheros se ejecutarán en el planificador *Metric-FF* que nos sacará un plan válido.

Se dispondrá de un par de ficheros por cada ejercicio (dominio y problema), los cuales ejecutaremos con la siguiente orden:

```
./ff -o <ruta-dominio>-f <ruta-problema>-s 0
```

El código para cada uno de los ejercicios es incremental con respecto al anterior, es decir, voy metiendo pequeñas modificaciones dependiendo de lo que se nos pida. Esto no sucede en el último ejercicio, ya que he tenido que cambiar varias cosas para que se ejecute en un tiempo razonable. Los cambios los explicaré más detalladamente después.

Ejercicio 1

En este ejercicio se nos pide que implementemos todo el dominio de *StarCraft*, es decir, edificios, unidades, recursos y acciones básicas como moverse, asignar o construir edificios.

La declaración de los tipos, las constantes y predicados no tiene mucho misterio, asi que viendo el código puede verse fácilmente cómo y por qué están declaradas. En cuanto a las acciones si las vamos a explicar mejor:

- **Navegar.** Mueve una unidad entre una localización y otra. Para ello, primero comprueba si la localización en la que se encuentra está conectada con la localización a la que va; y despues se asegura que la unidad no estaba extrayendo ningún recurso. Si esto se cumple, añade la unidad a la nueva localización y la elimina de la anterior.
- **Asignar.** Pone a un VCE a extraer recursos de un nodo. Si una unidad del tipo VCE se encuentra en la posición del recurso a extraer y no lo estaba extrayendo previamente, comienza a extraerlo.
- **Construir.** Ordena a un trabajador libre que construya un edificio. Al igual que antes, este trabajador tiene que ser un VCE, estar en la localización donde se va a construir y no puede estar ocupado extrayendo. Además, tenemos

que comprobar que no hay ningún edificio ya construido en esa localización, y para ello hemos utilizado el predicado '`exists`', que busca en el resto de localizaciones ese edificio. Por último, también tiene que existir otra unidad distinta, extrayendo el recurso que necesita el edifico para poder construirse.

Por otro lado, tenemos el fichero problema, en el cual declaramos y colocamos en el mapa tanto las unidades, como los edificios, como los minerales. Como el mundo está representado en forma de cuadrícula, también tendremos que definirla en el problema (en mi caso la hice gracias a la herramienta del editor online).

Ejercicio 2

En este ejercicio tenemos que modificar la acción *Asignar* para que en un recurso de gas haya que construir un **Extractor** antes de asignarlo.

Para ello, tenemos que añadir una precondición más para comprobar el recurso asignado en la localización. Una vez hecho esto, he utilizado un efecto condicional para que se asigne al recurso, bien si en la precondición no era un recurso de gas, o bien, si era un recurso de gas pero tiene un extractor construido.

El ejercicio nos pide construir un barracón, el cual no necesita gas, asi que para probarlo he puesto que en vez de minerales necesite estar extrayendo gas.

Ejercicio 3

Tal y como habíamos definido la acción de **Construir** en los ejercicios anterior, nos damos cuenta de que con un único VCE asignado a un recurso era suficiente para que lo construyese. Eso es lo que vamos a solucionar ahora, es decir, vamos a tener en cuenta que un edificio puede requerir de más de un tipo de recurso.

Hemos añadido el predicado '*generando ?recurso*', que se añade cuando un VCE se asigna en uno. Con esto podemos controlar más fácilmente los recursos que se están generando o no. Después, en la propia acción de construir, en vez de comprobar si hay alguien extrayendo o no, hemos utilizado el predicado '*forall*' para recorrer todos los recursos existentes y comprobar si el edificio a construir no lo necesita, o si lo necesita pero se está generando.

Por ahora, no vamos a necesitar ninguna función que elimine los predicados de generar ya que tampoco hay ninguna que desasigne a las unidades de la extracción

de recursos. Para comprobar que funciona, he puesto que también se construya el centro de mando, ya que es el único que necesita de todos los recursos.

