Práctica 3 de Inteligencia Artificial
Planificación

Curso 2021-2022 1Q

Miguel García Soler
Pau López Castillón
Adrián Rubio Rodríguez

Índice

1. Descripción del problema

Una central de reservas de un hotel necesita un sistema capaz de asignar peticiones de reserva a las habitaciones de las que dispone, siguiendo diferentes criterios y restricciones. Para simplificar consideraremos que las reservas son únicamente para un mes concreto (30 días).

Una habitación está descrita por su identificador y el número de personas que puede alojar (entre 1 y 4). Una reserva está descrita por su identificador, el número de personas en la reserva (entre 1 y 4) y los días de inicio y final de la reserva (entre el 1 y el 30).

2. Dominio

La solución que planteamos (extensión 4) distingue entre peticiones de reserva (identificadas por el día de inicio y final y el número de personas en la reserva), y habitaciones (un identificador unívoco de cada habitación con su capacidad máxima). El resto de elementos de la solución están descritos en la siguiente lista:

- Tipos
  - **peticion**
  - **habitacion**
- Funciones
  - **dia-inicio**, número entre 1 y 30 que representa el primer día de una reserva.
  - **dia-final**, número entre 1 y 30 que representa el último día de una reserva.
  - **capacidad**, número máximo de personas que caben en una habitación.
  - **cantidad**, número de personas en una petición de reserva.
  - **n-denegadas**, el total de peticiones de reserva no asignadas.
  - **n-sobrantes**, número total de plazas desperdiciadas.
  - **n-abiertas**, total de habitaciones abiertas.
- Predicados
  - **servida**, nos dice si una petición de reserva ha sido considerada. Es útil en la definición del *goal* de la versión básica del problema, ya que el objetivo es asignar todas las peticiones o ninguna. Además sirve como filtro de aquellas peticiones que quedan por atender.
  - **asignada-a**, indica a qué habitación ha sido asignada una petición de reserva. Facilita la tarea de comprobación de restricciones de solapamiento a la hora de asignar una petición pendiente.
  - **abierta**, informa de si una habitación está abierta.
- Acciones
  - **asignar**, dada una petición de reserva busca y le asigna una habitación con capacidad suficiente para alojar al número de personas de la petición y que no se solape con ninguna de las peticiones ya asignadas a esa habitación.
  - **denegar**, marca una petición como servida sin asignarla a ninguna habitación. La idea es poder obtener soluciones que dejan peticiones sin asignar con tal de optimizar el número total de peticiones asignadas.
  - **abrir**, establece una habitación como abierta que es requisito necesario para que se le pueda asignar una petición de reserva.

Asumimos que las fechas de una petición de reserva son siempre correctas, es decir, el día final de una reserva es siempre estrictamente mayor que el día de inicio. De esta manera la mínima reserva que se puede hacer es de un día y nos ahorramos hacer comprobaciones innecesarias. En consecuencia, no se considera solapamiento que el día final de una reserva coincida con el día de inicio de otra. Se entiende que a los clientes se les hace marchar el día final antes de cierta hora para que los siguientes puedan ocupar la habitación.

2.1. Extensiones del dominio

Para la segunda extensión del enunciado de la práctica, se pedía dotar a las habitaciones de una dirección y poder especificar en las peticiones qué dirección se prefiere. Para cumplir estos requisitos hemos ampliado el dominio de la lista anterior con los siguientes elementos:

- Tipos
  - **direccion**
- Funciones
  - **n-mal-orientadas**, número total de peticiones de reserva que se han asignado y no cumplen las preferencias de orientación.
- Predicados
  - **orientada**, predicado polimórfico que indica la dirección donde apunta una habitación, o la orientación que se pide en el caso de las peticiones.

Cabe destacar que el enunciado solo pedía cubrir cuatro direcciones distintas (N/S/E/O), pero el modelo que hemos desarrollado permite trabajar con tantas direcciones como se desee, solo hay que incluirlas en la declaración de objetos al inicio del problema.

