Examen
Grado en Robótica. Curso 2022-2023.
Departamento de Electrónica y Computación

Nombre:.....

1. Problema 1 (4 puntos)

Se desea gestionar automáticamente un ascensor inteligente que se encuentra en un edificio con un número arbitrario de plantas n, $n > 2$, y que tiene una capacidad máxima de 10 personas. En cada planta hay usuarios, para cada uno de los cuales se sabe la planta en la que se encuentran, y aquella a la que desean acceder. Una vez que una persona entra en el ascensor, ya no debe salir hasta que haya llegado a su planta de destino. Inicialmente, el ascensor se encuentra en la planta baja, y no importa donde acabe cuando ha atendido todas las peticiones.

Se pide responder razonadamente las siguientes preguntas:

1. (**1 punto**) Representar el problema como un espacio de estados y conjunto de operadores especificando para cada uno de estos últimos sus precondiciones y efectos.
2. (**0.5 puntos**) ¿Cuál es el máximo factor de ramificación?
3. (**0.5 puntos**) Si se desea minimizar el recorrido total que debe hacer el ascensor para atender a todas las personas, ¿qué algoritmo de búsqueda no informada sugerirías para su resolución?
4. (**0.5 puntos**) Sabiendo que el ascensor tarda sólo 3 segundos en llegar desde una planta a otra inmediatamente adyacente y, por otra parte, la carga y descarga de personas dura siempre 30 segundos, ¿qué algoritmo de búsqueda no informada sugerirías para minimizar el tiempo total de operación para atender todas las peticiones?
5. (**1 punto**) Sugiere una función heurística *h()* que sea admisible e informada para el problema de minimizar la distancia total recorrida por el ascensor para atender todas las peticiones.
6. (**0.5 puntos**) Suponiendo que se dispone de una función heurística *h()* admisible, sugiere un algoritmo de búsqueda heurística que sirva para resolver óptimamente el problema.

2. Problema 2 (2 puntos)

Recorrer el grafo de la Figura 1 utilizando el algoritmo IDA* teniendo en cuenta los costes indicados en cada arco. En todos los casos, indicar en qué orden se visitan los nodos, distinguiendo nodos generados de nodos expandidos. Para cada nodo indicar, además, su valor correspondiente a la función de evaluación, la función de coste y su valor heurístico. Tomar como estado inicial el nodo S y como único estado meta el nodo G. Cada nodo del grafo tiene el valor heurístico descrito en la Figura 2.

3. Problema 3 (4 puntos)

Un robot "aprieta tuercas" se encarga de apretar las tuercas en una fábrica. Para ello, tiene a su disposición una serie de llaves. Además, las llaves son muy frágiles, por lo que una vez que se usan para apretar una tuerca se rompen y luego no se pueden usar para apretar otra. Las llaves, tuercas y el robot mismo se encuentran

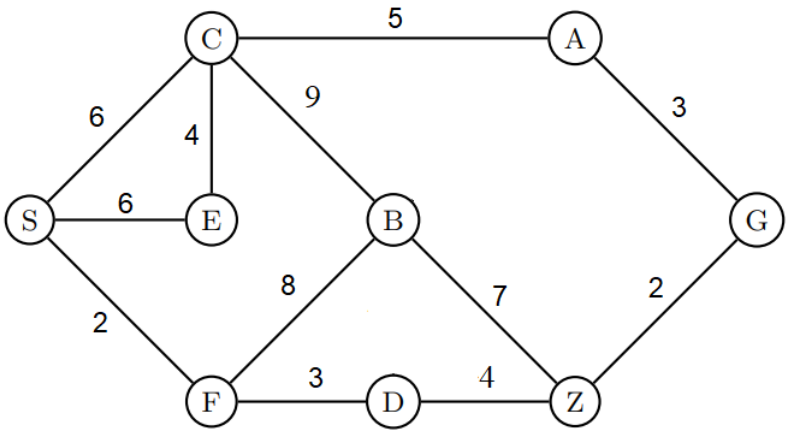

Figura 1: Representación del grafo de búsqueda.

n	S	A	B	C	D	E	F	G	Z
h(n)	7	2	5	6	3	7	7	0	1

Figura 2: Valores heurísticos.

distribuidos por diferentes localizaciones de la fábrica, y el objetivo del robot es navegar por la fábrica, recoger las llaves, y apretar todas las tuercas.

