Práctica: *Planificación automática*
Agentes Inteligentes
Grado en Robótica. Curso 2023-2024.
Departamento de Electrónica y Computación

1. Objetivo

El objetivo de esta práctica es que el alumno aprenda a trabajar con planificación automática y a modelar problemas utilizando lógica de predicados. Durante el desarrollo de esta práctica se empleará el paquete ROSPlan de ROS.

2. Instalación de componentes

Antes de comenzar con la instalación de componentes, se recomienda actualizar los repositorios:

```
sudo apt-get update
```

2.1. Instalación de ROSPlan

Para instalar ROSPlan, seguir los siguientes pasos:

1. Instalar las dependencias requeridas:

   ```
   sudo apt install flex bison freeglut3-dev libbdd-dev python-catkin-tools
   ros-kinetic-tf2-bullet
   ```

2. Para evitar problemas, se recomienda regenerar el *catkin workspace*:

   ```
   rm -rf catkin_ws/src
   mkdir -p catkin_src/src
   cd catkin_ws/
   ```

3. Obtener el código:

   ```
   cd src/
   git clone https://github.com/KCL-Planning/rosplan
   ```

4. Compilar todo:

   ```
   cd ..
   catkin build
   ```

2.2. Instalación del paquete `ai_rosplan`

Al inicio de esta práctica se proporciona un paquete "`ai_rosplan`" con la descripción del escenario. Este paquete se puede encontrar en la carpeta "`Recursos`" de Campus Virtual con el nombre "`ai_rosplan.tar.gz`".:

1. Descomprimir el fichero "`ai_rosplan.tar.gz`" en `~/catkin_ws/src`:

   ```
   tar -xvzf ai_rosplan.tar.gz
   ```

2. Ahora hay que compilar el paquete "`ai_rosplan`", y hacerlo visible a ROS. Para ello, ejecutar la siguiente secuencia de comandos:

   ```
   cd ~/catkin_ws
   catkin_make
   source ~/catkin_ws/devel/setup.bash
   ```

3. El paquete "`ai_rosplan`" está compuesto por dos paquetes: "`ai_domain`" y "`ai_planning`". Para comprobar que ambos son visibles para ROS:

   ```
   rospack find ai_domain
   rospack find ai_planning
   ```

3. Descripción del entorno

En el escenario propuesto tenemos a un robot situado dentro de un *grid*. En la Figura 1 se puede ver una representación gráfica de este escenario donde tenemos este *robot* y un *grid* de tamaño 4×4. El robot puede encontrarse en cualquier celda dentro de este *grid* salvo en aquellas ocupadas por cajas (en la Figura 1 tenemos dos cajas: una verde y otra azul). Cada una de estas celdas está asociada a un *waypoint* que se denominará `wpXY`, donde X se corresponde con la fila, y la Y se corresponde con la columna donde se encuentra. De esta forma, el robot de la imagen se encuentra en la celda o *waypoint* `wp00`, mientras que la caja verde está en el `wp11` y la caja azul en el `wp12`.

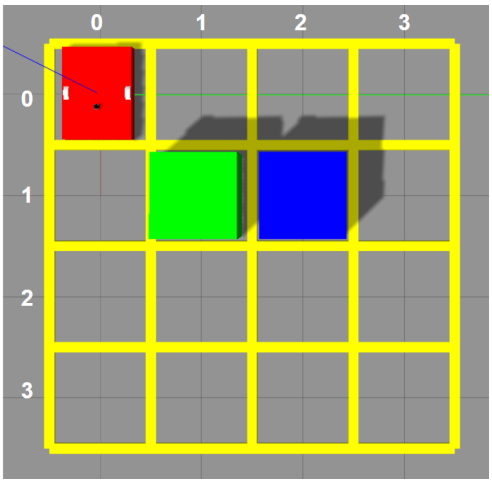

Figura 1: El robot junto con las cajas en el escenario propuesto.

El objetivo es mover el robot para empujar las cajas a los *waypoints* de destino indicados por el usuario. En el siguiente video se muestra un ejemplo, en el que el robot debe situar la caja verde en el *waypoint* `wp21` y la caja azul en el *waypoint* `wp13`. Dado este entorno, se proponen diferentes ejercicios.

