Práctica PDDL

Inteligencia Artificial

Q1 2022-23

Pere Carrillo

Marc Ordóñez

Laura Pérez

Índice

1. Types

Los tipos se mantienen iguales en todas las extensiones, tenemos los objetos en general y un tipo en específico llamado *móvil*, que son aquellos objetos que se pueden transportar de una ubicación a otra con la ayuda del rover:

1.1 Object

```
rover movil lugar - object
```

Todos los elementos heredan de object, tenemos tres tipos de objeto: el rover, el móvil y el lugar, que conformarán los elementos básicos de nuestro problema. El rover nos servirá para mover los elementos a transportar. Los móviles, son los elementos del problema que serán transportados. Y finalmente el lugar, que servirá para indicar dónde están situados los objetos en el grafo, y dónde deben ir, según especifique el problema.

1.2 Móvil

```
persona suministro - movil
```

Hay dos móviles diferentes, la persona y el suministro, que serán transportados por el rover según las indicaciones del problema.

2. Nivel básico

En el nivel básico del problema debemos realizar el transporte de todos los suministros y personal. No hay limitación para la capacidad de carga de los rovers y pueden transportar a la vez tantos suministros y personal como quieran.

2.1 Predicados

En objeto lugar

```
(en ?o - object ?l - lugar)
```

Este predicado nos indica el lugar en el que está situado un objeto. Es muy importante para localizar un objeto.

Lo usamos para el rover, nos sirve para saber dónde está, o deja de estar después de una acción, en un momento e indicar el destino al que tiene que ir.

También lo usamos para indicar dónde está o deja de estar un móvil, es decir, una persona o un suministro.

Contiene rover movil

```
(contiene ?r - rover ?m - movil)
```

El segundo predicado que usaremos indica el móvil o los móviles (máximo dos) que contiene un rover. Por defecto ningún rover tendrá objetos, lo iremos modificando cada vez que coja o deje un objeto.

Camino lugar1 lugar2

```
(camino ?l1 - lugar ?l2 - lugar)
```

Finalmente tenemos el predicado que nos indica si existe un camino entre dos lugares. Sirve para crear el grafo por el cual los rovers transportaran los móviles, y restringir los caminos, ya que no podemos ir directamente de un lugar a cualquier otro. Los caminos son bidireccionales, por lo que si existe el predicado *camino* A *B* también añadiremos el predicado *camino* *B* A.

2.2 Functions

Petición

```
(peticion_persona ?l - lugar)
(peticion_suministro ?l - lugar)
```

Tenemos dos tipos de petición, una para cada tipo de móvil, ya sea persona como suministro. Cada petición la hace un lugar, y el objetivo del problema será que las peticiones de todos los lugares estén cubiertas.

Recursos restantes

```
(recursos_restantes)
```

Nos servirá para los casos en los que no haya recursos suficientes para cubrir todas las peticiones, de manera que se acabe la ejecución cuando para todos los lugares se cumplan sus peticiones o bien se acabe cuando no queden más recursos.

2.3 Operadores

Mover

```
(:action mover
      :parameters (?r - rover ?s - lugar ?d - lugar)
      :precondition (and (en ?r ?s) (camino ?s ?d))
      :effect (and (en ?r ?d) (not (en ?r ?s)))
)
```

La primera acción es mover, que desplaza un rover de un lugar a otro. Como precondición tenemos que el rover debe estar en un lugar y debe existir un camino del origen, es decir el lugar actual al destino. Esto tiene sentido ya que trasladar un rover que tiene como origen un lugar en el que no está, o enviarlo a un lugar al que no puede acceder no sería posible. El resultado de esta acción es que el rover ya no está en el origen y está en el destino.

Coger

```
(:action coger

      :parameters (?r - rover ?m - movil ?l - lugar)

      :precondition (and (en ?r ?l) (en ?m ?l) (not (contiene ?r
?m)))

      :effect (and (not (en ?m ?l)) (contiene ?r ?m))

)
```

La segunda acción hace que el rover coja un objeto. Como precondición tenemos que el movil y el rover tienen que estar en el mismo lugar y el rover no debe contener el

objeto, ya que no podríamos coger un objeto que esté en un lugar diferente al rover, y no tendría sentido coger un objeto que ya tenemos. Como efecto, el movil deja de estar en el lugar actual, y el rover contiene el objeto.