Dado este dominio, se pide resolverlo utilizando *planificación automática*. Para ello:

1. (1.5 puntos) Empleando la **lógica de predicados**, indicar qué predicados sirven para describir cada estado.

2. (1 punto) Pon un ejemplo de **estado inicial** y **final**.

3. (1.5 puntos) Describir brevemente e **informalmente** los operadores que utilizarías para solucionar el problema y modelar uno de ellos **formalmente** en PDDL.

4. Solución Problema 1

1. El espacio de estados es una formalización que habilita la aplicación de algoritmos de búsqueda (informados o no) para la resolución de problemas. Consiste únicamente, en la definición de estados y operadores que sirven para transitar entre ellos. Relacionando, después, el conjunto de posibles estados con otro de vértices V y el de operadores convenientemente instanciados) con otro de arcos E, resulta entonces de forma natural la definición de un grafo, el grafo de búsqueda que se recorrerá eficientemente con el uso de árboles de búsqueda. Este ejercicio propone ejercitar todos estos conceptos y, en el primer apartado, el del espacio de estados

   - **Estados.** En este problema un estado se representa con información del estado del ascensor, los usuarios y todas las peticiones pendientes de ser atendidas. A continuación se presentan definiciones estructuradas para cada uno de estos conceptos:
     - **Usuario.** Cada usuario debe identificarse de forma única con algún campo como su nombre o un código que le haga indistinguible de los demás, `usuario.id`. Además, tal y como advertía el enunciado, para cada uno se conoce la planta en la que se encuentra inicialmente (`usuario.from`), y aquella a la que desea llegar, `usuario.to`. Obviamente, todos los campos indicados aquí son númericos —con la salvedad del primero, `usuario.id` que, tal vez, podría representarse con otros tipos diferentes. Como eventualmente los usuarios irán entrando en el ascensor, puede parecer que debe crearse un campo para mantener esta información. En su lugar, se sugiere que sea el ascensor el que sepa quiénes están dentro de él.
     - **Ascensor.** El ascensor tiene una capacidad máxima (`ascensor.C`) (que, en este caso, es igual a 10). Además, en cada estado podría encontrarse en una planta diferente, `ascensor.planta` y, podría llevar a un conjunto diferente de usuarios dentro, `ascensor.usuarios`. Los dos primeros campos son necesariamente numéricos. Ahora bien, el último consiste simplemente en un contenedor de instancias de usuario descritas anteriormente.
     - **Peticiones.** Las peticiones consisten, simplemente, en un contendor de instancias de usuario, cada una de las cuales ya mantiene información de la planta en la que se encuentre cada uno, y a la que desean acceder.

     Sea s una instancia de estado representando con todas las estructura indicadas anteriormente que represente el estado inicial. El problema advertía explícitamente que en el estado inicial, el ascensor se encuentra en la primera planta, esto es, $s.ascensor.planta = 0$; asimismo, parece lógico asumir que no contendrá ningún pasajero dentro, de modo que $s.ascensor.usuarios = \emptyset$. Sea t una instancia de estado que represente el estado final. En realidad, la única condición que se tiene que dar para asegurar que se ha alcanzado el estado final es $s.peticiones = \emptyset$ y, además, que no hay pasajeros en el ascensor (porque han sido todos despachados), $s.ascensor.usuarios = \emptyset$.
   - **Operadores.** En la definición de operadores es importante describir sus precondiciones/postcondiciones y, además, el coste que tienen. En total pueden distinguirse tres acciones: una de ellas relacionadas con el movimiento del ascensor, y otras dos relacionadas con las operaciones con usuarios. En cuanto al ascensor, basta con crear una única operacion que desplace el ascensor desde su planta actual hasta cualquier otra: `mover(planta_destino)`. Después de verificar que la planta de destino efectivamente existe, lo único que debe hacer es actualizar la planta en la que acabará el ascensor. Ahora bien, también podría haberse modelado el movimiento del ascensor en pasos de un nivel cada uno, con dos operadores que no reciben parámetros: `subir` y `bajar`.

