INTELIGENCIA ARTIFICIAL
Práctica de Planificación

Pau Adell Raventós

Marc Falcón Barau

Jaya García Fernández

Otoño 2021 - 2022

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Índice

1. El problema

Una central de reservas de un hotel nos ha pedido un sistema para poder manejar todas las peticiones de reservas a habitaciones. El problema consiste en asignar un conjunto de peticiones de reservas a las habitaciones del hotel según una serie de criterios y restricciones. Inicialmente, para simplificar el problema se considerarán las siguientes pautas:

- las personas que se pueden alojar en una habitación variaran entre 1 y 4
- el número de personas por reserva también variaran entre 1 y 4
- el intervalo de tiempo en el que se puede hacer una reserva es entre 1 y 30 días

Debido a que estas pautas serán idénticas para todos los niveles de implementación, empezaremos explicando cómo hemos modelado estas para resolver el problema y las agregaciones según las expansiones implementadas. Cada expansión requerirá de diferentes criterios y se especificara en cada caso el punto del cual se parte considerando que todos contienen las pautas básicas.

2. Dominio

2.1 Estructura del dominio

Para modelar la base del problema hemos decidido definir dos objetos, *Habitación* y *Reserva*. Para añadir información a cada objeto, como por ejemplo indicar el número de personas que permite una habitación, usaremos una función que nombraremos *capacidadH* que solo contendrán los objetos habitación. Para indicar el número de personas de la reserva también usaremos una función que nombraremos *capacidadR* y, por último, para indicar el periodo de tiempo de la reserva usaremos *diaI* y *diaF* definidos como funciones también y que solo contendrán los objetos reserva. De este modo ya tendríamos casi toda la información de entrada necesaria para solucionar el problema, aunque para resolverlo nos falta un paso muy importante. Debemos indicar el estado de la reserva (la habitación a la que se va a asignar) para poder solucionar el problema.

Para determinar en qué estado se encuentra la reserva, definiremos dos predicados, uno que indique que la reserva esta libre, es decir, no está asignada a ninguna habitación (*libreR*), y otro que indique si la reserva esta asignada y a qué habitación esta asignada (*asignada*). Ahora sí que tendremos toda la información necesaria para poder resolver el problema.

La única acción que necesitaremos para solucionar el problema será la de asignar las reservas a las habitaciones y por ello hemos implementado *asignar_reserva*. Las precondiciones serán las siguientes: la reserva no está asignada a ninguna habitación (*libreR*), la capacidad de la reserva es menor o igual a la capacidad de la habitación y finalmente que no haya una reserva asignada que se solape con la que se quiere asignar en la misma habitación. Una vez comprobadas las condiciones, si se cumplen todas las condiciones, la reserva ya no será *libreR* y pasará a estar *asignada* a una habitación. De este modo si en el problema todas las reservas se pueden asignar a las habitaciones el programa al cumplir todas las restricciones obtendrá una solución. No necesitamos mas acciones pues el problema básico solo requiere de asignar reservas sin ningún criterio

2.2 Estructura del problema

Nuestro problema contendrá solamente dos tipos de objetos, habitaciones y reservas. La cantidad que haya de cada uno ya será variable dependiendo del problema que se quiera resolver. El estado inicial será el estado en el que se añade la información de cada objeto y del cual se parte para resolver el problema. El estado final será cuyo estado no contenga ninguna reserva libre (no haya ninguna reserva que no esté asignada) como se puede ver en la Imagen 1. De este modo, como hemos considerado en este nivel, en el caso de que no se pueda asignar una reserva, no se asignará ninguna ya que no habrá estado final.

```
(:goal (forall (?r2 - reserva) (not(libreR ?r2))))
```

Imagen 1: Estado final del fichero de problemas

3. Modelos del problema

Ahora pasaremos a hablar de los diferentes niveles que hemos implementado del problema. Considerando que partimos del nivel básico explicaremos solamente el que ha sido necesario añadir y aquello que se ha modificado para poder contemplar la optimización que pretende cumplir la extensión. En caso de lo contrario se considerará que la estructura es idéntica al caso más básico.

3.1 Nivel Básico

Se obtiene un conjunto de reservas con su capacidad y su fecha de asentamiento con el objetivo de distribuirlas entre habitaciones sin que se solapen y quepan en las habitaciones (la capacidad de la habitación asignada a una reserva no puede ser menor a la capacidad de la reserva). Para este nivel consideraremos que en el caso de que no se pueda asignar una reserva, no se asignará ninguna para hacerlo más sencillo.

3.2 Extensión 1

Esta extensión se diferencia al nivel básico por su optimización respecto el número de reservas asignadas. En este caso no se contemplará que si una reserva no se puede asignar no se asignen las demás debido a que se pretende maximizar el número de asignaciones posible. En el hipotético caso de que no se pueda asignar una reserva se interpretará que es debido a que no es posible y no habrá otra forma de asignar un mayor número de reservas.

Estructura del dominio:

Primero, se ha añadido una penalización como función que nos servirá para cualificar nuestras soluciones para entonces poder elegir entre ellas cual consideramos la más optima. Como para esta extensión queremos maximizar el número de asignaciones de reservas, en vez de contemplar cuales se han asignado, hemos elegido contemplar aquellas que no se han asignado y por tanto en vez de querer maximizar ese valor, lo queremos minimizar. Queremos que el número de reservas *libreR* sea el menor. Para poder cuantificarlas hemos añadido la acción *descartar_reserva*, su precondición será que la reserva no este asignada, es decir *libreR*, y si se cumple, la reserva ya no será libre e incrementaremos la penalización en uno. De esta forma podremos usar la métrica "*minimize (penalizacion)"* para conseguir la solución con la penalización más pequeña, es decir con el máximo número de reservas asignadas.

Como la acción de asignar reserva aún sigue siendo necesaria y suficiente, no hará falta añadir más acciones a parte de esta y la de descartar.

