Facultad de informática de Barcelona

Práctica de planificación

IA - Subgrupo 12

Marc Nebot
Víctor Teixidó
Florian Vogel

Profesor: Ignasi Gómez
Grado en ingeniería informática
Curso 2021 - 2022

Índice

1. Identificación

En esta práctica, se nos plantea generar una central de reservas de un hotel mediante la cual se puedan tratar distintas peticiones de reserva y asignarlas a las habitaciones disponibles del hotel en cuestión. Cabe destacar que las peticiones de las reservas se entienden como en un único mes, es decir del día 1 al día 30 y que, tanto las reservas como las habitaciones, están limitadas por un número máximo de personas, este límite será de 4 personas.

Inicialmente, las reservas contarán con 3 atributos que las definirán, la cantidad de gente que participa en dicha reserva y los días de entrada y salida del hotel, por otra parte, las habitaciones contarán inicialmente con un único atributo, la capacidad de la habitación. Más adelante, en algunas de las extensiones, tanto reservas como habitaciones podrán tener atributos adicionales para poder tratar con nuevas restricciones impuestas por el problema. Además, a la hora de asignar las reservas, hay que tener en cuenta que la capacidad de la habitación sea igual o superior a la demanda de la reserva y que dicha habitación esté disponible para los días de la reserva en cuestión.

Clasificamos este proyecto como un problema de planificación debido a las siguientes características que presenta el problema. Se nos plantea realizar la búsqueda hasta una solución deseada a partir de unos datos que forman una situación inicial, este camino o transformación se realizará a través de una serie de acciones predefinidas en el dominio del problema, estas definirán el plan a seguir para llegar a la solución encontrada. En nuestro caso, la solución vendrá dada como una lista de acciones, que podrán realizarse simultáneamente, mediante la cual el programa ha encontrado una solución final deseada y óptima.

Los problemas de planificación se establecen por un conjunto de operadores, un estado inicial y el objetivo final a alcanzar o a optimizar. En esta práctica, el conjunto de operadores variarán según qué extensión estemos realizando y dichos operadores no servirán para, a

través de estos, definir el plan final para alcanzar el objetivo. El estado inicial vendrá definido por una serie de instancias de reservas y habitaciones que definirán el planteamiento inicial, esté estado se podrá generar de forma aleatoria gracias al generador hecho. Por último, en el objetivo se definirá el estado a alcanzar o los atributos a optimizar, con dicha información, el programa que hemos diseñado en *PDDL* será capaz de, a través de los operadores definidos, ir trazando una ruta o plan con la cual alcanzaremos una solución deseada.

1.1 Dominio

El dominio representará el funcionamiento de cada uno de los "niveles" especificados en el enunciado de la práctica. En cada uno de los dominios tendremos representados los predicados, las funciones y las acciones necesarias para el correcto funcionamiento de dicha etapa. Empezaremos describiendo todas las propiedades del dominio inicial, el nivel básico, más adelante iremos especificando los añadidos que se han realizado en las distintas extensiones que se pedían.

Inicialmente, se nos pide crear un modelo capaz de asignar las reservas dadas a las habitaciones disponibles de tal manera que, no se produzcan solapamientos y las personas quepan en las habitaciones asignadas. Además, en caso de que alguna de las asignaciones no sea posible, no se tratará ninguna de las reservas hechas por los clientes.

Hemos establecido con dos predicados que nos permiten la representación del estado del problema en todo momento:

```
(:predicates
        (assigned ?bking - booking ?rm - room)
        (served ?bking - booking)
)
```

El primero de los predicados, *assigned*, indica a qué habitación ha sido asignada una reserva en caso de que esta reserva haya sido ya tratada. Esto nos permite comprobar y evitar los

posibles solapamientos que se puedan producir entre las distintas reservas asignadas. Por otro lado, el predicado *served* nos da información sobre si una reserva ha sido ya tratada o no, a pesar de que esta información puede parecer redundante con el predicado anterior, no lo es, nos será de utilidad en la definición de objetivo a conseguir donde definiremos que queremos que todas las reservas hayan sido tratadas. Además, en futuras extensiones que una reserva haya sido tratada no tendrá porqué ser sinónimo de que haya sido asignada a alguna de las habitaciones.