     En cuanto a los usuarios, estos pueden `entrar` o `salir` del ascensor. Para verificar que efectivamente pueden entrar basta con comprobar que el ascensor y el usuario están en la misma planta y, además, que no se viola la capacidad máxima. el único efecto de entrar en el ascensor es que son añadidos a la lista de usuarios dentro del ascensor. Análogamente, los usuarios pueden salir del ascensor si se encuentran dentro del ascensor y, además, este está en la misma planta que aquella a la que ellos quieran acceder, y tiene dos efectos: deben ser eliminados de la lista de usuarios en el ascensor y, además, de la lista de peticiones por atender, puesto que este cliente ya ha sido satisfecho.

La siguiente tabla muestra todas las acciones descritas anteriormente (donde `mover` es una alternativa al uso de las acciones de `subir` y `bajar`), caracterizando cada una con las precondiciones y postcondiciones necesarias.

Operador	Precondiciones	Efectos
mover(*planta_destino*)	$planta\_destino \geq 0 \wedge planta\_destino \leq n$	$ascensor.planta = planta\_destino$
entrar(*usuario*)	$ascensor.planta = usuario.from \wedge \lvert ascensor.usuarios \rvert < ascensor.C$	insertar usuario en $ascensor.usuarios$
salir(*usuario*)	$usuario \in ascensor.usuarios \wedge ascensor.planta = usuario.to$	eliminar usuario de $ascensor.usuarios$

Además, es importante determinar el coste de cada una de estas acciones. Esto no se ha hecho explícitamente en la tabla anterior porque dependerá de la métrica que se use. De hecho, en otras secciones, el enunciado distingue entre dos:

- **Distancia.** Cuando pretenda minimizarse la distancia total recorrida por el ascensor, el coste de cada acción debe ser, entonces, la distancia que recorre. En nuestra formalización anterior resulta obvio que mover tendrá entonces un coste que será igual al número de pisos que debe recorrer: $planta\_destino - ascensor.planta$. Alternativamente, las acciones de `subir` y `bajar` tienen todas un coste de una unidad, puesto que sólo desplazan el ascensor un piso. Por otra parte, las acciones de carga y descarga de usuarios no puede tener coste, puesto que durante su ejecución el ascensor no se mueve.
- **Tiempo.** En la minimización del tiempo deben entonces aplicarse los mismos costes que se explican en el cuarto apartado. Esto es, el coste de mover el ascensor será 3 veces el número de pisos que se desplaza: $3 \times (planta\_destino - ascensor.planta)$, puesto que tarda 3seg. en desplazarse cada nivel. Sin embargo, las acciones de `subir` y `bajar` tienen un coste siempre igual a 3, puesto que sólo desplazan el ascensor un piso. Por último, todas las operaciones de entrada y salida tendrán un coste de 30 segundos.

2. En este caso, el factor de ramificación máximo será igual al número de acciones que se han definido en el apartado anterior, convenientemente instanciadas. El número de personas que pueden subir en una planta es igual al número de personas que puede haber ahí, y éste número no puede ser mayor que el número de peticiones, $|peticiones|$. Análogamente, el número de personas que pueden bajar de un ascensor será, como máximo, el número de personas que hay dentro del ascensor, que no puede ser mayor que su capacidad (10, en el enunciado del problema). Por lo tanto, las acciones que se refieren a los usuarios pueden instanciarse como máximo de $10 + |peticiones|$ formas diferentes. Por último, si el movimiento del ascensor se modela con las acciones de `subir` y `bajar`, entonces deben añadirse otras dos acciones al cálculo del máximo factor de ramificación. Si, por el contrario, se modela con la acción de mover , entonces este operador podría instanciarse de $(n - 1)$ formas diferentes, con n el número de plantas que tiene el edificio.

   En conclusión, habría hasta $12 + |peticiones|$ descendientes de cada nodo como máximo con el uso de `subir` y `bajar`, o hasta $10 + (n - 1) + |peticiones|$ como factor de ramificación máximo si se emplea, en su lugar, el operador `mover`.