Estructura del problema:

Hay dos diferencias respecto a la solución base sobre la estructura del problema. La primera es la inicialización de nuestra nueva función penalización que se habrá de definir en el estado inicial con el valor de cero (Imagen 2).

```
(= (penalizacion) 0)
```

Imagen 2: Inicialización de penalización

La segunda y la más importante, la que utilizamos como la que determina el valor de los estados para elegir entre ellos que definiremos como métrica y el cual queremos que nuestro problema minimice. En las siguientes extensiones todas utilizan esta métrica, así que para no repetir no volveremos a mencionarlo ni enseñarlo en ningún juego de prueba.

```
(:metric minimize (penalizacion))
```

Imagen 3: Métrica del problema

3.2 Extensión 2

Esta extensión no parte del nivel básico sino de la extensión 1 en la que se optimiza la distribución de reservas al máximo posible. En esta extensión, se añade la orientación preferente de las habitaciones a la hora de hacer la reserva. Las orientaciones disponibles serán Norte, Sud, Este y Oeste y se pretende optimizar que las asignaciones estén en una habitación con la orientación solicitada. En el caso de que no se pueda asignar en la orientación solicitada, es más importante que se asigne a otra orientación a que no se asigne la reserva.

Estructura del dominio:

En esta extensión, partiendo de la extensión 1, hemos añadido una función tanto para reserva como para habitación que indique su orientación; *orientacionH*, *orientacionR*. Además, hemos modificado las acciones. *Asignar_reserva* se ha eliminado para sustituirla con dos otras acciones que son *asignar_reserva_con_orientacion* y *asignar_reserva_sin_orientacion*, para mantener en todo momento un coste sobre si las asignaciones se corresponden a la orientación solicitada. *Asignar_reserva_con_orientacion*, mantendrá los mismos prerrequisitos que *asignar_reserva*, pero incluye la nueva restricción de si las orientaciones tanto de la reserva como la de la habitación son equivalentes. En cuyo caso las restricciones se cumplan se asignará la reserva del mismo modo que *asignar_reserva*. En cambio, en la acción de *asignar_reserva_sin_orientación* no se tendrá en cuenta la restricción sobre la orientación de la reserva y la de la habitación, pero en este caso no solamente se asignará la reserva como anteriormente, sino que se incrementará nuestro penalizador en uno. Finalmente, la acción de *descartar_reserva* también será modificada para que en vez de incrementar la penalización en uno al no asignar una reserva se incrementará en dos para determinar que será mejor asignar una reserva en una orientación no pedida antes de no asignarla.

Estructura del problema:

La única diferencia respecto la extensión 1 a la hora de generar problemas es la indicación de la orientación a la hora de crear habitaciones y reservas. Las habitaciones estarán en una orientación y las reservas solicitarán una orientación preferente. Como usamos una función numérica hemos decidido que el 0 representara el Norte, el 1 el Este, el 2 el Sud y el 3 el Oeste.

3.3 Extensión 3

En esta expansión que parte de la expansión 1 en la que se optimiza la distribución de reservas al máximo posible también se minimizará el desperdicio de plazas lo máximo posible. En otras palabras, se intentará asignar las reservas a habitaciones con la capacidad más cercana posible, de esta forma, si es posible, sin desperdiciar capacidad de habitaciones.

Estructura del dominio:

La única diferencia en el dominio para esta extensión es la penalización. La variable cumplirá exactamente la misma función, pero se tendrá a cuenta a la hora de asignar una reserva y se cambiará su incremento a la hora de descartar una reserva. En el caso de que se asigne una reserva, penalizaremos en proporción a la diferencia entre la capacidad de la habitación y la capacidad de la reserva. De este modo obtendremos un coste que a medida que se desperdicie más espacio ira augmentando y en el caso de que no se desperdicie no augmentara, considerando que como hemos explicado en la expansión 1 ser quiere minimizar este coste. En el caso de descartar una reserva, es importante cambiar el incremento de la penalización ya que este tiene que penalizar más con relación al desperdiciar muchas plazas, es decir, a la hora de penalizar con un coste de tres al asignar una persona en una habitación de cuatro plazas, el no hacer esta asignación deberá costar mayor a ese tres que penaliza asignar esa reserva.

Estructura del problema:

Al solo modificar los costes de penalización respecto al extensión 1, no hay ninguna diferencia a la hora de definir los problemas.

3.4 Extensión 4

En esta expansión, que parte de la expansión 3 en la que se minimiza el desperdicio de plazas, también se optimizará el número de habitaciones que se ocupan durante el mes. Es decir, serán mejores las soluciones en las que las reservas se asignan a un menor número de habitaciones diferentes.

Estructura del dominio:

El dominio de esta extensión varia con el domino de la extensión 3 debido al añadimiento de la función *numAssig* para cada habitación, que mantendrá el valor de cuantas veces ha sido asignada esa habitación a una reserva cualquiera. Aparte de la adición de esta variable, se cambia la acción *asignar_reserva* de manera que ahora los pesos para los diferentes criterios han sido modificados. La forma en que se mira si una reserva se puede asignar a una habitación es la misma que en las otras extensiones, pero el efecto de respuesta es otro. En este caso, augmentaremos el valor de *numAssig* de la habitación en 1, calcularemos como en la extensión 3 el valor de penalización dependiendo de las plazas, pero a diferencia de la extensión anterior decrementaremos la penalización dependiendo de las veces que se haya asignado esa habitación a una reserva cualquiera, haciendo que las habitaciones que ya han sido asignadas anteriormente tengan ese plus de volver a ser asignadas tal como nos pide el enunciado.

Estructura del problema:

Al solo añadir la variable *numAssig*, lo único que cambiamos es que a la hora de inicializar las diferentes habitaciones tenemos que crear esa variable para todas i ponerles el valor a 0 al principio de la ejecución.

4. Instancias y Juegos de Prueba

En este apartado describiremos todos los juegos de prueba que hemos diseñado para demostrar que cada extensión funciona como debería. Para ello hemos incluido una imagen de la estructura del problema y también la salida que tendría que salir.

4.1 Nivel Básico

Prueba 1:

En la primera prueba comprobaremos que no se pueden asignar reservas en habitaciones donde no hay suficiente espacio. Es un caso muy sencillo por comprobar, pero necesario para confirmar el correcto funcionamiento posteriormente. El ejemplo que hemos escogido es simplemente generar un estado inicial con una habitación de capacidad 2 y una reserva sin asignar de capacidad 4 (y con unos días de alojamiento aleatorios).

Imagen 4: Fichero de problemas y resultado de la prueba 1 del nivel Básico

Como es esperado, el problema no se puede resolver ya que no se puede asignar la reserva a la única habitación que existe (Imagen 4).