3. Instancias del problema

Solo se necesitan dos tipos de objetos para crear un problema que pueda resolver el dominio que hemos modelado: peticiones y habitaciones, a excepción de la extensión 2, en cuyo caso necesitamos un tercer tipo de objetos: direcciones.

El estado inicial consiste en una enumeración de objetos de tipo *peticion*, que representan cada una de las peticiones de reserva, y objetos de tipo *habitacion*, que representan cada una de las habitaciones disponibles del hotel. Por cada petición hay que definir cuál es la cantidad de personas, el día de inicio y el día final de la reserva (y la dirección deseada en el caso de la segunda extensión). Por cada habitación, únicamente hay que definir cuál es su capacidad máxima (y la dirección a la que apunta en el caso de la segunda extensión).

La definición del estado final es bien sencilla, se tiene que cumplir el predicado *servida* por todas y cada una de las peticiones definidas.

Finalmente, se define una combinación lineal de criterios que servirá de guía al planificador para escoger qué acción aplicar entre estados hasta poder llegar al estado final, si es que es posible.

4. Desarrollo de los modelos

Para la creación del modelo final definido en este documento, hemos optado por un diseño incremental basado en un prototipo inicial que cumple los requisitos básicos del problema, y que poco a poco hemos ido adaptando a las distintas necesidades del guión de la práctica. En cada extensión hemos añadido elementos nuevos que han servido para construir el modelo final, a excepción de la extensión 2 que, según nuestra interpretación del enunciado, parecía una apartado opcional pero que igualmente hemos hecho.

Cabe destacar que la implementación del modelo se ha hecho sin desarrollar diversos prototipos entre extensiones, ya que la dificultad que requerían nos permitía saltar de extensión en extensión en pocos pasos. Además hemos desarrollado un generador de problemas que hemos ido adaptando para las extensiones 2, 3 y 4. Cuyo funcionamiento queda descrito en el siguiente apartado.

5. Juegos de prueba

5.1. Prueba básica

Este es el código que hemos utilizado para probar la solución del nivel básico del problema. Solo lo mostraremos en esta ocasión, con tal de ilustrar la notación empleada que facilita la posterior interpretación de los pasos que devuelve el planificador.

```
(define (problem prueba00)
    (:domain ReservasHotel)

    (:objects
        pet1_c2_12 pet2_c2_14 pet3_c2_230 - peticion
        hab1_c2 hab2_c2                   - habitacion)

    (:init
        (= (cantidad   pet1_c2_12)   2)
        (= (dia-inicio pet1_c2_12)   1)
        (= (dia-final  pet1_c2_12)   2)
        (= (cantidad   pet2_c2_14)   2)
        (= (dia-inicio pet2_c2_14)   1)
        (= (dia-final  pet2_c2_14)   4)
        (= (cantidad   pet3_c2_230)  2)
        (= (dia-inicio pet3_c2_230)  2)
        (= (dia-final  pet3_c2_230) 30)

        (= (capacidad hab1_c2) 2)
        (= (capacidad hab2_c2) 2)
    )

    (:goal
        (forall (?p - peticion) (servida ?p)))
)
```

Para el resto de pruebas solo mostraremos una tabla con el *input* y el correspondiente *output* del planificador.

	cantidad	**dia-inicio**	**dia-final**
Petición 1	2	1	2
Petición 2	2	1	4
Petición 3	2	2	30

	capacidad
Habitación 1	2
Habitación 2	2

Con esta entrada el planificador encuentra una posible solución que es la siguiente secuencia de pasos:

```
step  0: ASIGNAR PET3_C2_230 HAB2_C2
      1: ASIGNAR PET2_C2_14 HAB1_C2
      2: ASIGNAR PET1_C2_12 HAB2_C2
```

5.1.1. Explicación

En esta prueba queríamos comprobar el correcto funcionamiento de las restricciones de solapamiento. Como podemos observar, las peticiones PET1 y PET2 han sido correctamente asignadas a habitaciones distintas puesto que tienen rangos de fechas que se solapan. PET3 se solapa con PET2, y a pesar de haber solo dos habitaciones, puede ser asignada a HAB2 ya que no se solapa con PET1. También hemos comprobado que si modificamos PET1 para que se solape con PET3, el planificador no es capaz de encontrar ninguna solución, que es el resultado que esperábamos.