3. Tanto si el movimiento del ascensor se modeliza con las acciones de subir y bajar, como si se hace con la acción de mover, se trata de un problema de costes variables, puesto que las acciones de carga y descarga de personas tienen un coste nulo, distinto a los anteriores. Por ello, el algoritmo del primero en amplitud no encontraría ya soluciones óptimas, puesto que no existe una relación unívoca entre la profundidad a la que se encuentran las soluciones, y su coste. En su lugar, pueden usarse uno cualquiera de los algoritmos de fuerza bruta o no informados estudiados específicamente para el caso de costes variables, por ejemplo:

   - **Dijkstra**. Consiste en un algoritmo de el mejor primero donde la función de evaluación, $f(n)$ es, simplemente, el coste del camino desde el estado inicial hasta n, $g(n)$. Tiene un coste de memoria exponencial pero garantiza que cada vez que expande un nodo habrá encontrado la solución óptima hasta él puesto que los nodos se expanden en orden creciente de su valor de $f(n)$ (o, equivalentemente, de $g(n)$).

4. No cambia absolutamente nada. Una vez más, se trata de acciones con costes diferentes (tanto si se modela el movimiento del ascensor con un grupo de acciones u otro) y, por lo tanto, las recomendaciones serían exactamente las mismas que las del apartado anterior.

5. Una heurística muy sencilla que puede obtenerse con la técnica de relajación de restricciones, consiste en relajar la capacidad del ascensor. Asumiendo ahora que la capacidad del ascensor es infinita, el problema relajado resultante puede resolverse óptimamente considerando la distancia que hay entre las plantas más baja y alta que haya que visitar. Como quiera que el ascensor debe ir a una planta, ya sea a recoger a un usuario, o a dejarle, la planta más baja a la que debe llegar el ascensor es:

   $p^- = \text{mín}_{\forall usuario \in peticiones}\{usuario.from, usuario.to\}$

   La misma consideración debe hacerse, naturalmente, para el cálculo de la planta más alta hasta la que hay que llevar el ascensor:

   $p^+ = \text{máx}_{\forall usuario \in peticiones}\{usuario.from, usuario.to\}$,

   de donde resulta que la diferencia entre estas dos plantas es una estimación admisible (esto es, que no sobreestima el esfuerzo necesario): $h_1 = p^+ - p^-$.

6. La selección de un algoritmo de búsqueda informada para este caso depende, como en el caso de los algoritmos de búsqueda no informada, del número de transposiciones y también, naturalmente, de la dificultad de los problemas:

   - El algoritmo A* es admisible y garantiza, por lo tanto, que encontrará soluciones óptimas si la función heurística que lo guía también es admisible. Además, es un algoritmo rápido puesto que no reexpande nodos (y, con frecuencia, las ordenaciones de la lista abierta se pueden hacer en $O(1)$ con las estructuras de datos adecuadas si la función objetivo sólo toma valores enteros). Sin embargo, tiene un consumo de memoria exponencial.
   - El algoritmo IDA* reexpande nodos en caso de que haya transpociones pero no ordena nodos y, mucho más importante aún, tiene un consumo de memoria lineal en la profundidad de la solución (que en nuestro caso es siempre igual a N). Además, también es un algoritmo de búsqueda admisible.

   Tal y como se indicó anteriormente parece particularmente fácil evitar las transposiciones en este dominio con reglas muy sencillas, de modo que cualquiera de estos algoritmos pueden usarse para encontrar soluciones óptimas al problema planteado.

5. Solución problema 2

IDA*, a diferencia de A*, consiste en una serie de recorridos en profundidad hasta que $f(n) > \eta$ o se ha encontrado la solución, incrementando η en cada iteración al menor exceso cometido. Por lo tanto, además, al contrario que en profundidad iterativa, el valor de η no se incrementa en un valor k en cada iteración, sino al menor exceso cometido como veremos a continuación. Inicialmente, y para garantizar la admisibilidad, η se inicializa con $\eta = h(S) = 7$. Teniendo en cuenta todo esto, procedemos a aplicar el algoritmo al grafo propuesto.