En caso contrario, si la capacidad de la habitación es mayor o igual a la de la reserva, el problema se puede resolver y se hace como se ve en la Imagen 5.

Imagen 5: Fichero de problemas y resultado cambiando la capacidad de la habitación

Prueba 2:

En la segunda prueba comprobaremos que no se asignen dos reservas en la misma habitación en caso de que se solapen temporalmente. Para ello generaremos una habitación con capacidad máxima, dos reservas también con capacidad máxima, pero que estas se solapen en el tiempo y las dos estén sin asignar. Aprovecharemos también para confirmar que la comprobación de solapamiento del tiempo también tenga en cuenta que se solapen, aunque sea el último día de una reserva y el primero de la otra (solapados solo 1 día).

```
(:objects H1 - habitacion
          R1 R2 - reserva
)

View
(:init

    (= (capacidadH H1) 4)

    (= (capacidadR R1) 4)
    (= (capacidadR R2) 4)

    (= (diaI R1) 2)
    (= (diaF R1) 4)

    (= (diaI R2) 4)
    (= (diaF R2) 6)


    (libreR R1)
    (libreR R2)
)
```

```
problem proven unsolvable.
```

Imagen 6: Fichero de problemas y resultado de la prueba 2 del nivel Básico

En este caso debido a que las dos reservas requieren de una habitación el dia 4 y solamente hay una disponible, el sistema no puede asignar una de ellas y por ello no puede solucionar el problema (Imagen 6).

En caso de que no se solapen y que el *diaI* de la segunda reserva se mayor al diaF de la primera reserva, por ejemplo, haciendo que (= (*diaI* R2) 5), el programa resolverá el problema:

```
(= (diaI R2) 5)
(= (diaF R2) 6)
```

```
step    0: ASIGNAR_RESERVA H1 R1
        1: ASIGNAR_RESERVA H1 R2
```

Imagen 7: Fichero de problemas y resultado con el cambio de día de inicio de la reserva 2

Prueba 3:

En la tercera prueba demostraremos que en el caso de que una de las reservas no se pueda asignar, no se asignará ninguna. En este problema hemos usado el generador automático de pruebas que cumpla los siguientes requisitos: una o más reservas se pueden asignar a habitaciones, pero una de las reservas no se puede asignar. El problema que se ha generado y cumple todas es el siguiente:

```
View
(:init
    (= (capacidadH H0) 1)

    (= (capacidadH H1) 4)


    (= (diaI R0) 16)
    (= (diaF R0) 20)
    (= (capacidadR R0) 1)

    (= (diaI R1) 21)
    (= (diaF R1) 30)
    (= (capacidadR R1) 2)

    (= (diaI R2) 25)
    (= (diaF R2) 26)
    (= (capacidadR R2) 1)

    (= (diaI R3) 21)
    (= (diaF R3) 22)
    (= (capacidadR R3) 2)


    (libreR R0)
    (libreR R1)
    (libreR R2)
    (libreR R3)
)
```

```
problem proven unsolvable.
```

Imagen 8: Fichero de problemas y resultado de la prueba 3 del nivel Básico

En este caso podemos fijarnos en los días o en las capacidades, pero en cualquier caso queda claro que hay algunas reservas que no se pueden asignar, y por ello no asigna ninguna. Por ejemplo, las reservas R1 y R3 se solapan en días y como las dos tienen capacidad 2, no se pueden asignar en la habitación H0, pero entonces solo una podrá ser asignada a H1 y por eso el programa no asigna ninguna.

4.2 Extensión 1

Prueba 1:

En esta prueba comprobaremos que las asignaciones de reservas no se verán afectadas por aquellas que no se pueden asignar, caso que en la implementación básica si una no se podía asignar, el resto tampoco. El ejemplo más sencillo es aquel en el que una de las reservas que se quiere hacer no se puede dar debido al tamaño de las habitaciones o por solapamiento. Para simplificarlo usaremos nuestro generador para crear una habitación disponible con capacidad de 3, una reserva con la misma capacidad de la habitación para que se pueda asignar, otra reserva de capacidad mayor (4) la cual no se podrá asignar por su tamaño, y otra reserva que se solape en el tiempo con la asignada y por tanto tampoco se pueda asignar. De este modo podremos probar que, aunque más de una reserva no se pueda asignar, el resto sí que se asignan según sus restricciones.

```
View
(:init
    (= (capacidadH H0) 3)


    (= (diaI R0) 23)
    (= (diaF R0) 27)
    (= (capacidadR R0) 3)

    (= (diaI R1) 2)
    (= (diaF R1) 24)
    (= (capacidadR R1) 3)

    (= (diaI R2) 12)
    (= (diaF R2) 22)
    (= (capacidadR R2) 4)


    (libreR R0)
    (libreR R1)
    (libreR R2)
    (= (penalizacion) 0)

)
```

```
step    0: DESCARTAR_RESERVA R2
        1: ASIGNAR_RESERVA H0 R1
        2: DESCARTAR_RESERVA R0
```

Imagen 9: Fichero de problemas y resultado de la prueba 1 de la Extensión 1

En este caso, como explicado antes solamente se podría asignar una de las reservas debido a las distintas restricciones como la capacidad o el solapamiento del tiempo y aun así aquellas reservas que sí que se pueden asignar se asignan adecuadamente. Las otras dos se descartarán, como se ve en la Imagen 9.

