Práctica 3 - Planificación HTN

Técnicas de los Sistemas Inteligentes

David Gil Bautista

45925324M

Análisis del problema

Al igual que en la práctica anterior nos encontramos con un planificador, pero en este caso le añadimos unas mejores que nos permiten exprimir su potencial.

En esta práctica tratamos con un problema con un dominio basado en los viajes aéreos, dicho dominio se denomina ZenoTravel. Para resolver el problema se nos dan una serie de primitivas que nos permiten operar con las aeronaves y hacer que vuelen hacia una ciudad, que embarquen a pasajeros... etc.

Con el uso de HTN crearemos tareas que se ejecuten cuando cierto caso ocurra, y que cada caso tengo un efecto distinto.

Ejercicio 1

Para este ejercicio podemos ver que no es posible obtener un plan capaz de resolver el problema. Esto es así ya que contamos con un avión y un pasajero que se encuentran en ciudades distintas y que no son el destino. Para resolver esto debemos hacer que nuestro avión viaje a la ciudad en la que se encuentra y que después vaya a la ciudad destino.

```
    (:method Case3;
      :precondition (and (at ?p - person ?c1 - city)(at ?a - aircraft ?c2 - city))
      :tasks(
             (mover-avion ?a ?c2 ?c1) ; Ir a donde está el pasajero
             (board ?p ?a ?c1) ;        Embarcarlo
             (mover-avion ?a ?c1 ?c);   Ir a la ciudad destino
             (debark ?p ?a ?c);         Desembarcar
            )
    )
```

Añadiendo este método conseguimos resolver el problema, obteniendo la siguiente salida.

```
PS C:\Users\David\Desktop\p3> .\htnp\htnp.exe -d .\zenotravel-V00.pddl -p .\problema-zeno-v01.pddl

.\zenotravel-V00.pddl:41: warning: Duplicated variable ¿? `?x'.
################################################################################
0 errors 1 warnings parsing domain file: .\zenotravel-V00.pddl
--------------------------------------------------------------------------------
:action (fly a1 c4 c1) start: 05/06/2007 08:00:00 end: 05/06/2007 23:00:00
:action (board p1 a1 c1) start: 05/06/2007 23:00:00 end: 06/06/2007 00:00:00
:action (fly a1 c1 c5) start: 06/06/2007 00:00:00 end: 06/06/2007 10:00:00
:action (debark p1 a1 c5) start: 06/06/2007 10:00:00 end: 06/06/2007 11:00:00
:action (fly a1 c5 c2) start: 06/06/2007 11:00:00 end: 07/06/2007 02:00:00
:action (board p2 a1 c2) start: 07/06/2007 02:00:00 end: 07/06/2007 03:00:00
:action (fly a1 c2 c5) start: 07/06/2007 03:00:00 end: 07/06/2007 18:00:00
:action (debark p2 a1 c5) start: 07/06/2007 18:00:00 end: 07/06/2007 19:00:00
:action (fly a1 c5 c3) start: 07/06/2007 19:00:00 end: 08/06/2007 10:00:00
:action (board p3 a1 c3) start: 08/06/2007 10:00:00 end: 08/06/2007 11:00:00
:action (fly a1 c3 c5) start: 08/06/2007 11:00:00 end: 09/06/2007 02:00:00
:action (debark p3 a1 c5) start: 09/06/2007 02:00:00 end: 09/06/2007 03:00:00
Number of actions: 12 (12)
Expansions: 9
Generated nodes: 21
Inferences: 0
Time in seconds: 0
Real Time: 0.0325
Used Time: -8.9407e-11
System Time: 3.35276e-10
PS C:\Users\David\Desktop\p3>
```

Ejercicio 2

En este ejercicio debemos tratar con el combustible del avión para contemplar si es posible o no realizar un viaje. Para ello, a la hora de que un avión vaya a realizar un viaje con *mover-avión* debemos comprobar el combustible disponible y si es factible o no viajar.

Para resolver esto hemos añadido los siguientes métodos.

```
(:method fuel-suficiente
  :precondition ((hay-fuel ?a ?c1 ?c2))
  :tasks (
      (fly ?a ?c1 ?c2)
  )
 )
 (:method no-fuel-suficiente
   :precondition ((not(hay-fuel ?a ?c1 ?c2)))
   :tasks (
     (refuel ?a ?c1)
     (fly ?a ?c1 ?c2)
   )
 )
```

Si hay combustible se viaja, en caso contrario, se reposta y se viaja. Para no saturar la memoria con las salidas adjuntaré en un directorio las capturas con los resultados obtenidos.

Ejercicio 3

Para este ejercicio debemos añadir dos nuevas acciones a la hora de que el avión realice un viaje. Dependiendo del combustible del avión podrá realizar un viaje rápido (que consumirá más recursos pero llegará antes) o un viaje lento. Sabiendo esto tenemos que modificar lo añadido previamente para incorporar estos nuevos tipos de viaje.

```
; En caso de que tengamos el máximo combustible viajamos rápido
 (:method fuel-maximo
   :precondition (and (fuel-maximo ?a ?c1 ?c2) (hay-fuel ?a ?c1 ?c2) (fast-fly ?a ?c1 ?c2))
   :tasks(
           (zoom ?a ?c1 ?c2)
          )
  )

  ; No tenemos combustible máximo
  (:method no-fuel-maximo
    :precondition (and (not(fuel-maximo ?a ?c1 ?c2)) (not(hay-fuel ?a ?c1 ?c2)) (fast-fly ?a ?c1
?c2))
    :tasks(
      (refuel ?a ?c1)
      (zoom ?a ?c1 ?c2)
    )
  )
```

Ahora, comprobaremos si tenemos el máximo de combustible y no sobrepasamos el límite de fuel, lo cual nos permite realizar un viaje rápido, para el segundo método, debemos comprobar que no haya fuel máximo ni suficiente, lo que implica que tengamos que repostar, y una vez hecho ya podremos realizar un viaje rápido.