4. Ejercicio 1: Moviendo el robot

El objetivo de este ejercicio es construir un dominio `domain1.pddl` que permita mover al robot entre los diferentes *waypoints* que componen el *grid*, pero cumpliendo una serie de restricciones. Para ello, en este dominio habrá que indicar que:

- El robot está posicionado en un determinado *waypoint* `wpXY` dentro del *grid* y con una determinada orientación (donde esta orientación se puede representar, por ejemplo, como `down`, `up`, `left`, y `right`).
- Un determinado *waypoint* está conectado con otro *waypoint* en una determinada orientación (donde como en el caso anterior, esta orientación, puede representarse como `down`, `up`, `left`, y `right`).

Además, dentro de `domain1.pddl` habrá que definir una acción `move` que permite mover el robot de su *waypoint* actual a un *waypoint* destino siempre que estos dos *waypoints* estén conectados y se encuentren en la misma orientación que el robot. Por ejemplo, dado el escenario que se muestra en la Figura 1, el robot solo podría moverse al *waypoint* `wp10`, puesto que es adyacente al *waypoint* `wp00` que es donde está el robot y esta conexión comparte la misma orientación que el robot. Dicho de otra manera, **con este operador el robot solo puede moverse a la celda que se encuentre justo delante del mismo**, y, por tanto, mediante la concatenación de varias acciones `move` podrá moverse en línea recta. Para la definición del dominio, se puede utilizar como plantilla el dominio `domain.pddl` que se encuentra dentro del paquete `ai_domain` en el directorio `common`.

Una vez definido el dominio, hay que crear diferentes problemas. Por ejemplo, el robot inicialmente está posicionado en `wp00` y debe llegar a `wp30`, para lo cual debe recorrer los *waypoints* que estén conectados entre sí que le permitan componer la ruta que une `wp00` con `wp30`[1]. Para construir estos problemas, se puede utilizar como plantilla el fichero `problem.pddl` que se encuentra en el paquete `ai_domain` en el directorio `common`.

Una vez generado el dominio y los problemas, se puede utilizar el *script* `plan.bash` que se encuentra en el paquete `ai_planning` dentro del directorio `scripts`, para obtener un plan que resuelva el problema dado. Para ello, ejecutar [2]:

```
./plan.bash <ruta_dominio>/domain1.pddl <ruta_problema>/problem1.1.pddl
```

que debería generar un plan que mueva el robot de su posición inicio `wp00` a su posición indicada. Continuando con nuestro ejemplo, si asumimos que el robot debe moverse a `wp31` el plan resultante podría ser:

```
0: MOVE KENNY WP00 WP10
1: MOVE KENNY WP10 WP20
2: MOVE KENNY WP20 WP30
```

donde `wp00` está conectado con `wp10`, `wp10` está conectado con `wp20` y así sucesivamente hasta llegar a `wp30`.

Una vez que el dominio y los problemas generan planes correctamente, es hora de ejecutarlos en nuestro escenario mediante ROSPlan. Para ello, copiar el dominio y problemas dentro de la carpeta `common` del paquete `ai_domain` y ejecutar en la carpeta `common`:

```
roslaunch ai_planning main.launch domain_path:=$PWD/domain1.pddl
problem_path:=$PWD/problem1.1.pddl
```
[3]

donde `domain1.pddl` es el dominio `pddl` que acabamos de crear y `problem1.1.pddl` es uno de nuestros problemas. Este comando lanzará los diferentes componentes de ROSPlan, y cargará los hechos iniciales definidos en `problem1.1.pddl` en la base de hechos. En otra terminal, ejecutamos:

[1] En todos estos problemas, el robot debe partir de `wp00` y la notación para definir los *waypoints* debe ser la indicada, es decir, `wpXY` donde la *X* es la fila y *Y* la columna dentro del *grid*

[2] En el caso de que los scripts en el directorio `scripts` no tengan permisos de ejecución: `chmod +x <nombre_fichero>`

[3] La variable de entorno `$PWD` contiene la ruta del directorio actual

```
rosrun ai_planning robot_plan.bash
```

que creará el plan correspondiente a partir de los hechos que se encuentran cargados en la base de hechos en el momento de la ejecución del comando, y comenzará la ejecución del mismo. Si todo es correcto, el robot debería moverse desde su posición inicial a la final definida en el problema.