Dejar

```
(:action dejar_persona_no_final
      :parameters (?r - rover ?m - persona ?l - lugar)
      :precondition (and (en ?r ?l) (contiene ?r ?m) ?l) 0))
      :effect (and (en ?m ?l) (not (contiene ?r ?m)))
)

(:action dejar_persona_final
      :parameters (?r - rover ?m - persona ?l - lugar)
      :precondition (and (en ?r ?l) (contiene ?r ?m) (>
(peticion_persona ?l) 0))
      :effect (and (not (contiene ?r ?m)) (decrease
(peticion_persona ?l) 1) (decrease (recursos_restantes) 1))
)

(:action dejar_suministro_no_final
      :parameters (?r - rover ?m - suministro ?l - lugar)
      :precondition (and (en ?r ?l) (contiene ?r ?m) ?l) 0))
      :effect (and (en ?m ?l) (not (contiene ?r ?m)))
)

(:action dejar_suministro_final
      :parameters (?r - rover ?m - suministro ?l - lugar)
      :precondition (and (en ?r ?l) (contiene ?r ?m) (>
(peticion_suministro ?l) 0))
      :effect (and (not (contiene ?r ?m)) (decrease
(peticion_suministro ?l) 1) (decrease (recursos_restantes) 1))
)
```

Finalmente tenemos todas las acciones de dejar un móvil. Tendremos dos tipos para cada tipo de móvil, la de dejarlo en un lugar final, es decir, que haya hecho una petición de su tipo, o la de dejarlo en un lugar que no es final.

Todas tienen como precondición que el rover debe estar en un lugar, contiene un móvil y en el lugar haya una petición del tipo de móvil en el rover. El resultado de la operación es que el rover deja de contener el objeto, se resta una petición al lugar y se restan los recursos restantes.

Si dejamos el móvil en un lugar final no añadimos el predicado “en móvil lugar” al lugar destino, ya que una vez cumplida la petición no volveremos a tocar ese móvil y por lo

tanto mejor si lo quitamos y así cada vez hay menos objetos posibles para los otros operadores.

3. Extensión 1

En la primera extensión del problema se limita el máximo de móviles que puede transportar un rover a la vez. De manera que podrá contener dos personas o una carga de suministros en un mismo instante. Para añadir esta restricción, usamos el nivel básico y hacemos los cambios que explicaremos a continuación.

3.1 Predicados

Se mantienen los predicados del nivel básico

3.2 Functions

Agregamos una nueva función:

Libre

```
(libre ?r - rover)
```

Nos servirá para indicar la capacidad de un rover, y si podemos coger más móviles o no. Por defecto estará totalmente libre, cada vez que el rover coja o deje un objeto, se modificará su capacidad restante.

3.3 Operadores

Coger

Dividiremos la acción de coger en dos, coger_persona y coger_suministro, que se diferencian en el espacio libre del rover como precondición y efecto. Necesitando y ocupando un espacio por persona, y dos por suministro.

```
(:action coger_persona
      :parameters (?r - rover ?m - persona ?l - lugar)
      :precondition (and (en ?r ?l) (en ?m ?l) (not (contiene ?r
?m)) (>= (libre ?r) 1))
      :effect (and (not (en ?m ?l)) (contiene ?r ?m) (decrease
(libre ?r) 1))
)
```

```
(:action coger_suministro
      :parameters (?r - rover ?m - suministro ?l - lugar)
      :precondition (and (en ?r ?l) (en ?m ?l) (not (contiene ?r
?m)) (>= (libre ?r) 2))
      :effect (and (not (en ?m ?l)) (contiene ?r ?m) (decrease
(libre ?r) 2))
)
```

Dejar

También cambiará el efecto de dejar un móvil, liberando un o dos espacios dependiendo del tipo de móvil.

4. Extensión 2

Para esta segunda extensión, se añade el combustible limitado de los rovers. Se carga a principio de día, de manera que solo pueden hacer un número limitado de movimientos. Cada vez que el rover se desplace, gastará una unidad de combustible. Habrá dos versiones, una en la que se minimice el gasto de combustible y otra en la que no se haga.

Tanto en la extensión 2 como en la 3, si el rover se queda sin combustible sin haber realizado el número máximo de peticiones (tantas como personas y suministros haya), no se dará ninguna respuesta. En caso de realizar todas las peticiones posibles sí que se nos mostrará la planificación.