Las funciones nos son de utilidad para representar atributos numéricos tanto de las reservas como de las habitaciones. Contamos con las siguientes funciones:

```
(:functions
        (peopleBooking ?bking - booking)
        (iniDay ?bking - booking)
        (finDay ?bking - booking)
        (peopleRoom ?rm - room)
)
```

La primera función representa el número de personas que forman parte de una reserva, la segunda y tercera función define el día de llegada y salida de una reserva al hotel y, por último, la última explicita la cantidad de personas que caben en una habitación.

Por último tenemos las acciones, estas son los operadores que se utilizan para poder llegar a un plan final deseado y óptimo, en el nivel básico contaremos con una única acción *assignRoom*. Se encarga de asignar una habitación a una reserva siempre y cuando está habitación pueda albergar al número de personas de la reserva y no se produzca ningún tipo de solapamiento con alguna reserva ya asignada a dicha habitación. Cuando se haya encontrado una habitación disponible para la reserva, se marcará esta como servida.

1.2 Problema

El problema representa una instancia de un estado inicial en el que además, declaramos un objetivo a alcanzar u optimizar. En el problema, definimos unos objetos y a partir de estos y del dominio con el que trabajamos, el programa realiza la búsqueda de un plan que satisfaga nuestros objetivos previamente especificados. Para cada una de las reservas y habitaciones instanciadas, hay que declarar los atributos que hemos mencionado anteriormente, también habrá que definiar las reservas como no asignadas incialmente. Aquí mostramos un pequeño ejemplo del instanciamiento de una habitación y dos reservas, además del estado final a alcanzar para el nivel básico con el que estamos trabajando.

```
(:objects
        room1 - room
        booking1 booking2 - booking)

    (:init
        (not (served booking1))
        (not (served booking2))

        (= (peopleBooking booking1) 1)
        (= (iniDay booking1) 1)
        (= (finDay booking1) 3)

        (= (peopleBooking booking2) 2)
        (= (iniDay booking2) 4)
        (= (finDay booking2) 9)

        (= (peopleRoom room1) 3)
    )

    (:goal
        (forall (?bking - booking) (served ?bking))
    )
```

Para todas las demás extensiones la declaración del problema será mayoritariamente idéntica a la que acabamos de ver. Como mucho podrán aparecer nuevos atributos de las reservas o habitaciones, como en la extensión 2, o habrá que añadir una métrica a optimizar.

Adicionalmente a todo esto, hemos diseñado un generador de problemas el cual, indicando el nivel deseado, generará un fichero problema con instancias de reservas y habitaciones aleatorias. Esto permite testear con mayor veracidad el buen funcionamiento del código diseñado. Esta generación mantendrá la restricción de un máximo de 4 personas tanto en reservas como en la capacidad de las habitaciones.

2. Problema básico y extensiones

Para el desarrollo tanto del nivel básico como del resto de las extensiones, se ha realizado un diseño incremental tal y como indicaba el enunciado de la práctica. Por tanto, primero se ha diseñado el nivel básico y a partir de este, se ha formado la extensión 1. De esta extensión han surgido tanto la extensión 2 como la extensión 3 y por último, a partir de la tercera de las extensiones, se ha diseñado la cuarta versión del programa. Como resulta evidente, cada uno de los nuevos dominios, basados en anteriores, que se han realizado, aportan nuevos añadidos y optimizaciones a la nueva versión.

2.1 Nivel básico

Este nivel lo hemos explicado con más detalle anteriormente en el apartado de dominio y problema. En esta etapa se nos pedía que no se produjeran solapamientos y que la habitación asignada tuviera la capacidad suficiente para albergar a las personas de la reserva. Además todas las reservas tienen que ser asignadas, si existe una sola que no se puede asignar, no se trata ninguna de ellas. Los predicados, funciones y acciones ya han sido descritos anteriormente al igual que la declaración del problema para el dominio del nivel básico.