Otra modificación que hemos probado es hacer que la cantidad de personas en una petición sea mayor que la capacidad de cualquiera de las habitaciones del hotel, en cuyo caso, el planificador es capaz de simplificar el objetivo a falso y termina de inmediato.

5.2. Prueba 1

	cantidad	**dia-inicio**	**dia-final**
Petición 1	2	1	2
Petición 2	2	2	3
Petición 3	2	3	4
Petición 4	2	2	30

	capacidad
Habitación 1	2

Solución obtenida por el planificador:

```
step  0: ASIGNAR PET1_C2_12 HAB1_C2
      1: DENEGAR PET4_C2_230
      2: ASIGNAR PET3_C2_34 HAB1_C2
      3: ASIGNAR PET2_C2_23 HAB1_C2
```

5.2.1. Explicación

La intención de esta prueba es crear un escenario donde no sea posible asignar todas las peticiones, y forzar al planificador a escoger cuáles asignar y cuáles descartar. En este caso la petición crítica es la número cuatro. Es la única que se solapa con otras dos, y si se asigna, supone tener que descartar PET2 y PET3. Como podemos observar, la solución que el planificador encuentra (habiendo indicado que minimice el número total de peticiones denegadas) descarta PET4 tal y como esperábamos.

5.3. Prueba 2

Para esta prueba y las siguientes usaremos el generador que hemos desarrollado que genera entradas aleatorias y acepta algunos parámetros. Todos los experimentos son reproducibles incluso sin disponer del archivo de problema, solo hay que pasar como argumento la semilla adecuada al generador.

	cantidad	**dia-inicio**	**dia-final**	**orientacion**
Petición 1	4	15	27	N
Petición 2	1	5	9	S
Petición 3	1	3	4	N
Petición 4	1	6	23	O

	capacidad	**orientacion**
Habitación 1	4	N
Habitación 2	1	E
Habitación 3	1	S

A continuación proporcionamos el comando completo para obtener esta entrada (el símbolo '$' representa el *prompt* del terminal):

```
$ python gen-e2.py --seed 1234 --pet 4 --hab 3
```

Cabe notar que las entradas que se obtienen, no solo están condicionadas por el número de habitaciones y peticiones, sino también por la cantidad de personas en las peticiones. Internamente el generador garantiza que, como mínimo, siempre haya una habitación con suficiente capacidad para recibir cualquiera de las peticiones. Hemos tomado esta decisión con tal de reducir la probabilidad de generar problemas triviales, que no aportan demasiada información a los experimentos.

5.3.1. Explicación

Analizando la entrada esperábamos que las peticiones PET1 y PET3 fueran asignadas a la habitación HAB1, puesto que está orientada al norte y las peticiones no se solapan. También esperábamos que PET2 fuera asignada a HAB2, ya que tiene la orientación y capacidad requerida. Esto dejaría a HAB3 como la única opción posible para PET4, dado que se solapa con PET2 y PET1.

Efectivamente, esta es exactamente la solución que encuentra el planificador, después de haber jugado un poco con el peso de los criterios para acabar de refinar los resultados.

```
step  0: ASIGNAR PET3 HAB1
      1: ASIGNAR PET4 HAB2
      2: ASIGNAR PET2 HAB3
      3: ASIGNAR PET1 HAB1
```

Nótese cómo en este caso no ha hecho falta denegar ninguna petición.