3.1 Predicados

Se mantienen los predicados del nivel básico

3.2 Functions

Agregamos una nueva función:

Combustible

```
(combustible ?r - rover)

(combustible_total)
```

Nos servirá para indicar el combustible de un rover, y si podemos desplazarnos o no. Cada vez que el rover se mueva, restamos 1 al combustible restante del rover y al combustible total.

Estos fluentes también nos serán útiles para minimizar el gasto de combustible. Para ello, al final del problema añadiremos una métrica maximizando el combustible restante.

```
(:metric maximize (combustible_total))
```

3.3 Operadores

Mover

```
(:action mover
:parameters (?r - rover ?s - lugar ?d - lugar)
:precondition (and (en ?r ?s) (camino ?s ?d) (> (combustible ?r)
0))
```

```
:effect (and (en ?r ?d) (not (en ?r ?s)) (decrease (combustible
?r) 1))
)
```

Modificaremos la acción de mover. De manera que antes de hacerlo comprobemos que tenemos combustible suficiente, y si nos hemos movido, restemos en una unidad el combustible restante.

5. Extensión 3

Finalmente, para la última extensión, las peticiones tienen prioridad [1-3] (En nuestro caso puede haber más), de manera que buscamos el plan que maximice la prioridad de las peticiones servidas. Como en la segunda extensión, habrá dos versiones, una en la que no importe el combustible y otra en la que se optimice una combinación entre prioridades y combustible total, asignando diferentes pesos a cada criterio para poder comparar soluciones.

5.1 Tipos

Hemos añadido el tipo "prioridad", para definir las prioridades de cada petición. No nos limitamos a 3 prioridades como dice el enunciado, sino que hemos extendido nuestro código para que sea el usuario final quien decida cuántas prioridades hay y qué valor tienen. Para hacerlo solo tiene que añadir una variable más del tipo prioridad con el nombre que quiera y definir su orden como veremos más adelante.

5.2 Predicados

Se mantienen los predicados del nivel anterior.

5.3 Functions

Para manejar las prioridades hemos modificado las funciones petición persona y petición suministro, añadiendoles un nuevo parámetro de tipo prioridad.

```
(peticion_persona ?l - lugar ?p - prioridad)
```

```
(peticion_suministro ?l - lugar ?p - prioridad)
```

Con esto permitimos que cada lugar pueda pedir varias personas o suministros con distinta prioridad.

Hemos añadido la función valor_prioridad `(valor_prioridad ?p - prioridad)` que nos permite escoger el orden de las prioridades. Por ejemplo, si tenemos 3 prioridades Alta, Media y Baja le pondremos valor 3 a Alta, 2 a Media y 1 a Baja.

También hemos agregado las funciones num_max_prioridad y suma_prioridades, que no tienen parámetros y sirven para definir la métrica.

5.4 Operadores

Hemos modificado todos los operadores de dejar para añadir el cambio a petición persona y suministro. Además a los de dejar_*_final hemos añadido un cálculo a los efectos para calcular la suma de prioridades correctamente.

5.5 Métrica

Para la primera versión la métrica es la suma de las prioridades.

Para la segunda versión hemos decidido que la métrica sería 2*combustible + prioridades, ya que el combustible restante por lo general es menor a la suma de las prioridades y de esta manera los equilibramos.

```
(:metric maximize (+ (* (combustible_total) 2) (* (suma_prioridades) 1)))
```

6. Modelado de problemas

6.1 Objetos

Para cada problema inicializaremos todos los objetos que hay en el dominio, es decir, lugares, rovers, personas y suministros.

6.2 Estado inicial

Nivel básico

Crearemos el grafo, asignando caminos entre lugares. Asignaremos un lugar a los demás objetos, de manera que cada uno esté en un lugar. Finalmente se harán las peticiones, tanto de personas como de suministros.

Extensión 1

Además de lo inicializado en el nivel básico, deberemos determinar el espacio libre en cada rover, por defecto 2. Y también indicaremos los recursos restantes, que serán todos los móviles que hayamos añadido en el problema.

Extensión 2

Indicamos el combustible para cada rover.

6.3 Objetivo

```
(:goal (or
      (forall (?l - lugar) (and ?l) 0) ?l) 0)))
      (= (recursos_restantes) 0)
))
```

El objetivo del nivel básico hasta la segunda extensión es cubrir todas las peticiones de personas y suministros de los lugares. Y a malas, distribuir los recursos disponibles.

6.4 Métrica

En los ejercicios que se nos pide minimizar o maximizar ciertos parámetros tendremos que hacer uso de métricas.