2.2 Extensión 1

Adicionalmente de los requerimientos del nivel básico, en esta extensión añadimos la optimización del número de reservas asignadas. Es decir, a diferencia de en el nivel básico, ahora será posible que tengamos reservas sin asignar pero nos interesa contar con el máximo número de reservas posibles. Para conseguir esto hemos añadido la acción *notAssignRoom* que, como su nombre indica, no asigna una reserva a una habitación, es decir marca una reserva como tratada pero sin asignarle habitación alguna. Junto a esta nueva acción, hemos añadido una función que recoge el valor de penalización actual del plan, el valor de esta penalización aumenta en 10 cuando una reserva no sea asignada a ninguna de las habitaciones del hotel. El valor en el que aumenta la penalización es un valor totalmente arbitrario que ofrece resultados deseados y esperados.

```
(:action notAssignRoom
        :parameters (?bking - booking)
        :precondition (not (served ?bking))
        :effect (and (served ?bking) (increase (penalitation) 10))
)
```

Fijándonos ahora en el problema, hemos añadido un par de cosas necesarias para el correcto funcionamiento del nuevo dominio diseñado. Por una parte hay que inicializar el valor del nuevo atributo *penalitation* a cero, por otro lado, además del objetivo, hemos definido una métrica a través de la cual indicamos al programa que queremos minimizar al máximo el valor de la penalización en nuestro estado final.

```
(:metric minimize (penalitation))
```

2.3 Extensión 2

La segunda extensión del proyecto que se nos plantea deriva de la extensión anteriormente explicada, en este añadido se nos pide tener en cuenta lo siguiente. Las habitaciones cuentan ahora con un nuevo atributo, la orientación de la habitación que puede ser Norte, Sur, Este u Oeste. Además de las habitaciones, las reservas también tienen ahora dicho atributo en el que se especifica qué orientación tienen como preferencia cada una de las

reservas. Cabe destacar que hemos codificado las orientaciones con números del 1 al 4 en el orden en el que hemos mencionado las orientaciones anteriormente. La máxima prioridad en esta extensión es asignar habitaciones con la orientación deseada de la reserva y lo menos deseado sigue siendo no asignar una reserva a una habitación. Para conseguir esto seguiremos haciendo uso de la función *penalitation* que añadimos en la primera extensión. Además ahora asignar una habitación cualquiera a una reserva tiene una penalización de 5, la penalización de 10 al no asignar una habitación se mantiene. Hemos añadido la función *assignOrientedRoom* para poder garantizar la asignación de habitaciones bien orientadas a las reservas.

```
(:action assignOrientedRoom
        :parameters (?bking - booking ?rm - room)
        :precondition (and      (not (served ?bking))
                                (>= (peopleRoom ?rm) (peopleBooking ?bking))
                                (forall (?auxBking - booking)
                                        (or     (not (assigned ?auxBking ?rm))
                                                (or     (< (finDay ?auxBking) (iniDay
?bking))
                                                        (> (iniDay ?auxBking) (finDay
?bking))
                                                )
                                        )
                                )
                                (= (orientationRoom ?rm) (orientationBooking ?bking))
                        )
        :effect (and (served ?bking) (assigned ?bking ?rm))
)
```

Respecto a la definición del problema que irá vinculado al nuevo dominio, el único cambio que se produce es añadir la instancia que representa la orientación de la habitación y la orientación deseada por las distintas reservas. La descripción del objetivo a alcanzar y la métrica no sufren ningún cambio y por tanto se mantienen como en la primera de las extensiones.