Esto entra en conflicto con el método *no-fuel-suficiente* implementado anteriormente, porque ahora, **en caso de que no haya fuel suficiente**, obviamente no habrá fuel máximo, y **deberemos repostar**, una vez repostado contamos con el máximo combustible, **por lo que podríamos hacer un viaje rápido.**

Para solucionar esto, se usa el tipo de vuelo, que nos permite saber si con un viaje rápido o lento sobrepasamos el límite de combustible. **En caso de que no lo sobrepasemos, los vuelos rápidos siempre tendrán preferencia**, pero es posible que con el combustible al máximo se realice un viaje lento para no sobrepasar el límite.

```
  ; Para los métodos anteriores añadimos que el tipo de viaje sea lento
  (:method fuel-suficiente
   :precondition (and(hay-fuel ?a ?c1 ?c2) (not(fuel-maximo ?a ?c1 ?c2)) (slow-fly ?a ?c1 ?c2))
   :tasks (
           (fly ?a ?c1 ?c2)
          )
  )

  (:method no-fuel-suficiente
    :precondition (and(not(hay-fuel ?a ?c1 ?c2)) (not(fuel-maximo ?a ?c1 ?c2)) (slow-fly ?a ?c1
?c2) )
    :tasks (
      (refuel ?a ?c1)
      (fly ?a ?c1 ?c2)
    )
  )
```

Al igual que en los ejercicio anteriores, adjunto la salida por pantalla. Con una salida he utilizado lo anterior, sin embargo, con la otra, he supuesto que siempre se realizan viajes rápidos cuando tenemos el combustible máximo, lo cual ha tardado un poco más en planificar.

Ejercicio 4

a) Avión con capacidad máxima

Para resolver este problema hemos modificado las primitivas de ZenoTravel, añadiendo un *increase* y un *decrease* del número de pasajeros actuales en el avión. También creamos una función con la capacidad máxima del avión, y para embarcar a una persona comprobamos que el número de pasajeros sea menor que la capacidad máxima.

```
(:derived
  (plaza-libre ?a - aircraft)
  (> (capacidad-maxima ?a) (pasajeros ?a))
)
```

b) Tareas compuestas

En este problema debemos conseguir que varios pasajeros sean capaces de embarcar y desembarcar al mismo tiempo. Para ello hemos hecho que la función de transportar personas sea recursiva, de esta forma podemos conseguir que varias personas ejecuten una misma acción de forma continua.

Hemos contemplado los siguientes casos:

1. **Pasajero en avión - Avión en destino:** Se desembarca al pasajero
2. **Pasajero en avión - Avión no en destino:** Viajar al destino
3. **Pasajero no en avión - Avión no en destino:** Avión viaja hasta ciudad pasajero
4. **Pasajero no en avión - Avión en destino:** Avión viaja hasta ciudad pasajero
5. **Pasajero no en avión - Avión en ciudad pasajero:** Se embarca al pasajero
6. **Pasajero no en avión, en ciudad destino:** No se hace nada

Para esto hemos creado los 6 casos en la tarea *transport-person*.

c) Duración limitada del viaje

Ahora debemos añadir que los vuelos de los aviones tengan un límite de tiempo. En este caso añadimos dos nuevos predicados que nos indicarán que tipo de vuelo se podrá realizar según el tiempo. Dichos predicados se activarán cuando el tiempo de un vuelo rápido o lento sea menor que el total de tiempo volado por un avión. Añadimos 3 funciones en las que guardamos el tiempo total, el tiempo que consume un vuelo rápido y el de un vuelo lento.

```
(:derived
  (slow-time ?a - aircraft ?c1 - city ?c2 - city)
  (< (+ (total-time-used) (* (distance ?c1 ?c2) (duration-slow))) 10000)
)

(:derived
  (fast-time ?a - aircraft ?c1 - city ?c2 - city)
  (< (+ (total-time-used) (* (distance ?c1 ?c2) (duration-fast))) 10000)
)
```

También modificamos las primitivas, añadiendo en *fly* y *zoom* la actualización del tiempo volado por un avión.

d) Añadido problema con 5 personas y 2 aviones

Adjunto la salida en el directorio del ejercicio 4

e) Añadido dominio con distancias reales

He creado dos problemas adicionales usando la matriz de distancias reales entre aeropuertos y cambiando valores para ver cómo interactúa el planificador. Tras crear un método que me copia las distancias, puesto que al crear la distancia de Granada a Barcelona, no creaba la de Barcelona a Granada, he desistido. Al haber comprobado que funciona correctamente con los ejemplos anteriores, añadiendo personas y más aviones, es obvio que también funcionará si le cambiamos el nombre a las ciudades.

Conclusiones

Hemos aprendido a usar una herramienta muy potente que puede ser utilizada para automatizar sistemas basados en reglas. El pequeño problema es que al ser un dominio cerrado, el pasar por alto un caso puede hacer que todo el sistema falle, no llegando a concluir. Lo bueno es que es una herramienta muy eficiente que nos muestra los pasos seguidos y a la cual podemos añadir heurísticas para obtener resultados diferentes.