Ejercicio 4

En este ejercicio vamos a añadir un par de unidades nuevas a nuestro dominio. Estas son el **Marine** y el **Segador**, cada uno con sus requerimientos propios y ambos producidos en un *Barracon*. También se cambiará el número de VCEs inicial a uno, y los nuevos se generarán en un *Centro de Mando*.

Las dos unidades las añadiremos como constantes, al igual que los VCE. También añadiremos un predicado para saber que recursos necesita cada unidad para ser creada (similar a la del ejercicio anterior con los edificios), y otro para saber en que edificio es entrenada cada unidad.

Para generar las distintas unidades, hemos creado la acción **Reclutar**. Esta acción primero comprueba el tipo de unidad a reclutar, en qué edificio se entrena y que no haya sido previamente reclutada. Posteriormente, con un predicado *exists* hay que buscar el edificio que la entrena, es decir, si está construido; y si lo ha encontrado, comprobar que se están generando todos los recursos necesarios (al igual que hicimos en la acción de construir).

El objetivo de este ejercicio es poner un marine en una localización y, otro marine y un segador en otra; asi que podremos ver perfectamente el funcionamiento de los nuevos cambios.

Ejercicio 5

En este ejercicio tendremos que crear una nueva acción **Investigar**, la cual nos permitirá desbloquear nuevas unidades que reclutar. Para realizar estas investigaciones necesitaremos un nuevo edificio llamado '*Bahía de Ingeniería*' y los recursos necesarios para llevar la investigación a cabo.

En el dominio hemos añadido el edificio a las constantes, un nuevo predicado para saber que recurso necesita una unidad para ser investigada, y otro predicado que nos indica si ya está investigada o no. En cuanto a las acciones tenemos:

- En la acción de `Reclutar` simplemente hemos puesto la precondición de que la unidad esté investigada.

- Para la nueva acción de `Investigar`, tenemos que comprobar que la unidad no ha sido investigada previamente, que *exists* un edificio en alguna localización del tipo BahíaDeIngeniería, y al igual antes, o que no necesite un recurso, o que lo necesite pero lo esté generando.

En cuanto al fichero problema, declaramos una nueva variable para la bahía, los recursos que necesita cada unidad, y, en mi caso, he puesto las unidades VCE y marine como ya investigadas (ya que según el enunciado disponemos de ellas desde un principio). El objetivo será el mismo que en el ejercicio anterior.

Ejercicio 6

Hasta ahora, con tener asignada una unidad a un recurso nos bastaba para tener una cantidad ilimitada del mismo, pero en este ejercicio se nos pedirá que cambiemos esto. Cada edificio, unidad o investigación costará una cantidad determinada de recursos, y además, tendremos que añadir nuevas acciones para poder llevarlo a cabo: recolectar y desasignar.

Por otro lado, también hemos "reducido" la cantidad de predicados (realmente hay más, pero por temas de optimización que después explicaré), ya que nos deshacemos de los '*generando ?recurso*' y de los que indican qué recurso se necesita para construir, reclutar o investigar. Esto se debe a que ahora sólo comprobamos los recursos que tenemos almacenados.

Para **almacenar recursos** hemos creado las siguientes funciones:

```
1  (:functions
2      (recursoAlmacenado ?rec - tipoLocalizaciones)
3      (capacidadMaxima)
4  )
```

Además de las dos nuevas acciones, también modificamos las anteriores para consumir un cierto número de recursos. Vamos a explicarlas todas a continuación:

- La nueva acción **Recolectar**, por cada trabajador asignado a un nodo, incrementará en 20 el recurso (para hacerlo en un tiempo razonable). Esta acción se puede ejecutar en cualquier momento, ya que sólo va a tener la condición de no sobrepasar la capacidad máxima del almacenamiento. En el *effect* tenemos que poner un `forall` para que incremente por cada trabajador asignado.
- La otra acción **Desasignar**, como su propio nombre indica, hacemos que un VCE deje de extraer un recurso en un nodo. Para que se puede ejecutar, sim-

plemente tiene que haber un VCE extrayendo un recurso en una localización determinada.

