Memoria Práctica 2

Técnicas de los Sistemas Inteligentes. Grupo Lunes.

Antonio Coín Castro

Curso 2018-2019

Consideraciones generales

Se proporciona en la carpeta `Generador` un *parser*, como se pide en el enunciado de los ejercicios. El nombre de los dominios será siempre `ejX-domain`, donde `X` $\in [1, 7]$ es el número del ejercicio en cuestión. Se considera que todas las líneas necesarias están presentes en los archivos de mapas, y además en el orden en el que se especifican en los ejemplos.

Para cada ejercicio se proporcionan dos problemas, uno con 11 zonas y otro con 25 zonas. El segundo de ellos ha sido generado a partir del *parser*, y los ficheros de mapas utilizados para ello se encuentran en la carpeta `Generador/Mapas`.

Un detalle a tener en cuenta es que se asume que en todos los ejercicios **uno de los objetivos es siempre que cada personaje reciba al menos un objeto**. Este comportamiento puede modificarse para los ejercicios del 4 en adelante en el *parser*, modificando la variable booleana `goal_objects`.

Todos los problemas proprcionados tienen solución, y se encuentra en un tiempo más que razonable (unos segundos). La mayoría de ejemplos contienen zonas conectadas aleatoriamente y personajes y objetos colocados de forma también arbitraria. Sin embargo, tienen la suficiente variedad como para servir de comprobación de la corrección de los dominios.

Por último, se incluye la salida de la ejecución con `ff` de cada problema en las carpetas `EjXsoluciones`. Tan solo el primer problema del ejercico 2 se ha ejecutado con optimización, especificando `-O -h 1` como parámetros de `ff`. Se omite en el fichero correspondiente la salida relativa a la optimización, pues ocupa un número excesivo de líneas.

A continuación se explican las decisiones de diseño de cada ejercicio. Al tratarse de un trabajo incremental, se asume que las cosas que no se comenten en un ejercicio son iguales que en el ejercicio anterior.

Ejercicio 1

Tipos

Para el dominio consideramos los tipos `character` y `obj` como subtipos de un tipo `locatable`, lo que nos permite localizar objetos y personajes en una zona concreta. Definimos también los tipos `npc` y `player` como subtipos de `character`, que representan a los personajes pasivos y al robot jugador, respectivamente. Por último, definimos los tipos `zone` y `orient` para representar las zonas del mapa y la orientación del jugador en cada momento.

Predicados

Establecemos los predicados básicos para el funcionamiento del juego:

- `(player_looking ?o - orient)` nos dice si el (único) jugador está orientado según la orientación `?o`. La orientación inicial del jugador es **siempre** `n` en los problemas desarrollados.
- `(connected ?z1 ?z2 - zone ?o - orient)` nos dice si la zona `?z1` está conectada con la zona `?z2` según la orientación `?o`. Por ejemplo, si `(connected z10 z4 n)` es verdadero, quiere decir que la zona `z10` tiene al norte una conexión con la zona `z4`. Notemos que entonces la zona `z4` debe estar necesariamente conectada al sur con la zona `z10`.
- `(at ?l - locatable ?z - zone)` nos permite saber si una entidad `?l` se encuentra en la zona `?z`.
- `(has-obj ?c - character)` controla si el personaje (jugador o *npc*) `?c` tiene un objeto. Estamos teniendo en cuenta que el jugador solo tiene una mano, y por tanto solo puede tener un objeto cada vez. No necesitamos contar cuántos objetos tienen el resto de personajes, pues nos basta con saber que al menos han recibido uno.
- `(hand ?p - player ?o - obj)` es verdadero cuando el jugador `?p` tiene en la mano el objeto `?o`. Este predicado tiene como parámetro un jugador porque se añadió después de haber realizado otros ejercicios, en los que podía haber más de un jugador.

Acciones

- `turn-left` y `turn-right` permiten al jugador girarse a izquierda o a derecha, resultando únicamente en un cambio de orientación. Necesitamos para ello conocer la orientación actual del jugador.

- `move` es la única acción de movimiento real, que permite al jugador avanzar de una zona a otra conectada con ella, siempre que la orientación del mismo sea la correcta. El jugador cambia así de localización.
- `give` permite entregar un objeto que el jugador tenga en la mano a un personaje, siempre y cuando ambos se encuentren en la misma zona del mapa.
- `drop` representa la acción de soltar un objeto en un zona concreta del mapa. Para ello, el jugador debe tener un objeto en la mano, y pasará a dejar de tenerlo (el objeto estará ahora en la zona en la que se ha soltado).
- `pick-up` es la acción inversa a `drop`: permite recoger (en la mano) un objeto de una zona concreta.