Prueba 2:

Una vez comprobado que todas aquellas reservas que se puedan asignar se asignan y aquellas que no se pueden asignar, se descartan, tenemos que comprobar que realmente se optimice el espacio. No queremos que nuestro programa asigne una reserva que forzaría a descartar otras, en vez de descartar esa y que se pudiesen asignar las dos que esa descartaba. En este caso crearemos nosotros la prueba forzosamente para comprobar que se optimiza adecuadamente. Crearemos solamente una habitación con capacidad máxima y tres reservas las cuales una de ellas se solapará con otras dos. De este modo podremos comprobar que, si nuestro sistema realmente optimiza, debería descartar esa que se solapa con las dos y así poder asignar dos en vez de una reserva.

```
(:objects H1 - habitacion
          R1 R2 R3 - reserva
)

View
(:init

    (= (capacidadH H1) 4)

    (= (capacidadR R1) 4)
    (= (capacidadR R2) 4)
    (= (capacidadR R3) 4)

    (= (diaI R1) 2)
    (= (diaF R1) 18)

    (= (diaI R2) 4)
    (= (diaF R2) 6)

    (= (diaI R3) 8)
    (= (diaF R3) 10)

    (libreR R1)
    (libreR R2)
    (libreR R3)

    (= (penalizacion) 0)
)
```

```
step    0: DESCARTAR_RESERVA R1
        1: ASIGNAR_RESERVA H1 R2
        2: ASIGNAR_RESERVA H1 R3
```

Imagen 10: Fichero de problemas y resultado de la prueba 2 de la Extensión 1

En este caso hemos decidido que la primera reserva fuera la que se solape con las demás y no la segunda o tercera para asegurar que, aunque el programa asigne la primera reserva a una habitación, el orden no alterara la solución. En este caso la solución es la esperada descartando la primera reserva que impide que las otras dos se asignen.

4.3 Extensión 2

Prueba 1:

En la primera prueba queremos comprobar que la prioridad de asignar una reserva a una habitación que no tiene la orientación preferente es mayor a no asignar esa reserva. Como ejemplo usaremos el generador automático generando dos habitaciones en orientaciones diferentes y dos reservas en la que una de ellas sea la de una habitación de las dos. Queremos verificar que la reserva con orientación de una de las habitaciones se asignará correctamente y la segunda reserva, aunque la habitación no esté en la orientación preferente se asignará igualmente.

```
(:objects H1 H2 - habitacion
          R1 R2 - reserva
)
    no commands
    (:init

    (= (capacidadH H1) 4)
    (= (orientacionH H1) 0)

    (= (capacidadH H2) 4)
    (= (orientacionH H2) 3)

    (= (capacidadR R1) 4)
    (= (diaI R1) 7)
    (= (diaF R1) 14)
    (= (orientacionR R1) 1)

    (= (capacidadR R2) 4)
    (= (diaI R2) 10)
    (= (diaF R2) 23)
    (= (orientacionR R2) 0)

    (libreR R1)
    (libreR R2)

    (= (penalizacion) 0)
)
```

```
step    0: ASIGNAR_RESERVA_CON_ORIENTACION H1 R2
        1: ASIGNAR_RESERVA_SIN_ORIENTACION H2 R1
```

Imagen 11: Fichero de problemas y resultado de la prueba 1 de la Extensión 2

Como resultado, deberíamos asignar la reserva con la misma orientación que una de las habitaciones y la otra se debería asignar, aunque no sea la preferente pero debido a que es más importante a no asignarla.

Confirmando que nuestro programa asignará las peticiones si cumples las restricciones y que, aunque no haya una habitación con la misma orientación preferente, se asignará si se puede.

Prueba 2:

La segunda prueba nos servirá para identificar si realmente el programa prioriza aquellas reservas con orientaciones similares a las de las habitaciones preferentemente a aquellas que no contienen la misma orientación. Para demostrarlo, crearemos nosotros un ejemplo en el que todas las asignaciones se puedan asignar menos una. En el caso de que todas se puedan asignar a habitaciones con las mismas orientaciones preferentes y una no tenga la misma orientación, el programa debería no asignar aquella antes que el resto. Nuestro ejemplo se compondrá de tres habitaciones, dos orientadas al norte y otra al este, cuatro peticiones todos los mismos días y que todas quepan en las habitaciones, pero una que no tenga como preferencia la orientación ni de norte ni de este.

```
(:objects H1 H2 H3 - habitacion
          R1 R2 R3 R4 - reserva
)
no commands
(:init

    (= (capacidadH H1) 4)
    (= (orientacionH H1) 0)

    (= (capacidadH H2) 4)
    (= (orientacionH H2) 1)

    (= (capacidadH H3) 4)
    (= (orientacionH H3) 0)

    (= (capacidadR R1) 4)
    (= (diaI R1) 2)
    (= (diaF R1) 4)
    (= (orientacionR R1) 2)

    (= (capacidadR R2) 4)
    (= (diaI R2) 2)
    (= (diaF R2) 4)
    (= (orientacionR R2) 0)

    (= (capacidadR R3) 4)
    (= (diaI R3) 2)
    (= (diaF R3) 4)
    (= (orientacionR R3) 1)

    (= (capacidadR R4) 4)
    (= (diaI R4) 2)
    (= (diaF R4) 4)
    (= (orientacionR R4) 0)

    (libreR R1)
    (libreR R2)
    (libreR R3)
    (libreR R4)

    (= (penalizacion) 0)
)
```

Como podemos ver con la solución, las reservas se asignan correctamente respecto la orientación preferente, y además la reserva que se descarta es aquella que no aparece en ninguna habitación. Demostrando que en ningún caso una reserva con diferente orientación a la habitación nunca será preferente respecto a una que tenga la misma orientación.

```
step    0: ASIGNAR_RESERVA_CON_ORIENTACION H1 R2
        1: ASIGNAR_RESERVA_CON_ORIENTACION H2 R3
        2: DESCARTAR_RESERVA R1
        3: ASIGNAR_RESERVA_CON_ORIENTACION H3 R4
```

Imagen 12: Fichero de problemas y resultado de la prueba 2 de la Extensión 2

Prueba 3:

Para la tercera prueba, como ya hemos comprobado los casos límite, simplemente comprobaremos que el programa haga lo que tiene que hacer al recibir una entrada aleatoria. Para ello, utilizaremos el generador de problemas aleatorio. En la imagen 13 tenemos tanto el fichero de problemas como la traza de la ejecución.

```
View
(:init
    (= (capacidadH H0) 1)
    (= (orientacionH H0) 0)

    (= (capacidadH H1) 4)
    (= (orientacionH H1) 2)


    (= (diaI R0) 16)
    (= (diaF R0) 20)
    (= (capacidadR R0) 1)
    (= (orientacionR R0) 2)

    (= (diaI R1) 21)
    (= (diaF R1) 30)
    (= (capacidadR R1) 2)
    (= (orientacionR R1) 2)

    (= (diaI R2) 25)
    (= (diaF R2) 26)
    (= (capacidadR R2) 1)
    (= (orientacionR R2) 2)
```

```
step    0: ASIGNAR_RESERVA_CON_ORIENTACION H1 R0
        1: ASIGNAR_RESERVA_CON_ORIENTACION H1 R1
        2: ASIGNAR_RESERVA_SIN_ORIENTACION H0 R2
```

Imagen 13: Fichero de problemas y resultado de la prueba 3 de la Extensión 2

En este caso, como hay dos habitaciones, las reservas se pueden repartir entre todas, pues solo hay solapamiento entre R1 y R2, pero como estos dos se pueden asignar a ambas habitaciones, con poner uno de ellos en alguna de las dos habitaciones ya basta, para que las otras dos reservas puedan ir en la otra habitación. Como también queremos maximizar el número de reservas con la orientación correcta, asigna R1 y R0 a H1 y R2 a H0. Como todos tiene la misma orientación, da igual si R1 o R2 va a H0 pues el resultado dará la misma penalización.