- Hay que modificar la acción **Construir**. Las precondiciones serán las mismas que teníamos antes, sin embargo, los efectos cambian totalmente. Primero, con un `when`, vemos que edificio se está construyendo y si tenemos materiales suficientes para hacerlo, y dependiendo de cuál sea, decrementamos la cantidad de recursos correspondiente. Esto lo haremos con la función `(decrease (recursoAlmacenado ?recurso) ?valor)`. También añadiremos el predicado que indica la unidad que se puede reclutar en este edificio.

  En caso de construir el **Depósito**, tendremos que incrementar la capacidad máxima de almacenamiento en 100, y en caso de construir la `Bahía de Ingeniería`, habilitamos las investigaciones.

- Con la acción **Reclutar** tenemos una situación similar a la anterior, ya que dependiendo de la unidad que reclutemos tenemos que decrementar una cantidad de recursos u otra, comprobando previamente si disponemos de los recursos.

- En la acción **Investigar** también nos sucede igual que las dos anteriores, decrementando los recursos que se requieren para investigar la unidad.

Tras implementar todo esto, incrementando el código del ejercicio anterior, me he dado cuenta que la ejecución tardaba demasiado. La solución que obtenía era correcta, por lo que decidí que había que optimizar el código. Esto lo he hecho eliminando el máximo de bucle posibles, es decir, todos los *exists* y todos los *forall* posibles. Pese a esto, el código se sigue manteniendo igual que antes, pero en vez de tener una precondición o efectos más complejos, tiene más predicados.

Por ejemplo: previamente en la acción `Construir` teníamos los siguiente:

```
; el edificio no puede estar ya construido
(not (exists (?otraLoc - Localizaciones) (edificioEn ?edi ?otraLoc)) )

; y no puede existir un edificio en esa localizacion previamente
(not (exists (?otroEd - Edificios)(edificioEn ?otroEd ?loc)) )
```

Si añadimos los siguientes predicados nos evitamos esas dos búsquedas:

```
(not (hayEdificio ?loc))          ; no puede haber un edificio en esa
    localizacion
(not (construido ?edi))           ; el edificio a construir no puede
    estar ya construido
```

Estos predicados también tendrán que ser añadidos en la propia acción de construir, en este caso. Al igual que estos, he creado cuatro predicados más:

```
; Unidades que han sido reclutadas
(reclutada ?uni - Unidades)

; Unidades que puede reclutar un tipo de edificio
(puedeReclutarEn ?tipoU - tipoUnidades ?loc - Localizaciones)

; Recurso que esta extrayendo un VCE determinado
(extrayendoRecurso ?vce - Unidades ?rec - tipoLocalizaciones)

; Indica si hay una bahia de ingenieria disponible para investigar
(investigaciondisponible)
```

Los cambios que hacen estos predicados son como los explicados en el ejemplo, por eso no los voy a mostrar uno a uno. La esencia del código sigue siendo la misma y las precondiciones también, solo que expresadas de otra forma. Pese a esto, el código tardaba bastante tiempo, por lo que decidí cambiar la cantidad de recursos recolectados al doble (de 10 a 20 por unidad asignada), y con esto el plan lo encuentra en menos de 2 segundos.

Por último, en el fichero problema hemos tenido que añadir los variables correspondientes al ejercicio, como el depósito; añadir los predicados necesarios, como el almacenamiento de los recursos y su capacidad máxima (inicializados en 0 y 100 respectivamente); y eliminar los que sobraban, como los que indican que recurso necesita cada unidad para crearse.

También han sido añadidos los correspondientes a los predicados para optimizar. En cuanto al objetivo del problema, será el mismo que en el ejercicio anterior.