Cabe destacar que en la implementación de las acciones de girar se asume que los objetos que representan las cuatro posibles orientaciones se llaman **exactamente** `n`, `s`, `e` y `w`, por lo que tendrán que estar declarados así en cualquier fichero de problema. De esta forma podemos comprobar qué orientación tenía el jugador, y ajustar la orientación final en consecuencia. Por ejemplo, si estamos mirando al norte y giramos a la derecha acabaremos mirando al este, lo que codificamos como

```
(when (player-looking n) (player-looking e)).
```

Problemas

Necesitamos declarar tantos objetos de tipo `zone` como zonas tenga nuestro mapa, y establecer mediante predicados `connected` las conexiones entre ellas (teniendo en cuenta que para conectar dos zonas necesitamos hacer verdaderos dos predicados). Además, definimos los objetos (de tipo `obj`) y personajes (de tipo `npc`) que habrá en nuestro mapa, empleando predicados `at` para especificar la localización de cada uno de ellos.

El robot jugador será un objeto de tipo `player`, del que también tenemos que conocer su posición (mediante `at`) y su orientación (mediante `player-looking`).

Como ya dijimos, es necesario declarar también objetos `n`, `s`, `e` y `w` de tipo `orient`, que representan las orientaciones (*north*, *south*, *east*, *west*).

Por último, establecemos el *objetivo* como ya dijimos: el predicado `has-obj` debe ser verdadero sobre cada personaje del mapa.

Ejercicio 2

Dominio

En este ejercicio únicamente añadimos dos **funciones** (*fluents*) al dominio: `(distance ?z1 ?z2 - zone)` para medir la distancia entre dos zonas, y `(total-cost)` para acumular

el coste total de recorrer el mapa. Esto último se hace añadiendo en el efecto de la acción `move` la línea `(increase (total-cost) (distance ?z1 ?z2))`.

Problema

Como es natural, debemos inicializar las distancias entre todas las ciudades (de nuevo atendiendo a la simetría entre ellas), que las ponemos a `1` salvo una de ellas que ponemos a `2` para que no se consuma demasiado tiempo optimizando el plan. También debemos inicializar `total-cost` a `0`, y especificar la métrica que queremos optimizar (minimizar) con `(:metric minimize (total-cost))`.

Por lo demás, los problemas usados en este ejercicio son idénticos a los del ejercicio 1.

Para comprobar el efecto de la optmización, hemos ejecutado el mapa más pequeño (`Ej2problema1`) con y sin optimización del coste. De la primera forma conseguimos un plan con coste `17`, mientras que de la segunda el coste del plan aumenta hasta `22`.

Ejercicio 3

Tipos

Consideramos que los nuevos objetos que se añaden **no son entregables** a los personajes, por lo que cambiamos los tipos de los objetos. Definimos un tipo de objeto normal `normal-obj` (con supertipo `obj`) que representa a los objetos entregables, y dejamos el tipo `obj` para los nuevos objetos.

También añadimos el tipo `type` para representar los distintos tipos de zonas. Como con las orientaciones, se asume que los nombres de las zonas (objetos del problema) son exactamente `arena`, `agua`, `bosque`, `piedra` y `precipicio`.

Predicados

Añadimos nuevos predicados para reflejar la mecánica de los tipos de zonas, su interacción con los nuevos objetos y la mochila.

- `(zone-type ?z - zone ?t - type)` establece que la zona `?z` es de tipo `?t`.
- `(backpack-empty ?p - player)` nos permite saber si la mochila del jugador `?p` está vacía.
- `(backpack ?p - player ?o - obj)` es verdadero cuando el jugador `?p` tiene en la mochila el objeto `?o`.
- `(is-bikini ?o - obj)` y `(is-sneakers ?o - obj)` sirven para saber si un objeto `?o` es un bikini o unas zapatillas.

- `(has-bikini ?p - player)` y `(has-sneakers ?p - player)` permiten conocer si el jugador `?p` tiene (en la mano o en la mochila) un bikini o unas zapatillas.

Acciones

Modificamos las precondiciones de la acción `move` para tener en cuenta las nuevas restricciones impuestas por los tipos de las zonas. Así, no podremos avanzar si la zona de destino es un `precipicio`, si es un `bosque` y no tenemos las zapatillas, o si es `agua` y no tenemos el bikini.

En las zonas que necesitan objetos para llegar a ellas suponemos que el único impedimento es **avanzar** hacia ellas, es decir, podemos estar sobre ellas sin el objeto necesario (por ejemplo, si lo soltamos en esa casilla).

Por otro lado, en las acciones `pick-up` y `drop` debemos controlar ahora si cogemos o dejamos un bikini o unas zapatillas.

Por último, añadimos las acciones `put-in-backpack` y `remove-from-backpack` para meter y sacar objetos de la mochila, respectivamente. Para meter un objeto, necesitamos que la mochila esté vacía y que tengamos ese objeto en la mano, y para sacarlo necesitamos que esté en la mochila y tener la mano vacía. En ningún caso se producen transiciones directas mochila-suelo.