4.4 Extensión 3

Prueba 1:

En la primera prueba demostraremos que el programa minimiza el número de plazas desperdiciadas. Para ello usaremos el generador de problemas con el objetivo de que reservas se solapen en el tiempo y se tenga que elegir entre ellas dependiendo de las plazas que se desperdicien (suponiendo que las reservas tienen diferentes capacidades). Para ello generaremos solo una habitación y unas cuantas reservas para que alguna de ellas se solape y tengan diferente capacidad.

```
(:objects H1 - habitacion
          R1 R2 R3 R4 - reserva
)
View
(:init

    (= (capacidadH H1) 3)


    (= (capacidadR R1) 2)
    (= (diaI R1) 3)
    (= (diaF R1) 14)

    (= (capacidadR R2) 3)
    (= (diaI R2) 12)
    (= (diaF R2) 17)

    (= (capacidadR R3) 1)
    (= (diaI R3) 8)
    (= (diaF R3) 23)

    (= (capacidadR R4) 4)
    (= (diaI R4) 24)
    (= (diaF R4) 29)

    (libreR R1)
    (libreR R2)
    (libreR R3)
    (libreR R4)

    (= (penalizacion) 0)
)
```

```
step    0: DESCARTAR_RESERVA R4
        1: DESCARTAR_RESERVA R3
        2: DESCARTAR_RESERVA R1
        3: ASIGNAR_RESERVA H1 R2
```

Imagen 14: Fichero de problemas y resultado de la prueba 1 de la Extensión 3

En este caso, se solapan todas las reservas y el programa tiene que decidir cuál de ellas desperdicia menos espacio. Como cada una de las reservas contiene diferentes capacidades, como podemos ver en la solución, entre las diferentes reservas que se podían asignar en la única habitación, se ha elegido la que menos espacio desperdicia entre todas ellas. Debido a que la capacidad de la habitación era de tres y no de cuatro, se ha asignado esa reserva con tres plazas y el resto se han descartado. Así demostramos como mantiene prioridad aquellas reservas que se minimizan el desperdicio de espacio. En la imagen 14 tenemos la inicialización de los parámetros y el resultado de la ejecución.

Prueba 2:

En este problema demostraremos que la optimización respecto el número de reservas asignadas sigue vigente independientemente de la prioridad que mantenga el minimizar el desperdicio de plazas. Este problema lo generaremos nosotros con números elegidos a priori para demostrar que efectivamente se elige un conjunto de reservas óptimo para la solución final contenga el mínimo desperdicio de plazas posibles. Como ejemplo pondremos una habitación con cuatro de capacidad y tres reservas, la primera de capacidad 3 que se solapara con la segunda de capacidad 4 y la tercera que se solapara con la segunda y de capacidad 2.

```
(:objects H1 - habitacion
          R1 R2 R3 - reserva
)
View
(:init

    (= (capacidadH H1) 4)


    (= (capacidadR R1) 3)
    (= (diaI R1) 10)
    (= (diaF R1) 16)

    (= (capacidadR R2) 4)
    (= (diaI R2) 15)
    (= (diaF R2) 24)

    (= (capacidadR R3) 2)
    (= (diaI R3) 20)
    (= (diaF R3) 30)

    (libreR R1)
    (libreR R2)
    (libreR R3)

    (= (penalizacion) 0)
)
```

```
step    0: ASIGNAR_RESERVA H1 R3
        1: DESCARTAR_RESERVA R2
        2: ASIGNAR_RESERVA H1 R1
```

Imagen 15: Fichero de problemas y resultado de la prueba 2 de la Extensión 3

Aunque la segunda reserva tenga capacidad de cuatro personas no tendría sentido asignarla ya que solapa las otras dos que en total podrían asignar a cinco en vez de asignar cuatro y desperdiciar cinco. Como debería ser, el programa asigna las dos reservas para desperdiciar el mínimo posible de plazas correctamente.

4.5 Extensión 4

Prueba 1:

En esta prueba lo que miraremos es que se cumple la optimización descrita en el problema, de que se asignen las reservas si es posible a las habitaciones las cuales tengan un número de asignaciones mayor. Es decir, que se minimice el número de habitaciones abiertas cada mes. Con esa idea en la cabeza hemos creado un problema el cual delimite unas habitaciones dónde, la primera que reciba una reserva tendrá más prioridad a la hora de darle otras reservas que cumplan con las condiciones de esa habitación. De esta manera, veremos si realmente el problema minimiza el número de habitaciones abiertas o no. En este problema hemos creado 4 habitaciones y 5 reservas para poder probar esta problemática.

```
(:init

    (= (capacidadH H1) 2)
    (= (numAssig H1) 0)

    (= (capacidadH H2) 3)
    (= (numAssig H2) 0)

    (= (capacidadH H3) 4)
    (= (numAssig H3) 0)

    (= (capacidadH H4) 2)
    (= (numAssig H4) 0)

    (= (capacidadR R1) 2)
    (= (diaI R1) 3)
    (= (diaF R1) 14)

    (= (capacidadR R2) 3)
    (= (diaI R2) 12)
    (= (diaF R2) 17)

    (= (capacidadR R3) 2)
    (= (diaI R3) 1)
    (= (diaF R3) 2)

    (= (capacidadR R4) 4)
    (= (diaI R4) 5)
    (= (diaF R4) 29)

    (= (capacidadR R5) 1)
    (= (diaI R5) 17)
    (= (diaF R5) 23)

    (libreR R1)
    (libreR R2)
    (libreR R3)
    (libreR R4)
    (libreR R5)

    (= (penalizacion) 0)

)
```

```
step    0: ASIGNAR_RESERVA H4 R5
        1: ASIGNAR_RESERVA H3 R4
        2: ASIGNAR_RESERVA H4 R3
        3: ASIGNAR_RESERVA H2 R2
        4: ASIGNAR_RESERVA H4 R1
```

Imagen 16: Fichero de problemas y resultado de la prueba 1 de la Extensión 4

En este problema se espera que la habitación número 4 se lleve la mayoría de las reservas por el nuevo peso de las asignaciones ya dadas en reservas anteriores.