2.4 Extensión 3

La tercera extensión que se nos plantea deriva también de la primera extensión realizada en el proyecto. Además de la optimización de asignaciones que hacemos en dicha extensión, en esta se busca minimizar el desperdicio de número de plazas al asignar una habitación a una reserva, es decir, es mejor cuanto menor sea la diferencia entre la gente que forma parte de la reserva y la capacidad de la habitación, teniendo en cuenta que la capacidad tiene que ser siempre mayor a la demana de personas. Para conseguir esta nueva restricción no hemos tenido que añadir nuevas acciones ni funciones, lo que sí hemos hecho es modificar los valores de las penalizaciones en la acción *assignRoom* y en la *notAssignRoom*. De tal manera que daremos prioridad a aquellas asignaciones que desperdicien un menor número de plazas pero manteniendo la penalización al hecho de dejar una reserva sin asignar. En la función *notAssignRoom* el único cambio que se ha introducido es modificar la penalización de 10 a 15. La penalización que hemos aplicado en la función de asignación es la diferencia entre las plazas de la habitación y la demanda de plazas de la reserva, de esta manera manteniendo la métrica de la extensión 1, damos una valor mayor a aquellas asignaciones que seas más eficientes en lo que a plazas desperdiciadas se refiere.

```
(:action assignRoom
        :parameters (?bking - booking ?rm - room)
        :precondition (and      (not (served ?bking))
                                (>= (peopleRoom ?rm) (peopleBooking ?bking))
                                (forall (?auxBking - booking)
                                        (or     (not (assigned ?auxBking ?rm))
                                        (or     (< (finDay ?auxBking) (iniDay ?bking))
                                                (> (iniDay ?auxBking) (finDay ?bking))
                                                )
                                        )
                                )
                        )
        :effect (and    (served ?bking) (assigned ?bking ?rm)
                (increase (penalitation) (- (peopleRoom ?rm) (peopleBooking ?bking)))
                )
)
```

2.5 Extensión 4

La última extensión del proyecto, se realiza a partir de la tercera extensión, vista en el apartado anterior, en la que se nos pide minimizar el número de plazas desperdiciadas al realizar una asignación. En este añadido se busca minimizar el número de habitaciones que se abren durante el mes, es decir, asignar dos reservas a una misma habitación, siempre que sea posible, será mejor que asignar dichas reservas a habitaciones distintas. Teniendo en cuenta esta nueva optimización y el resto de variables importadas de las extensiones anteriores, hay que equilibrar los pesos por tal de garantizar que obtenemos una solución razonablemente buena y óptima.

En esta última extensión hemos añadido el predicado nuevo *used*, este indica si una habitación ya ha sido utilizada, lo cual nos será de utilidad para saber si una habitación ha sido abierta previamente por otra reserva o, por lo contrario, nadie ha hecho uso de ella. Además de este predicado, hemos diseñado una nueva acción, *assignNewRoom*, como su nombre indica en está acción se asignan habitaciones que tienen el valor de *used* como falso, es decir, que todavía no tienen ninguna reserva asignada para ese mes. Añadir una habitación nueva, tiene una penalización de 5, la penalización de 15 al no asignar una reserva se mantiene de la extensión 3.

```
(:action assignNewRoom
        :parameters (?bking - booking ?rm - room)
        :precondition (and (not (served ?bking)) (>= (peopleRoom ?rm) (peopleBooking ?bking)) (not (used ?rm)))
        :effect (and      (served ?bking) (assigned ?bking ?rm) (used ?rm)
                (increase (penalitation) (- (peopleRoom ?rm) (peopleBooking ?bking)))
                        (increase (penalitation) 5)
                )
)
```

Respecto a la descripción del problema que estará vinculado a este dominio no sufre ningún cambio respecto al anterior ya que no inicializar el valor de *used*, equivale a inicializarlo como falso que es justamente lo que queremos. Respecto a la definición del objetivo y la

métrica, tampoco se produce ningún cambio ya que seguimos con la búsqueda de la minimización de la función *penalitation*.

3. Juegos de prueba

3.1 Nivel básico

Para el nivel básico, recordemos que existe la restricción que nos fuerza a distribuir todas las reservas en algunas de las habitaciones disponibles del hotel. Si se da el caso en que alguna de las reservas no puede ser asignada, entonces ninguna de las reservas será tratada y asignada por el planificador. Hemos ideado dos problemas a partir del generador diseñado, uno dará una solución válida y el segundo no encontrará solución alguna.