Para conseguir el movimiento del robot, la acción `move` de nuestro dominio de planificación tiene asociada una interfaz de acción que permite *traducir* nuestra acción de alto nivel `move` a sus correspondientes comandos de bajo nivel interpretables por el robot. Esta interfaz se encuentra definida dentro del fichero `RMoveBase.cpp` que se encuentra en el directorio `src` del paquete `ai_planning`, y en particular dentro del método `concreteCallback`. La clase `RMoveBase.cpp` hereda de `RPActionInterface` que ejecuta el siguiente proceso:

```
IF action name matches THEN:
   check action for malformed parameters
   publish (action enabled)
   success = concreteCallback()
   IF success THEN:
      update knowledge base (effects)
      publish (action achieved)
   ELSE
      publish (action failed)
   RETURN success
```

es decir, si nuestra definición de la función `concreteCallback` se ejecuta correctamente, se asume que la acción se ha llevado a cabo sin problemas, y se actualiza la base de hechos con los predicados recogidos en los efectos de la acción. En caso contrario, no se produce la actualización de estos hechos y, por tanto, es imposible continuar con la ejecución del plan puesto que se incumplen las precondiciones de la siguiente/s acción/es del plan. Por lo tanto, uno de los componentes principales en la arquitectura es esta **base de hechos** o **base de conocimiento**, que recoge en cada momento los predicados instanciados que se corresponden con el estado actual del robot.

5. Ejercicio 2: Girando el robot

Con la acción `move` anterior, el robot es capaz de moverse a la celda adyacente que tiene delante, y mediante la concatenación de estas acciones de movimiento es capaz de moverse en línea recta. En este nuevo ejercicio se va a crear una nueva acción que permita al robot girar a una nueva orientación. Para ello, se creará un nuevo dominio `domain2.pddl`, ampliación del anterior, donde se debe tener en cuenta que:

- El robot tiene en todo momento una determinada orientación (e.g., `down`, `up`, `left`, y `right`).
- El robot puede girar a cualquier otra orientación.

Por lo tanto, en este ejercicio se va a crear un nuevo operador `turn` que reciba como parámetro el robot, la orientación del mismo, y la nueva orientación que debe alcanzar. De esta forma, mediante la combinación de las acciones `move` y `turn` se permite al robot moverse por todo el *grid* y no solo en línea recta. Por ejemplo, se podría mover al *waypoint* `wp31` mediante la concatenación de acciones:

```
0: MOVE KENNY WP00 WP10
1: MOVE KENNY WP10 WP20
2: MOVE KENNY WP20 WP30
3: TURN KENNY DOWN RIGHT
3: MOVE KENNY WP30 WP31
```

Una vez que nuestro nuevo dominio y problemas generan planes correctamente utilizando `plan.bash`, es hora de ejecutarlos con ROSPlan siguiendo los mismos pasos que en el ejercicio anterior. Ahora bien, tenemos una nueva acción en nuestro dominio, `turn`, que hay que traducir del alto nivel al bajo nivel para que el robot pueda ejecutarla correctamente. Para traducir `turn`, hay que modificar el fichero `RPTurn.cpp` dentro de la carpeta `src` del paquete `ai_planning`, e implementar la función `concreteCallback`. Para esta implementación, simplemente haremos girar al robot desde su orientación actual a la de destino. Una vez implementada la función, recompilar el código:

```
catkin build
```

y ejecutar como en el ejercicio anterior.

6. Ejercicio 3: Creando un robot organizador

En este ejercicio se pretende extender la funcionalidad del dominio anterior para permitir que el robot empuje cajas que se encuentran en determinados *waypoints*. Para ello, se creará un nuevo dominio `domain3.pddl` donde se deberá tener en cuenta que:

- Las cajas pueden ocupar determinados *waypoints* en el grid. En la Figura 1 de ejemplo, hay dos cajas, una verde y otra azul, ocupando los *waypoints* `wp11` y `wp12`, respectivamente.
- El robot puede mover estas cajas a *waypoints* adyacentes que se encuentren vacíos.