Iteración 1: $\eta = h(S) = 7$

En la Figura 3 se muestra la primera iteración del algoritmo. Se muestran en verde los nodos expandidos, y en gris los nodos generados. En los arcos del árbol se muestra el coste de transitar de un nodo a otro, y al lado de los nodos se muestra entre paréntesis en primer lugar el valor de $h(n)$ seguido por el valor de $f(n)$ de ese nodo en particular. Teniendo todo esto en cuenta, la expansión de S genera tres nodos: C, E y F. Siguiendo nuestro recorrido en profundidad, deberíamos tratar de expandir el nodo C. Sin embargo, $f(C) = 12 > \eta = 7$,

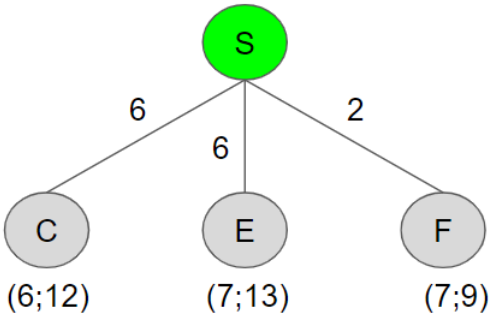

Figura 3: Primera iteración de IDA*.

luego hacemos *backtracking* y continuamos nuestro recorrido en profundidad por el nodo E. En E nos encontramos con una situación similar, $f(E) = 13 > \eta = 7$, así que hacemos nuevamente *backtracking* y continuamos nuestro recorrido en profundidad para llegar a F. $f(F) = 9 > \eta = 7$, luego en esta primera iteración únicamente se expande S. En la siguiente iteración, se elige el valor de η entre el menor de $f(C) = 12$, $f(E) = 13$, $f(F) = 9$, es decir, en la siguiente iteración $\eta = 9$.

Iteración 2: $\eta = 9$

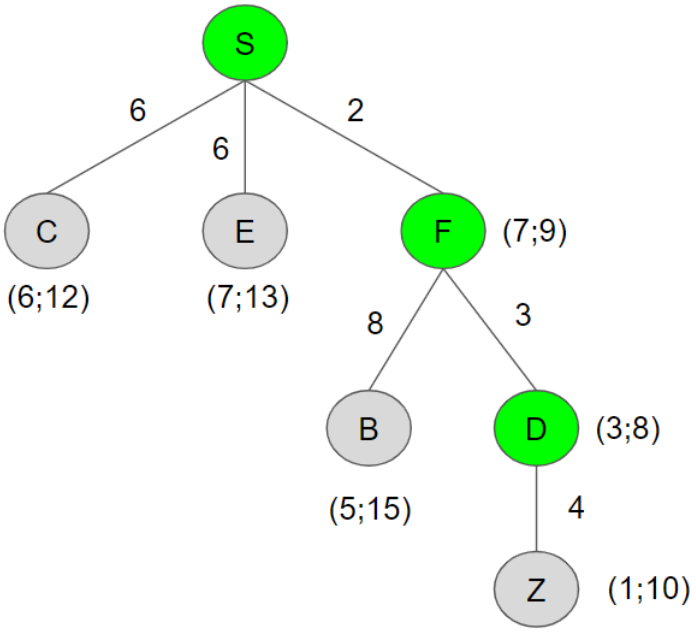

Figura 4: Segunda iteración de IDA*.

En la segunda iteración de IDA* con $\eta = 9$, se generan en primer lugar los mismos nodos generados en la iteración anterior, solo que esta vez sí que podremos expandir el nodo F puesto que $f(F) = 9 = \eta$[1]. La expansión del nodo F genera los nodos B y D. El primero con una función de evaluación $f(B) = 15$ y el segundo con una $f(D) = 8$[2]. Procederíamos en un recorrido en profundidad: tratamos de expandir B, pero no es

[1]Es importante recordar que en cada nueva iteración de IDA*, se comienza con un nuevo proceso de búsqueda en profundidad desde el nodo de inicio.