Prueba 1

Como podemos ver en el primer problema del nivel básico, tenemos tres reservas para cuatro habitaciones, como resulta evidente, el planificador debería encontrar una solución posible y eso es lo que ocurre exactamente. Si nos fijamos en la salida por consola del programa, podemos ver que se encuentra una asignación válida para nuestro objetivo, asignar todas las reservas. Podemos ver además, que en la forma en la que ha precedido, hay 2 habitaciones que no han sido utilizadas.

ff: found legal plan as follows
step 0: ASIGNROOM BOOKING0 ROOM2
1: ASIGNROOM BOOKING2 ROOM0
2: ASIGNROOM BOOKING1 ROOM2

time spent: 0.00 seconds instantiating 0 easy, 6 hard action templates
0.00 seconds reachability analysis, yielding 24 facts and 6 actions
0.00 seconds creating final representation with 18 relevant facts, 0 relevant fluents
0.00 seconds computing LNF
0.00 seconds building connectivity graph
0.00 seconds searching, evaluating 12 states, to a max depth of 0
0.00 seconds total time

Prueba 2

A diferencia de en la anterior prueba, en esta buscamos un fichero problema que dé una no solución. Para conseguir esto, hemos decidido indicarle al generador de problemas que instancie una única habitación y seis reservas, de tal manera que las probabilidades de solapamiento se incrementen considerablemente. Haciendo esto, y como podemos ver en la salida del programa, eso es justamente lo que ha ocurrido. El planificador ha ido asignando reservas hasta que ha alcanzado una situación en la que se producía un solapamiento ineludible, por tanto, no se asigna ninguna reserva.

ff: goal can be simplified to FALSE. No plan will solve it

3.2 Extensión 1

Para esta primera extensión, recordemos que buscamos optimizar el número de reservas asignadas, por tanto es posible que haya reservas sin una habitación asignada. Para esta extensión hemos realizado dos pruebas distintas en las que hemos probado distintos casos especiales que podrían darse.

Prueba 1

En esta primera prueba hemos decidido generar un fichero de problema con una única habitación y cinco reservas. Si nos fijamos en el fichero problema, adjunto en el proyecto, veremos que tenemos dos reservas que ocupan todo el mes y otras tres que no se solapan nunca. Viendo nuestro objetivo y nuestra métrica, parece lógico que el planificador coja las tres reservas en vez de solo dos. Fijándonos en la salida podemos comprobar que esto es exactamente lo que sucede.

metric established (normalized to minimize): ((1.00[RF0](PENALITATION)) - () + 0.00)*

checking for cyclic := effects --- OK.

*ff: search configuration is best-first on 1*g(s) + 5*h(s) where*
metric is ((1.00[RF0](PENALITATION)) - () + 0.00)*

advancing to distance: 5
4
3
2
1
0

ff: found legal plan as follows

step 0: NOTASIGNROOM BOOKING0
1: ASIGNROOM BOOKING2 ROOM0
2: NOTASIGNROOM BOOKING1
3: ASIGNROOM BOOKING4 ROOM0
4: ASIGNROOM BOOKING3 ROOM0

time spent: 0.00 seconds instantiating 5 easy, 5 hard action templates
0.00 seconds reachability analysis, yielding 20 facts and 10 actions
0.00 seconds creating final representation with 20 relevant facts, 1 relevant fluents
0.00 seconds computing LNF
0.00 seconds building connectivity graph
0.00 seconds searching, evaluating 64 states, to a max depth of 0
0.00 seconds total time

Prueba 2

Este segundo juego de prueba está basado en el primero ya que se trata de una ampliación. En este caso hemos añadido una segunda habitación pero con tan solo 2 plazas. Si nos fijamos, las tres reservas de la prueba anterior demandan una o dos plazas, y las dos reservas que ocupaban todo el mes demandan 4 plazas. Viendo esto, una solución poco eficiente sería asignar las tres reservas a la habitación de 4, ya que estaríamos perdiendo posibles asignaciones adicionales. Viendo el output resultante, podemos ver como la solución es la más eficiente posible y la esperada, se asignan todas las reservas a alguna habitación.

metric established (normalized to minimize): ((1.00[RF0](PENALITATION)) - () + 0.00)*

checking for cyclic := effects --- OK.