Problemas

Debemos declarar objetos de tipo `type` que representen las zonas que hemos definido, y establecer con predicados `zone-type` de qué tipo es cada zona de nuestro mapa. También debemos cambiar la definición de los objetos para adaptarlos a los nuevos tipos, acordándonos de hacer verdaderos los predicados `is-bikini` y `is-sneakers` para los bikinis y las zapatillas. Por último, debemos establecer que la mochila está inicialmente vacía con `backpack-empty`.

Ejercicio 4

Dominio

Como en el ejercicio 2, en este caso solo necesitamos añadir al dominio dos *fluents*: `(points ?n - npc ?o - normal-obj)` para decir cuántos puntos se obtienen al entregar el objeto `?o` al personaje `?n`, y `(total-points)` para acumular los puntos totales del jugador. Esto último se consigue añadiendo al efecto de la acción `give` la línea `(increase (total-points) (points ?n ?o))`.

Problema

Debemos inicializar `(total-points)` a `0` y reflejar la tabla de puntuaciones de cada personaje y cada objeto mediante predicados `points`. Como esta tabla es estática, el *parser* se encarga de escribirla de la misma forma en todos los problemas.

Por último, modificamos el objetivo (*goal*) para conseguir al menos un número de puntos prefijado de antemano. Por ejemplo, `(>= (total-points) 34)`. Modificando esta cantidad y añadiendo o eliminando objetos y/o personajes del mapa podemos conseguir situaciones en las que objetos concretos tengan que ser entregados a personajes concretos para alcanzar la puntuación pedida.

Ejercicio 5

Dominio

Añadimos la función `(pocket ?n - npc)` para controlar cuántos objetos ha recibido el personaje `?n`, y la función `(pocket-size ?n - npc)` para saber el tamaño del bolsillo del personaje `?n`. Así, en la acción `give` comprobamos como precondición que `(pocket ?n)` sea menor que `(pocket-size ?n)`, y añadimos como efecto incrementar `(pocket ?n)` en `1`.

Problemas

Debemos inicializar a `0` la función `pocket` para todos los personajes que intervengan en el problema, y establecer también para cada uno su `pocket-size`.

En este ejercicio se plantea un problema (`Ej5problema1`) en el que solo hay una bruja y una princesa, ambas con un bolsillo de tamaño `2`, y se pide alcanzar `34` puntos. Para ello es necesario repartir los objetos entre los dos personajes, y además de forma eficiente.

Ejercicio 6

Dominio

En primer lugar, se modifica el predicado `player-looking` para que acepte también como parámetro un jugador (ya que ahora hay dos jugadores). Con el mismo espíritu, se modifica la función `total-points` para que se refleje el jugador cuyos puntos se están acumulando.

Problema

Simplemente debemos definir dos objetos de tipo jugador, teniendo cuidado de especificar para cada uno su orientación, su posición, sus puntos iniciales (`0`) y que tienen inicialmente la mochila vacía, de la misma forma en que lo hacíamos para un solo jugador.

En el objetivo requerimos que cada jugador tenga un número mínimo de puntos, y también que entre los dos sumen unos puntos prefijados de antemano. Por ejemplo, un objetivo posible sería el siguiente:

```
(>= (+ (total-points jugador1) (total-points jugador2)) 45)
(>= (total-points jugador1) 20)
(>= (total-points jugador2) 20)
```

Ejercicio 7

Dominio

Se añaden dos subtipos de `player`: `dealer` y `picker`. Se modifica la función `total-points` para recibir como parámetro un `dealer`, pues estos son los únicos que pueden entregar objetos y conseguir puntos.

La acción `give` se modifica para que solo pueda realizarla un `dealer`. Del mismo modo, la acción `pick-up` se modifica para que solo pueda realizarla un `picker`. Se asume que el `dealer` sigue pudiendo soltar objetos en el suelo (por ejemplo, para posibilitar que el `picker` recoja un bikini o unas zapatillas que haya dejado el `dealer` en el suelo).

Finalmente, es necesario añadir una acción `give-to-player` mediante la cual un `picker` puede proporcionarle a un `dealer` un objeto genérico de tipo `obj`, siempre que ambos se encuentren en la misma zona. Debe cumplirse que el que `picker` tenga el objeto en la mano, y que el `dealer` tenga la mano vacía.

Problemas

Podemos declarar ahora tantos jugadores como queramos, teniendo en cuenta que deben ser de tipo `picker` o `dealer`, y realizar para cada uno la inicialización pertinente comentada en el ejercicio anterior. Debemos tener en cuenta que en este caso la inicialización de `total-points` solo se hacer para los jugadores de tipo `dealer`.

Se asume que si solo hay un `dealer`, sus puntos mínimos coinciden con los puntos totales exigidos (pues si hay algún plan, alcanzará ese número de puntos).