Además, habrá que definir una nueva acción: `push`, que permite mover una determinada caja. Para ello, recibirá como parámetro el robot, la caja que se desea mover, la localización del robot, la localización de la caja, la nueva localización a la que se desea mover la caja, y la dirección del robot. Con esta información, se debe comprobar que el robot está orientado de tal manera, que podría empujar la caja hasta dejarla en la posición deseada, siempre que esta posición esté vacía. Una vez ejecutada la acción, el robot pasaría a ocupar la posición de la caja, y la caja ocuparía la nueva posición indicada. Por ejemplo, en el escenario de la Figura 1, un robot en el *waypoint* `wp01` y orientado `down`, podría mover la caja roja que ocupa la posición `wp11` hasta `wp21`. Una vez creados el nuevo dominio y problemas relativos a este ejercicio, y se haya comprobado que son capaces de generar planes correctamente utilizando el *script* `plan.bash`, es hora de ejecutarlos con ROSPlan siguiendo los mismos pasos que en el ejercicio anterior, para ver la ejecución de estos planes en el simulador.

7. Ejercicio 4: Añadiendo costes

En este punto, el robot debería ser capaz de encontrar y ejecutar planes que le permitan mover las cajas por el escenario. En este nuevo ejercicio, se propone además que el robot mueva estas cajas de forma eficiente optimizando el consumo de batería. Para ello, se hará uso de predicados numéricos o *fluents*. En el nuevo dominio `domain4.pddl` correspondiente a este nuevo ejercicio, habrá que indicar que:

- El robot dispone de una batería que inicialmente está totalmente cargada.
- Desplazarse entre *waypoints* conectados tiene asociado un consumo de batería que puede ser diferente cada vez, i.e., el consumo de batería de desplazarse de `wp00` a `wp01` puede ser diferente al de desplazarse de `wp01` a `wp02`.
- Mover cada caja a un determinado *waypoint* también tiene asociado un consumo de batería.

Dada esta información, habrá que hacer las modificaciones necesarias a las acciones `move` y `push` para decrementar la batería del robot con los consumos indicados. Además, en los problemas habrá que indicar la métrica que se pretende optimizar, en este caso, maximizar el nivel de batería del robot una vez finalizada la ejecución del plan. Habrá que construir problemas de tal manera que se compruebe que el robot prefiere planes óptimos frente a los que no lo son de acuerdo a la métrica indicada. Seguir los pasos de los ejercicios anteriores tanto para comprobar que el dominio y los problemas generados son correctos utilizando el *script* `plan.bash`,

como para lanzar la ejecución del robot en *gazebo*. Como en este caso se pretende optimizar una métrica, la invocación de estos comandos será:

```
./plan.bash <ruta_dominio>/domain4.pddl <ruta_problema>/problem4.1.pddl opt
```

para invocar al planificador, y:

```
roslaunch ai_planning main.launch domain_path:=$PWD/domain4.pddl
problem_path:=$PWD/problem4.1.pddl optimise:=true
```

para ejecutar el dominio y problema en ROSPlan.

8. Ejercicio 5: Replanificación basado en el sensado

Aunque se ha simplificado en esta práctica, la base de hechos del robot se debería estar actualizando continuamente a partir de la información sensorial. Así, un cambio en el entorno actualizará la base de hechos, lo que podría hacer que el plan falle y el robot tenga que replanificar en busca de un nuevo plan. En este ejercicio vamos a simular uno de estos eventos. Para ello, vamos hacer uso del *script* `updateKB.bash` del directorio `scripts` del paquete `ai_planning`. Abrir el *script* y estudiar su contenido. Su ejecución provoca la eliminación de un predicado de la base de hechos, en particular, elimina la conexión entre dos *waypoints*. Este *script* de ejemplo elimina el predicado `connected` que une `wp00` con `wp01` en la orientación `right`. Habrá que modificar el nombre `connected` con el nombre del predicado que se haya elegido para unir dos *waypoints*, y `wp00`, `wp01` y `right` con el nombre de los *waypoints* cuya conexión se desee eliminar y su orientación. A continuación, **durante** la ejecución del plan en *gazebo*, elimina una conexión que el robot vaya a utilizar más adelante en el plan. ¿Qué es lo que sucede? ¿Cuál es el contenido de la base de hechos en este momento? ¿Qué se te ocurre que podrías hacer para que el robot continúe con la ejecución? Ejecuta el comando necesario para que el robot continúe la ejecución y consiga sus metas en el caso de que sea posible.