*ff: search configuration is best-first on 1*g(s) + 5*h(s) where*
metric is ((1.00[RF0](PENALITATION)) - () + 0.00)*
advancing to distance: 5
4
3
2
1
0
ff: found legal plan as follows
step 0: ASIGNROOM BOOKING4 ROOM1
1: ASIGNROOM BOOKING3 ROOM1
2: ASIGNROOM BOOKING2 ROOM1
3: ASIGNROOM BOOKING1 ROOM0
4: ASIGNROOM BOOKING0 ROOM0
time spent: 0.00 seconds instantiating 5 easy, 9 hard action templates
0.00 seconds reachability analysis, yielding 29 facts and 14 actions
0.00 seconds creating final representation with 28 relevant facts, 1 relevant fluents
0.00 seconds computing LNF
0.00 seconds building connectivity graph
0.00 seconds searching, evaluating 39 states, to a max depth of 0
0.00 seconds total time

3.3 Extensión 2

Tal y como hemos explicado anteriormente, en esta extensión se busca no solo maximizar las reservas asignadas sino que además, nos interesa maximizar las reservas asignadas a habitaciones con orientación deseada. Para conseguir esto hemos hecho uso de una métrica la cual ya hemos explicado con anterioridad. Hemos planteado dos juegos de pruebas para comprobar el buen y correcto funcionamiento de esta extensión.

Prueba 1

En esta primera prueba hemos planteado el siguiente caso, recordar que están los ficheros problema adjuntos al proyecto. Contamos con una única habitación y cuatro reservas, una de estas reservas está bien orientada pero impide la asignación de cualquier otra de las reservas, las otras tres están mal orientadas. Tal y como está planteado el enunciado, y por tanto, tal y como hemos planteado la métrica, se debería dar prioridad a la reserva bien orientada, a pesar de impedir la asignación de nuevas reservas. Tal y como vemos en la salida, eso es lo que sucede.

metric established (normalized to minimize): ((1.00[RF0](PENALITATION)) - () + 0.00)*
checking for cyclic := effects --- OK.
*ff: search configuration is best-first on 1*g(s) + 5*h(s) where*
metric is ((1.00[RF0](PENALITATION)) - () + 0.00)*
advancing to distance: 4
3
2
1
0
ff: found legal plan as follows
step 0: NOTASIGNROOM BOOKING4
1: NOTASIGNROOM BOOKING3
2: NOTASIGNROOM BOOKING2
3: ASIGNORIENTEDROOM BOOKING1 ROOM1
time spent: 0.00 seconds instantiating 4 easy, 5 hard action templates
0.00 seconds reachability analysis, yielding 16 facts and 9 actions
0.00 seconds creating final representation with 16 relevant facts, 1 relevant fluents
0.00 seconds searching, evaluating 25 states, to a max depth of 0
0.00 seconds total time

Prueba 2

En esta segunda prueba, hemos modificado el problema anterior con los siguientes añadidos. Tenemos una nueva habitación orientada también en la misma dirección que la otra, pero en esta nueva habitación solo cabe una persona. La reserva que coincide con dichas orientaciones es una reserva de una única habitación, el resto de reservas, en cambio,

demandan más capacidad. Por tanto, para ser eficientes lo mejor sería asignar dicha reserva a la habitación de capacidad uno y el resto de reservas, a pesar de que la orientación no sea la correcta, asignar a la otra habitación, de esta manera conseguimos el máximo número de asignaciones. Como podemos comprobar, la salida es la correcta y esperada.

metric established (normalized to minimize): ((1.00[RF0](PENALITATION)) - () + 0.00)*
checking for cyclic := effects --- OK.
*ff: search configuration is best-first on 1*g(s) + 5*h(s) where*
metric is ((1.00[RF0](PENALITATION)) - () + 0.00)*
advancing to distance: 4
3
2
1
0

ff: found legal plan as follows

step 0: ASIGNROOM BOOKING4 ROOM1
1: ASIGNROOM BOOKING3 ROOM1
2: ASIGNROOM BOOKING2 ROOM1
3: ASIGNORIENTEDROOM BOOKING1 ROOM2

time spent: 0.00 seconds instantiating 4 easy, 7 hard action templates
0.00 seconds reachability analysis, yielding 21 facts and 11 actions
0.00 seconds creating final representation with 18 relevant facts, 1 relevant fluents
0.00 seconds computing LNF
0.00 seconds building connectivity graph
0.00 seconds searching, evaluating 27 states, to a max depth of 0
0.00 seconds total time