[2]En el cálculo de las funciones de evaluación de este par de nodos, hay que tener en cuenta el coste de llegar desde el nodo de inicio al nodo en el que nos encontramos. Por lo tanto, $f(B)$ es igual a la suma de llegar del nodo S al nodo F (2), más el coste de llegar del nodo F al nodo B (8), más el valor heurístico del nodo B (5), obteniendo así $f(B) = 15$. De igual forma se procedería con el cálculo de

posible puesto que $f(B) = 15 \geq \eta = 9$. El siguiente nodo que expandimos es D puesto que $f(D) = 8 \leq \eta = 9$. La expansión de D genera el nodo Z, que trataríamos de expandir comprobando que no es posible puesto que $f(Z) = 10 \geq \eta = 9$, cerrando así la segunda iteración de IDA*. Incrementamos η al menor de los excesos cometidos entre las funciones de evaluación de los nodos que aún no han sido expandidos, es decir, $\eta = \text{mín}\{f(C) = 12, f(E) = 13, f(B) = 15, f(Z) = 10\}$. Por lo tanto, en la nueva iteración $\eta = 10$.

Iteración 3: $\eta = 10$

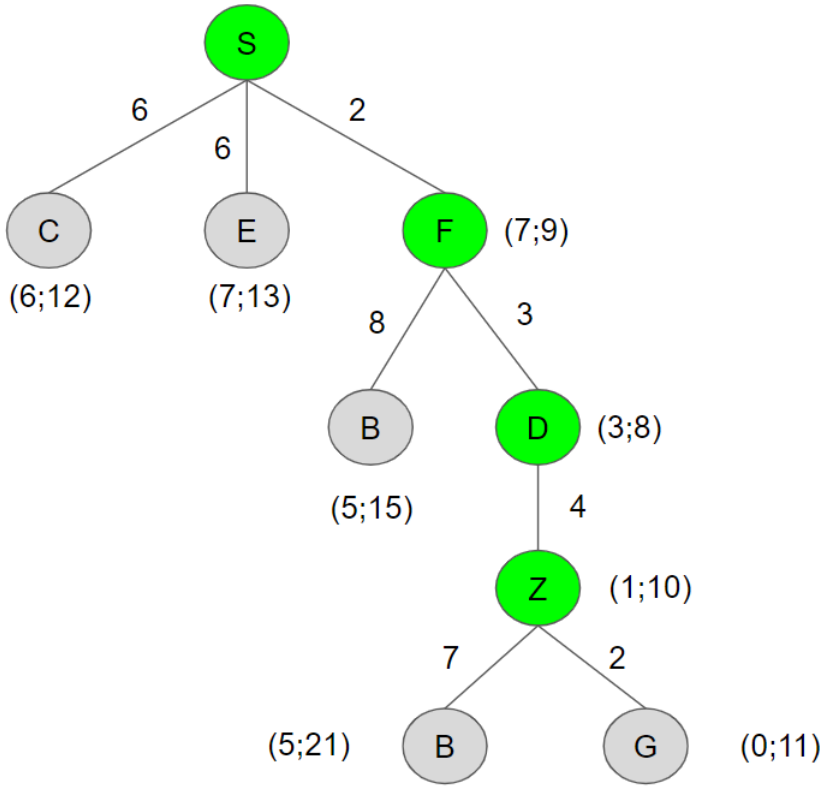

Figura 5: Tercera iteración de IDA*.

La Figura 5 muestra como quedaría la tercera iteración de IDA*. En esta ocasión, cuando lleguemos al nodo al nodo Z sí que podremos expandirlo, puesto que $f(Z) = 10 \leq \eta = 10$. La expansión de este nuevo nodo genera los nodos B con una $f(B) = 21$ y el nodo meta G con una $f(G) = 11$. Hemos encontrado un camino hasta el nodo G de coste 11, pero como $f(G) = 11 \geq \eta = 10$, no podemos garantizar que sea la ruta óptima. Debemos ejecutar necesariamente una nueva iteración incrementando η al menor de los excesos cometidos, es decir, $\eta = 11$.

Iteración 4: $\eta = 11$

En el nuevo árbol generado mediante un recorrido en profundidad del grafo con un valor $\eta = 11$, volvería a generar el mismo árbol que se muestra en la Figura 5, solo que en este caso generando el nodo G con una función de evaluación $f(G) = 11 \leq \eta = 11$, luego podemos garantizar que el camino óptimo para llegar del nodo S al nodo G es $S \rightarrow F \rightarrow D \rightarrow Z \rightarrow G$ con coste 11.

la $f(D)$.