5.4. Prueba 3

Para esta prueba hemos escogido manualmente los valores de cantidad y capacidad, el resto han sido generados a partir de la semilla 1651.

	cantidad	**dia-inicio**	**dia-final**
Petición 1	1	5	12
Petición 2	2	9	20
Petición 3	1	8	27

	capacidad
Habitación 1	1
Habitación 2	2
Habitación 3	3
Habitación 4	4
Habitación 5	2

La solución que encuentra el planificador es la siguiente:

```
step  0: ASIGNAR PET3 HAB1
      1: ASIGNAR PET1 HAB2
      2: ASIGNAR PET2 HAB5
```

5.4.1. Explicación

En esta prueba queríamos centrarnos en la distribución de las peticiones cuando hay exceso de habitaciones. Obviamente, dependerá del peso que se le haya dado a cada criterio. La combinación lineal de las métricas que hemos usado en este caso es:

$$1.0 * denegadas + 0.25 * sobrantes$$

Podemos ver claramente, como tiene más peso el hecho de no asignar una petición, que asignar una habitación con más plazas de las necesarias. En consecuencia, la solución del planificador no descarta ninguna petición y da prioridad a las habitaciones 1, 2 y 5, que son las tres que se ajustan mejor a la cantidad de personas en las peticiones.

Si intercambiamos los pesos de los criterios, es interesante observar que la nueva solución descarta la petición 1, ya que no se puede asignar a una habitación sin desperdiciar espacio.

```
step  0: ASIGNAR PET3 HAB1
      1: DENEGAR PET1
      2: ASIGNAR PET2 HAB2
```

5.4. Prueba 4

	cantidad	dia-inicio	dia-final
Petición 1	4	23	24
Petición 2	2	15	21
Petición 3	1	17	28
Petición 4	3	11	20
Petición 5	1	22	27
Petición 6	4	1	6

	capacidad
Habitación 1	4
Habitación 2	3
Habitación 3	3
Habitación 4	4
Habitación 5	4
Habitación 6	2

Esta entrada la hemos generado con el siguiente comando:

```
$ python gen-e4.py -o prueba04 --seed 2345 --pet 6 --hab 6
```

La solución que encuentra el planificador es:

```
step  0: ABRIR HAB5
      1: ABRIR HAB6
      2: ASIGNAR PET4 HAB5
      3: ASIGNAR PET5 HAB5
      4: ASIGNAR PET6 HAB5
      5: DENEGAR PET3
      6: ASIGNAR PET2 HAB6
      7: ASIGNAR PET1 HAB5
```

5.4.1. Explicación

Analizando la entrada y teniendo en cuenta que la combinación de criterios a minimizar es la siguiente:

$$1.2 * denegadas + 0.2 * sobrantes + 0.5 * abiertas$$

Esperábamos que las peticiones 2 y 5 fueran asignadas a la habitación 5, ya que tiene una capacidad que se ajusta al tamaño requerido, por tanto implica no desperdiciar plazas, y las peticiones no se solapan (favorece el no tener que abrir habitaciones nuevas).

Sin embargo, no esperábamos que la solución fuera a descartar alguna petición como en el caso de la número 3. Lo que significa que el peso del criterio *denegadas* es demasiado bajo, puesto que las soluciones tienen que priorizar asignar el máximo número de peticiones posible aunque se desperdicien plazas o se abran más habitaciones. Si lo subimos hasta 1.5 la nueva solución sustituye el paso 5 por:

```
      5: ABRIR HAB4
      6: ASIGNAR PET3 HAB4
```

6. Escalabilidad del modelo

En este apartado estudiaremos cómo afecta el tamaño de la entrada al tiempo de ejecución del planificador. Utilizaremos el modelo descrito en este documento con el que realizaremos una série de pruebas donde iremos incrementando el tamaño de la entrada de los problemas. Un aspecto importante a tener en cuenta es que *metric-ff*, el planificador usado en este experimento, tiene un límite en cuanto al número de líneas que puede tener el archivo de problema. Esto ya da una ligera idea del tipo de limitaciones que presenta el abordar un problema con este tipo de herramientas.

6.1. Gráfico

7. Conclusión

Hemos resuelto un problema de planificación a partir de un modelo y hemos usado el mismo sistema de planificación para su resolución automática. Hemos podido comprobar que este tipo de sistemas son capaces de ofrecer al usuario una buena planificación en un tiempo de ejecución razonable, aunque el tamaño de los problemas que puede resolver no es demasiado escalable.