3.4 Extensión 3

Prueba 1

En esta prueba hemos decidido comprobar que nuestro sistema no desperdicia plazas, por lo tanto, hemos considerado el siguiente caso. Imaginemos que tenemos 3 habitaciones con sus respectivas personas permitidas,3,2 y 2 respectivamente y nuestros 4 solicitantes que piden habitaciones de 1 persona, 3 personas, 2 personas y 2 personas. Todos han solicitado habitaciones en las mismas fechas por lo tanto nuestro sistema tendría que ser capaz de no asignar la solicitud de una persona pues desperdiciaríamos plazas. Podemos observar en la salida que esto efectivamente se cumple.

metric established (normalized to minimize): ((1.00[RF0](PENALITATION)) - () + 0.00)*
checking for cyclic := effects --- OK.
*ff: search configuration is best-first on 1*g(s) + 5*h(s) where*
metric is ((1.00[RF0](PENALITATION)) - () + 0.00)*
advancing to distance: 4
3
2
1
0
ff: found legal plan as follows
step 0: ASIGNROOM BOOKING2 ROOM1
1: ASIGNROOM BOOKING1 ROOM2
2: NOTASIGNROOM BOOKING0
3: ASIGNROOM BOOKING3 ROOM0
time spent: 0.00 seconds instantiating 4 easy, 378 hard action templates
0.00 seconds reachability analysis, yielding 30 facts and 14 actions
0.00 seconds creating final representation with 28 relevant facts, 1 relevant fluents
0.00 seconds computing LNF
0.00 seconds building connectivity graph
0.00 seconds searching, evaluating 98 states, to a max depth of 0
0.00 seconds total time

Prueba 2

En esta prueba hemos decidido comprobar que nuestro sistema no desperdicia plazas, por lo tanto, hemos considerado el siguiente caso. Imaginemos que tenemos 3 habitaciones con sus respectivas personas permitidas,3,3 y 3 respectivamente y nuestros 4 solicitantes que piden habitaciones de 3 personas, 3 personas, 3 personas y 1 persona. Todos han solicitado habitaciones en las mismas fechas por lo tanto nuestro sistema tendría que ser capaz de no asignar la solicitud de una sola persona pues desperdiciaríamos cosa que no nos podemos permitir en este problema plazas. Podemos observar en la salida que esto efectivamente se cumple.

*ff: search configuration is best-first on 1*g(s) + 5*h(s) where*
metric is ((1.00[RF0](PENALITATION)) - () + 0.00)*

advancing to distance: 4
3
2
1
0

ff: found legal plan as follows

step 0: ASIGNROOM BOOKING1 ROOM2
1: ASIGNROOM BOOKING0 ROOM1
2: NOTASIGNROOM BOOKING3
3: ASIGNROOM BOOKING2 ROOM0

time spent: 0.00 seconds instantiating 4 easy, 540 hard action templates
0.00 seconds reachability analysis, yielding 32 facts and 16 actions
0.00 seconds creating final representation with 32 relevant facts, 1 relevant fluents
0.00 seconds computing LNF
0.00 seconds building connectivity graph
0.00 seconds searching, evaluating 193 states, to a max depth of 0
0.00 seconds total time

3.5 Extensión 4

Prueba 1

En esta prueba hemos decidido comprobar que nuestro sistema asigna el mínimo número de habitaciones a las reservas. Imaginemos que tenemos 4 habitaciones con sus respectivas personas permitidas,3,3,3 y 3 respectivamente y nuestros 3 solicitantes que piden habitaciones de 3 personas, 3 personas y 3 personas. Todos han solicitado habitación en fechas no solapadas. Por lo tanto esperamos que nuestro programa asigne a todos los solicitantes la misma habitación con tal de asignar el mínimo número de habitaciones. Podemos observar en la salida que esto efectivamente se cumple.