6. Solución problema 3

1. Como ocurre en todos los problemas que nos piden representar los conceptos que aparecen en el enunciado, no tiene por qué existir una única representación válida. En este caso se pide representar el problema utilizando lógica de predicados, y un conjunto válido de predicados podría ser:

   - `(at ?obj ?wp)`: Un determinado elemento `?obj` está en un determinado *waypoint* `?wp`. Este predicado es necesario puesto que el enunciado nos dice que todos los elementos que aparecen, ya sean llaves, tuercas o el propio robot, están en una determinada localización. En este caso, el argumento `?obj` podría tomar el valor de una llave, tuerca o el propio robot.
   - `(connected ?wp1 ?wp2)`: El *waypoint* `?wp1` está conectado al *waypoint* `?wp2`. Conjuntamente con el predicado anterior, podemos construir un grafo donde los vértices representan las localizaciones con las llaves, tuercas y el robot, y las aristas representan las transiciones que puede hacer el robot para moverse entre vértices.
   - `(useable ?spa)`: Una determinada llave `?spa` es todavía usable. El enunciado nos dice que cada llave solo se puede usar una única vez. Con este predicado nos aseguramos de que una llave está disponible para ese único uso.
   - `(carrying ?rb ?spa)`: Este predicado sirve para indicar que el robot `?rb` está transportando la llave `?spa`.
   - `(loose ?nut)`: Sirve para indicar que la tuerca `?nut` está todavía floja.
   - `(tightened ?nut)`: Sirve para indicar que la tuerca `?nut` está apretada.

   Hasta aquí tenemos todos los predicados necesarios para modelar el problema.

2. En este caso nos piden un estado inicial y uno final. En cuando al estado inicial, podríamos modelarlo de la siguiente manera:

   ```
   (at kenny wp0)
   (at spanner1 wp1)
   (at nut1 wp2)
   (connected wp0 wp1)
   (connected wp1 wp2)
   (loose nut1)
   ```

   Con este estado inicial, estamos indicando que tenemos un único robot `kenny` situado en el *waypoint* `wp0`, y que debe apretar una tuerca: *nut1* que se encuentra en el *waypoint* `wp2`. Para ello, dispone de una única llave `spanner1` situada en el *waypoint* `wp1`. Además, el `waypoint wp0` está conectado con el `wp1`, que a su vez está conectado con `wp2`.

   En cuanto al estado final, sería igual al anterior, pero eliminando el predicado `loose nut1` que sería sustituido por:

   ```
   (tightened nut1)
   ```

3. En cuanto a los operadores, con la representación utilizada, tendríamos 3 operadores diferentes:

   - `(move ?rb ?from ?to)`: Para indicar que el robot `?rb` se mueve de una determinada localización de partida `?from` a una localización de destino `?to`. Para ello, el robot se debe encontrar en la localización de partida y, una vez acabada la acción, el robot dejará de estar en la localización de partida y pasará a la de destino.
   - `(pickup_spanner ?rb ?spa ?wp)`: El robot `?rb` recoge la llave `?spa` del *waypoint* `?wp`. Para ello, `?rb` y `?spa` se deben encontrar en el *waypoint* `?wp`, y además `?spa` debe ser usable. En los efectos indicaríamos que `?rb` está transportando `?spa` mediante el predicado creado para tal efecto, y por lo tanto, dejaría de estar en `?wp`.

- `(tighten_nut ?rb ?spa ?nut ?wp)`: El robot `?rb` apreta la tuerca `?nut`. Para ello, `?rb` debe estar transportando la llave `?spa` y la tuerca `?nut` debe estar floja. Además, tanto `?rb` como `?nut` se deben encontrar en el *waypoint* `?wp`. En los efectos indicaríamos que la tuerca `?nut` ha sido apretada, por lo que dejaría de estar floja, y además, la llave `?spa` dejaría de ser usable.

Por último, como el enunciado nos pide además modelizar uno de ellos de manera formal mediante sintaxis PDDL:

```
(:action tighten_nut
        :parameters (?rb - robot ?spa - spanner ?nut - nut ?wp - waypoint)
        :precondition (and (at ?rb ?wp)
             (at ?nut ?wp)
             (carrying ?rb ?spa)
             (loose ?nut))
        :effect (and (not (loose ?nut))(not (useable ?spa)) (tightened ?nut)))
)
```