Como se puede ver en la Imagen 14, la habitación 1 no se abre pues con las otras ya se pueden asignar todas las reservas. Como hay conflictos de días y de capacidades, por fuerza tendremos que abrir más de una habitación pues, por ejemplo, H4 no puede albergar a R4 por insuficiencia de plazas.

Prueba 2:

En esta segunda prueba hemos modificado el problema eliminando una de las habitaciones, para que la competición entre reservas sea más dura y exigente para el planificador. El objetivo es que las habitaciones con capacidad 2, en este caso la H1 y la H4, compitan por el mayor número de reservas asignadas. Las reservas las hemos mantenido igual que la prueba anterior.

```
(:init

    (= (capacidadH H1) 2)
    (= (numAssig H1) 0)

    (= (capacidadH H2) 3)
    (= (numAssig H2) 0)

    (= (capacidadH H4) 2)
    (= (numAssig H4) 0)


    (= (capacidadR R1) 2)
    (= (diaI R1) 10)
    (= (diaF R1) 14)

    (= (capacidadR R2) 3)
    (= (diaI R2) 12)
    (= (diaF R2) 17)

    (= (capacidadR R3) 2)
    (= (diaI R3) 16)
    (= (diaF R3) 18)

    (= (capacidadR R4) 2)
    (= (diaI R4) 5)
    (= (diaF R4) 29)

    (= (capacidadR R5) 1)
    (= (diaI R5) 17)
    (= (diaF R5) 23)

    (libreR R1)
    (libreR R2)
    (libreR R3)
    (libreR R4)
    (libreR R5)

    (= (penalizacion) 0)

)
```

```
step    0: ASIGNAR_RESERVA H4 R5
        1: ASIGNAR_RESERVA H2 R4
        2: ASIGNAR_RESERVA H1 R3
        3: DESCARTAR_RESERVA R2
        4: ASIGNAR_RESERVA H4 R1
```

Imagen 17: Fichero de problemas y resultado de la prueba 2 de la Extensión 4

En este caso, vemos que no se puede dejar ninguna habitación cerrada por conflictos de tiempo y de capacidad. Aún así, el planificador hacer correctamente el plan, pues es peor descartar una reserva que abrir una nueva habitación. Por esta razón se abren todas las habitaciones y solo hay una reserva que no se puede asignar por problemas de incompatibilidad. En la Imagen 15 podemos ver la inicialización de las variables y el resultado obtenido.

Prueba 3:

En esta prueba queremos asegurarnos de que, para un problema generado aleatoriamente con el generador, el programa sigue haciendo lo que toca. En este caso hemos generado dos habitaciones y tres reservas.

```
(:init
    (= (capacidadH H0) 1)
    (= (numAssig H0) 0)

    (= (capacidadH H1) 4)
    (= (numAssig H1) 0)


    (= (diaI R0) 16)
    (= (diaF R0) 20)
    (= (capacidadR R0) 1)

    (= (diaI R1) 20)
    (= (diaF R1) 30)
    (= (capacidadR R1) 2)

    (= (diaI R2) 25)
    (= (diaF R2) 26)
    (= (capacidadR R2) 1)


    (libreR R0)
    (libreR R1)
    (libreR R2)
    (= (penalizacion) 0)

)
```

```
step    0: ASIGNAR_RESERVA H1 R2
        1: DESCARTAR_RESERVA R1
        2: ASIGNAR_RESERVA H1 R0
```

Imagen 18: Fichero de problemas y resultado de la prueba 3 de la Extensión 4

En la imagen 18 podemos ver que el planificador asigna todas las reservas menos R1, pues por conflictos de días con R0 y R2 no hay ninguna combinación posible en la que se pueda asignar R1 con alguna de las dos otras o las tres. Como no queremos abrir más de una habitación, el planificador correctamente asigna R2 y R0 a la misma habitación para no abrir de nuevas.

5. Estudio del tiempo de resolución del problema

Una parte opcional de la práctica era estudiar cómo evoluciona el tiempo de ejecución del planificador al aumentar el tamaño del problema. Para la experimentación, usaremos el generador aleatorio de problemas que hemos programado, que se puede ver en el fichero adjunto llamado "*generador-problemas.cpp*".

Como queremos hacer un estudio adecuado, fijaremos como dominio el nivel básico y generamos sus problemas adecuadamente. Haremos 3 experimentos diferentes para observar y comparar las tendencias:

- Fijaremos el número de habitaciones e iremos aumentando el número de reserves
- Fijaremos el número de reserves e iremos aumentando el número de habitaciones
- Aumentaremos el número de habitaciones y reservas de manera proporcional

Hemos de destacar que, al ser el generador aleatorio, es muy probable que algunos casos tarden más en ejecutarse que otros porque, por ejemplo, si todas las reservas se pueden asignar sin ningún problema, tardará menos que si hay muchos conflictos, pues de la primera manera no tendrá que probar todas las combinaciones.

5.1 Experimento 1

Observaciones: el número de reservas influye en el tiempo de ejecución del programa

Planteamiento: ejecutamos el planeador con diferentes problemas y analizamos el tiempo de ejecución

Hipótesis: el tiempo de ejecución del problema crece de manera exponencial respecto el tamaño (H_o).

Método:

- Fijaremos el número de habitaciones a 10
- Empezaremos con 1 reserva e iremos aumentando de uno en uno hasta llegar a 10
- Crearemos el fichero de problemas con el generador
- Ejecutaremos el planificador con el fichero de domino de la extensión 1 y con el fichero de problemas generado por el generador
- Anotaremos el tiempo de ejecución y haremos un análisis de los resultados.