*ff: search configuration is best-first on 1*g(s) + 5*h(s) where*
metric is ((1.00[RF0](PENALITATION)) - () + 0.00)*
advancing to distance: 3
2
1
0

ff: found legal plan as follows

step 0: ASIGNNEWROOM BOOKING2 ROOM0
1: ASIGNROOM BOOKING0 ROOM0
2: ASIGNROOM BOOKING1 ROOM0
time spent: 0.00 seconds instantiating 15 easy, 12 hard action templates
0.00 seconds reachability analysis, yielding 38 facts and 27 actions
0.00 seconds creating final representation with 38 relevant facts, 1 relevant fluents
0.00 seconds computing LNF
0.00 seconds building connectivity graph
0.00 seconds searching, evaluating 31 states, to a max depth of 0
0.00 seconds total time

Prueba 2

En esta prueba hemos decidido comprobar que nuestro sistema asigna el mínimo número de habitaciones a las reservas. Imaginemos que tenemos 8 habitaciones con sus respectivas personas permitidas,3 en cada habitación y nuestros 5 solicitantes que piden habitaciones de 3 personas. Todos han solicitado habitación en fechas no solapadas. Por lo tanto esperamos que nuestro programa asigne a todos los solicitantes la misma habitación con tal de asignar el mínimo número de habitaciones. Podemos observar en la salida que esto efectivamente se cumple y asigna a todos los solicitantes la room0.

*ff: search configuration is best-first on 1*g(s) + 5*h(s) where*
metric is ((1.00[RF0](PENALITATION)) - () + 0.00)*

advancing to distance: 5
4
3
2
1
0
ff: found legal plan as follows
step 0: ASIGNNEWROOM BOOKING4 ROOM0
1: ASIGNROOM BOOKING0 ROOM0
2: ASIGNROOM BOOKING1 ROOM0
3: ASIGNROOM BOOKING2 ROOM0
4: ASIGNROOM BOOKING3 ROOM0
time spent: 0.00 seconds instantiating 45 easy, 40 hard action templates
0.00 seconds reachability analysis, yielding 106 facts and 85 actions
0.00 seconds creating final representation with 106 relevant facts, 1 relevant fluents
0.00 seconds computing LNF
0.00 seconds building connectivity graph
0.00 seconds searching, evaluating 136 states, to a max depth of 0
0.00 seconds total time

4. Conclusión

Se nos planteó un problema inicial a resolver, la gestión de un sistema de reservas de hotel y se nos propuso técnicas de planificación y el uso de *PDDL* como herramientas para resolverlo. Tal y como hemos visto en el trabajo, estas técnicas son idóneas para este tipo de problemas en los que partimos de un estado inicial y buscamos un plan o una secuencia de acciones mediante las que llegar a un estado final deseado.

Podemos afirmar que nuestro planificador cumple con los objetivos propuestos y esperados, ya que a partir de un instanciamento del problema, nuestro planificador es capaz de encontrar el plan y las acciones a tomar para poder tener una solución razonablemente buena. El planificador nos devuelve las acciones que habría que tomar para alcanzar dicho objetivo y, como hemos visto en los distintos juegos de pruebas efectuados, estas soluciones son correctas y útiles para lo que podría ser un gestor de reservas funcional en la vida real. Suponiendo una hipotética aplicación real, partiendo de las extensiones realizadas se podrían tener en cuenta nuevos factores como podrían ser el precio o los días que se aloja una reserva, para obtener un planificador más detallado y preciso. Habría que cambiar también por tanto, la métrica y los predicados y acciones del dominio.

Por otro lado, también hemos intentado comprobar con nuestro generador que cuanto más aumenta el tamaño del problema más tardará en devolver una planificación. Estos archivos los hemos añadido en la carpeta de testing y hemos podido verificar que cuando el número de habitaciones es mucho menor al número de solicitudes el programa puede tardar un tiempo exponencialmente elevado en generar una planificación correcta. Pues si trabajamos con tamaños pequeños de entre 1 y 8, el planificador lo resuelve medianamente rápido, pero si subimos el tamaño a unas 17 solicitudes y 3 habitaciones sin exceder las 200 líneas de código, el programa tarda tanto que no hemos sido capaces de esperar a que terminase por el elevado tiempo que puede tardar en devolvernos una solución a nuestro problema.