9. Ejercicio 6: Planificación multi-agente centralizada

Si se abre el *script* `plan.bash`, se puede comprobar que la variable `PLANNER` tiene como valor `Metric-FF`. Aunque el planificador `Metric-FF` es capaz de optimizar la calidad del plan de acuerdo a las métricas establecidas, no es capaz de razonar con el tiempo. En este ejercicio, vamos a generar planes que involucren a varios robots y que permitan la ejecución de acciones en paralelo. Para ello, vamos a utilizar el planificador temporal `popf` [4]. Se puede tomar como punto de partida el dominio y problemas del ejercicio 3, y generar un nuevo dominio `domain6.pddl`, con acciones durativas. En los nuevos problemas que se generen para este nuevo dominio se deberán considerar varios robots, en lugar de solo uno, y se deberá estudiar como `popf` es capaz de encontrar planes que involucran acciones que se ejecutan en paralelo. En este ejercicio y para este estudio, se usará solo el *script* `plan.bash`, y **NO** se tratará de ejecutar estos planes con múltiples robots en el entorno *gazebo* proporcionado.

10. Ejercicio *Bonus*: Planificación multi-agente centralizada en ROSPlan

El objetivo de este ejercicio es conseguir tener a varios *turtlebots* moviéndose y ejecutando acciones en paralelo en el escenario propuesto. Para ello, se debe hacer uso del dominio y problemas generados en el ejercicio anterior. Además, habrá que hacer algunas modificaciones adicionales:

- En `main.launch` modificar el planificador de `Metric-FF` a `popf`.
- Modificar el plan *dispatcher* de *simple* a *esterel* en `rosplan_full.launch`.
- Modificar el escenario propuesto para que aparezcan más robots en lugar de solo uno.

[4] Modificar el script `plan.bash` para que `PLANNER` tenga el valor de `popf`

- Modificar el método *concreteCallback* de `RPMoveBase.cpp` para permitir situar inicialmente el robot en un punto diferente de `wp00`.

Además de estas modificaciones, se deberá investigar y realizar el resto de modificaciones necesarias para conseguir el objetivo propuesto.

11. Directrices para la Memoria

La memoria debe entregarse en formato .pdf y tener un máximo de 15 hojas en total, incluyendo la portada, contraportada e índice. Al menos, ha de contener:

1. Breve introducción explicando los contenidos del documento.
2. Descripción de los dominios y problemas generados en cada uno de los ejercicios, junto con ejemplos de algunos planes obtenidos, acompañados de las explicaciones correspondientes. Para el ejercicio 5, responder a las preguntas indicadas, y describir las pruebas que se han realizado. Por último, en el caso de realizar el ejercicio de *bonus*, indicar brevemente las modificaciones llevadas a cabo para obtener el funcionamiento requerido.
3. Conclusiones acerca de la práctica.

La memoria **no debe incluir código fuente** en ningún caso.

12. Evaluación

La evaluación de la práctica se realizará sobre 10 puntos siguiendo el siguiente reparto de puntos:

- Ejercicio 1, 2 y 3: 2 puntos
- Ejercicios 4 y 6: 1.5 puntos
- Ejercicio 5: 1 punto

Para obtener la totalidad de la puntuación en cada ejercicio no solo es necesario presentar los dominios y problemas en `.pddl` y que funcionen correctamente de acuerdo a los objetivos de cada ejercicio, sino que será necesario, además, acompañarlos de explicaciones adecuadas en la memoria.

13. Entrega

Se tiene de plazo para entregar la práctica hasta el 17 de Diciembre a las 23:55. Sólo un miembro de cada pareja de estudiantes debe subir a la sección de prácticas de Campus Virutal un único fichero `.zip`. Es importante cumplir las siguientes instrucciones para la entrega.

1. El fichero debe nombrarse `p3-Apellido1-Apellido2.zip`, donde `Apellido1` y `Apellido2` se corresponden con el primer apellido de los integrantes del grupo de trabajo. La descompresión de este fichero debe contener:

   a) La memoria en formato .pdf con nombre `memoria.pdf`

   b) Carpeta "`ai_rosplan`" que se corresponde con el paquete que se utilizará para desarrollar la solución. Al inicio de la práctica se proporciona una primera versión de este paquete, que se utilizará como base para construir los desarrollos requeridos. El paquete final que se entregue debe incluir todo el código desarrollado y, además, debe contener en el directorio `common` de `ai_domain` todos los dominios y problemas generados.

Importante: no seguir las normas de entrega puede suponer una pérdida de hasta 1 punto en la calificación final de la práctica.