Resultados:

En la tabla 1 podemos ver el tiempo de ejecución del planificador con los diferentes valores para el número de reservas y de habitaciones. Aunque queda muy claro que el crecimiento del tiempo de ejecución respecto el número de reservas es exponencial, en la figura 1 tenemos una gráfica con los datos de la tabla. Nos habría gustado hasta 10 reservas, pero como el tiempo de ejecución era muy alto, lo hemos dejado en 9, aunque así ya se ve la tendencia de crecimiento.

Tabla 1: Tiempo de ejecución para cada par de valores para las reservas y las habitaciones

Número de Habitaciones	Número de Reservas	Tiempo de ejecución
10	1	0.00 s
10	2	0.00 s
10	3	0.00 s
10	4	0.00 s
10	5	0.01 s
10	6	0.02 s
10	7	0.23 s
10	8	23.34 s
10	9	5,7 min

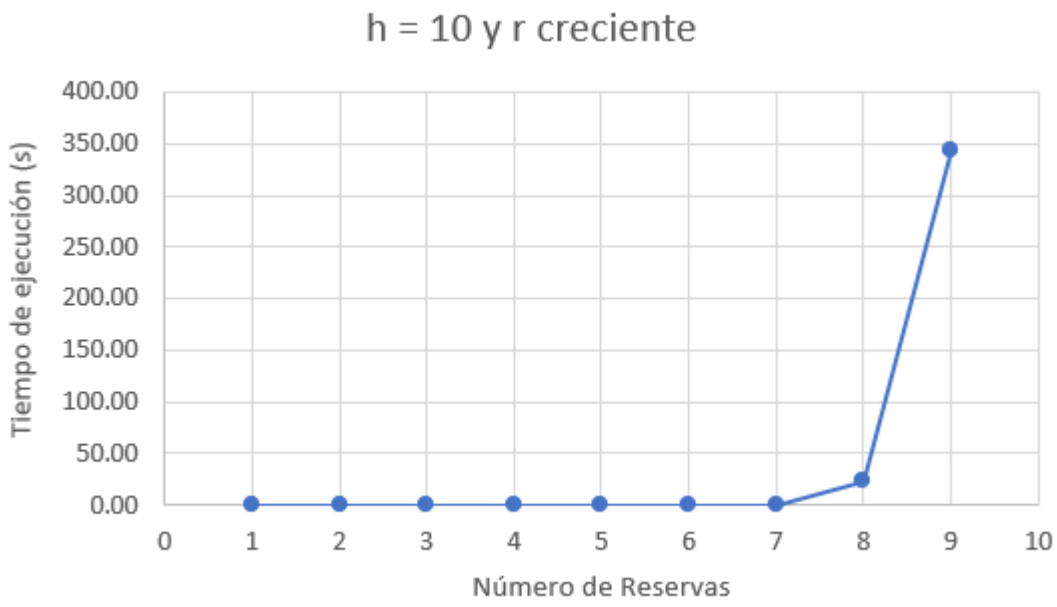


Figura 1: Comparativa del tiempo de ejecución para el primer experimento

Conclusiones:

De esta figura podemos extraer que la hipótesis que habíamos hecha es correcta, pues el tiempo de ejecución crece muy rápidamente. En este caso vemos que, fijando el número de habitaciones, el tiempo de ejecución se mantiene hasta que creamos 8 reservas, cuando ya se dispara un poco. A partir de 9, tenemos un crecimiento exponencial del tiempo de ejecución, y para más reservas no lo hemos probado pues tardaba demasiado.

5.2 Experimento 2

Observaciones: el número de habitaciones influye en el tiempo de ejecución del programa

Planteamiento: ejecutamos el planeador con diferentes problemas y analizamos el tiempo de ejecución

Hipótesis: el tiempo de ejecución del problema crece de manera exponencial respecto el tamaño (H_o).

Método:

- Fijaremos el número de reservas a 10
- Empezaremos con 1 habitación e iremos aumentando de uno en uno hasta llegar a 10
- Crearemos el fichero de problemas con el generador
- Ejecutaremos el planificador con el fichero de domino de la extensión 1 y con el fichero de problemas generado por el generador
- Anotaremos el tiempo de ejecución y haremos un análisis de los resultados.

Resultados:

Como antes, en la tabla 2 tenemos el tiempo de ejecución para diferentes números de reservas y habitaciones. En este caso, podemos observar que el tiempo de ejecución es más irregular, y no sigue un patrón más lineal como antes. Aun así, en la figura 2, podemos ver que a partir de 8 reservas el tiempo de ejecución aumenta, aún más que en el experimento anterior. Nos habría gustado llegar hasta diez habitaciones, pero como el tiempo crecía demasiado no hemos podido, y con 8 ya se ve la tendencia.

Tabla 2: Tiempo de ejecución para cada par de valores para las reservas y las habitaciones

Número de Habitaciones	Número de Reservas	Tiempo de ejecución
1	10	0.00 s
2	10	0.62 s
3	10	0.60 s
4	10	15.07 s
5	10	3.54 s
6	10	79.57 s
7	10	0.11 s
8	10	14.17 min

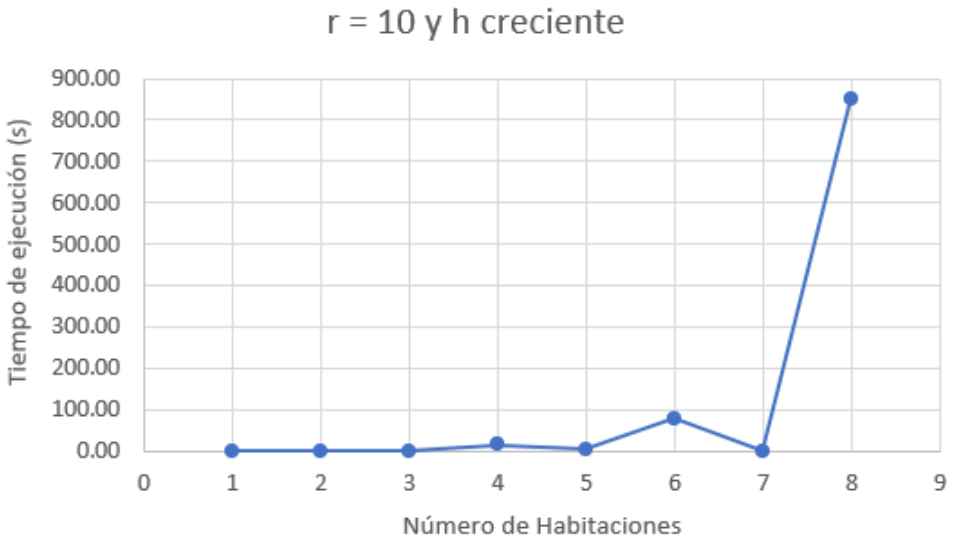


Figura 2: Comparativa del tiempo de ejecución para el primer experimento

Conclusiones:

Como podemos observar, al fijar el número de reservas y aumentar el número de habitaciones, el tiempo de ejecución crece más rápido que cuando fijamos las reservas. Esto diferencia con el experimento anterior se puede deber al hecho que, al haber más reservas que habitaciones, el espacio de exploración es mucho más grande, pues hay más configuraciones de repartición de reservas y menos que sean correctas. Aun así, el problema sigue aumentando de manera exponencial, como habíamos supuesto.

5.3. Experimento 3

Observaciones: el número de reservas y de habitaciones influye en el tiempo de ejecución del programa

Planteamiento: ejecutamos el planeador con diferentes problemas y analizamos el tiempo de ejecución

Hipótesis: el tiempo de ejecución del problema crece de manera exponencial respecto el tamaño (H_o).

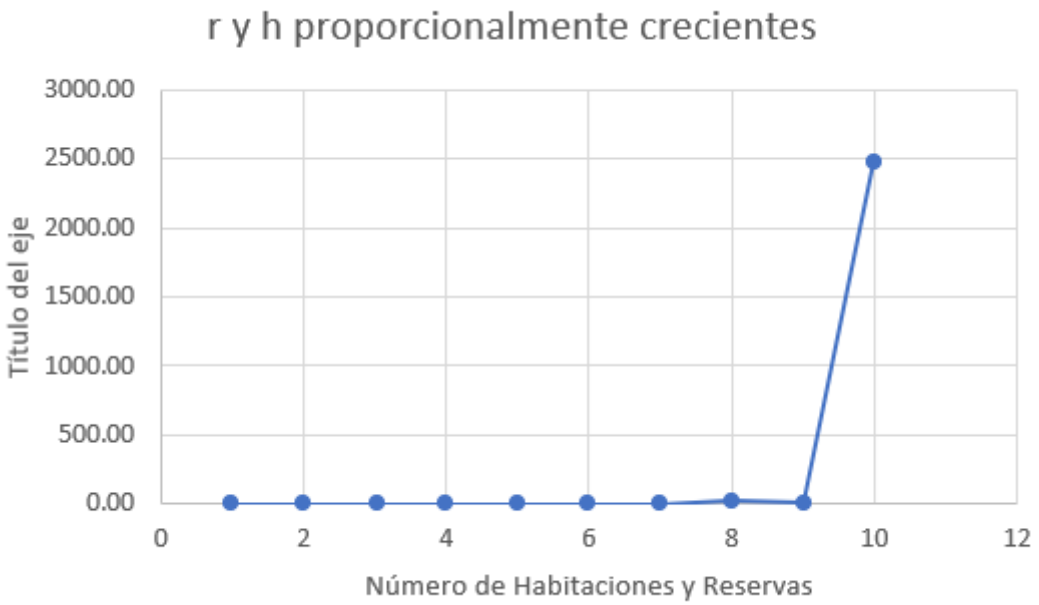
Método:

- Empezaremos con 1 reserva y 1 habitación e iremos aumentando ambos números de uno en uno hasta llegar a 10
- Crearemos el fichero de problemas con el generador
- Ejecutaremos el planificador con el fichero de domino de la extensión 1 y con el fichero de problemas generado por el generador
- Anotaremos el tiempo de ejecución y haremos un análisis de los resultados.

Resultados:

Como en los experimentos previos, en la tabla3 y la figura 3 tenemos los resultados de ejecutar el planificador con valores crecientes e iguales de número de reservas y habitaciones. Para este experimento, hemos obtenido un resultado un poco diferente a los anteriores, aunque no mucho. El tiempo de ejecución no ha sido alto hasta que hemos tenido 8 reservas y 8 habitaciones. A partir de ahí, aunque con valor 9 hay un pico de caída, el tiempo de ejecución crece de manera desmesurada, llegando a los 40 minutos con 10 reservas y habitaciones.

Número de Habitaciones	Número de Reservas	Tiempo de ejecución
1	1	0.00 s
2	2	0.00 s
3	3	0.00 s
4	4	0.00 s
5	5	0.00 s
6	6	0.00 s
7	7	0.00 s
8	8	15.77 s
9	9	0.06 s
10	10	41.32 min



Conclusiones:

En este experimento, hemos visto que el tiempo de ejecución tarda más en crecer, pues al tener el mismo número de habitaciones y reservas, es más fácil que se pueda hacer un emparejamiento de uno a uno, y como el espacio de soluciones no es muy amplio, se puede hacer en menos de 0.00 segundos. Como antes, hemos reafirmado que la hipótesis era correcta, pues el crecimiento del tiempo de ejecución es exponencial.

6. Conclusiones y posibles mejoras

En esta práctica hemos hecho varios diseños para el mismo problema con ciertas variaciones. De esta manera hemos podido ver cómo funciona un planificador y como se programa el domino para que este pueda ejecutar y resolver el problema que se le plantea. Hemos podido observar también, que el diseño que hagamos del domino es muy importante pues puede afectar enormemente en el tiempo de ejecución del planificador.

Nuestro diseño era un poco ineficiente, pues para cada reserva que queríamos asignar, teníamos que comprobar para todas las demás reservas, si había alguna que entraba en conflicto con esta y en ese caso no asignarla a esa habitación. Esto realmente es muy ineficiente con entradas muy grandes, razón por la cual el programa tardaba 47 minutos con 10 reservas y 10 habitaciones.

Con todo, nuestro diseño es mejorable y para un futuro se podría buscar hacer alguna mejora. Aun así, hemos aprendido como se diseña el domino y el problema para que el planificador pueda retornar un resultado.