Extensión 2

En la segunda extensión se pide que se minimice el combustible gastado. Esto equivale a decir maximizar el combustible restante. De manera que nos queda:

```
(:metric maximize (+ (combustible R1) ... (combustible Rn)))
```

7. Generador juegos de prueba

Hemos creado un generador de juegos de prueba con python. Tenemos un programa para cada extensión (nivel básico, 1, 2.2, 3.2) adecuando la información que se genera a los predicados y funciones que se usan en cada extensión.

El programa contiene una serie de *flags* que son los que nos permitirán crear juegos con distintos números de rovers, lugares, etc. Estos flags son:

- **Lugares (-l)**: Número de lugares que queremos generar en el grafo del problema.
- **Rovers (-r)**: Número de rovers.
- **Personas (-p)**: Número de personas.
- **Suministros (-s)**: Número de suministros.
- **Num (-n)**: Número de semilla que determinará el valor de todos los elementos del problema que se generen de manera aleatoria. De esta manera siempre que usemos el mismo número se creará el juego de pruebas de la misma manera.
- **Seed (-ss):** Semilla usada para determinar la forma del mapa y qué caminos se crean. Todos los juegos de prueba con el mismo número de lugares y la misma seed del mapa crearán el mismo grafo con los mismos caminos, sin importar la otra seed, ni el número de rovers ni ninguna otra variable del problema.

Hemos decidido usar estas dos semillas porque aunque el generador debe crear juegos de prueba aleatorio, vimos necesario añadir una opción que nos permitiese recrear el mismo mapa en todas las versiones sin importar el número de rovers, personas, suministros, etc. De esta forma podemos analizar mejor el comportamiento del programa cuando tenemos el mismo mapa pero variamos el número de alguno de los otros elementos.

La otra seed llamada *num* nos da el control sobre el resto de variables. Si queremos generar un juego A con 4 rovers y otro jeugo B con 8 pero queremos que los 4 rovers del juego A mantengan su posición en el juego B la forma de hacerlo es usar el mismo valor para el valor *num*. Es decir, al usar la misma semilla hacemos que los juegos de prueba más grandes sean una extensión de los pequeños pero manteniendo una parte en común.

Por ejemplo, en el generador básico asignamos cada persona a un lugar, asignamos cada persona a un lugar, asignamos cada suministro a un lugar y se realizan peticiones de suministros y de personas desde todos los lugares. Todos estas asignaciones se realizan de manera aleatoria a partir de la semilla *num*. De este modo si tenemos dos juegos de prueba con la misma semilla pero en uno tenemos dos suministros y en el otro cinco y

el número de lugares es el mismo, la ubicación inicial de los dos primeros suministros será la misma para ambos juegos. Si no queremos que esto pase, es tan fácil como usar una semilla diferente para el segundo juego.

Hemos decidido hacerlo de esta forma para que en los juegos de prueba que analizamos para ver el comportamiento del programa si aumentábamos el número de rovers solía tardar más pasos, lo cual no tenía mucho sentido. Nos dimos cuenta que esto se debía a que al aumentar los rovers cambiaba el problema por completo, incluido las ubicaciones iniciales de los rovers. De modo que en la versión con más rovers podían estar colocados de una forma que hiciese que tardasen más.

Gracias a este cambio, si usamos la misma semilla *num* y *seed* manteniendo el número de lugares, de personas y de suministros, se generará el mismo grafo para el mapa y la ubicación de los rovers será la misma (rover R0 en la misma ubicación para todos los juegos, lo mismo con R1, etc).

Un ejemplo de comando para generar un juego de pruebas sería:

```
python generator2.py --lugares=3 --rovers=2  --personas=2 --suministros=1 --num=1 --seed=1111
```

o también la versión corta:

```
python generator2.py -l 3 -r 2 -p 2 -s 1 -n 1 -ss 1111
```

Este comando nos generaría un juego compatible con la extensión 2 con 3 lugares, 2 rovers, 2 personas, 1 suministro, la semilla num 1 y la semilla seed (la del mapa) 1111.

7.1 Evolución del tiempo según el número de rovers

Una vez hecho el generador de problemas hemos hecho pruebas para ver la evolución temporal de la resolución de los problemas aumentando el número de rovers.

Para eso, con cada generador, hemos usado la misma seed y num, de modo que mantengamos el mapa y las variables, y que lo único que cambie sean los rovers.

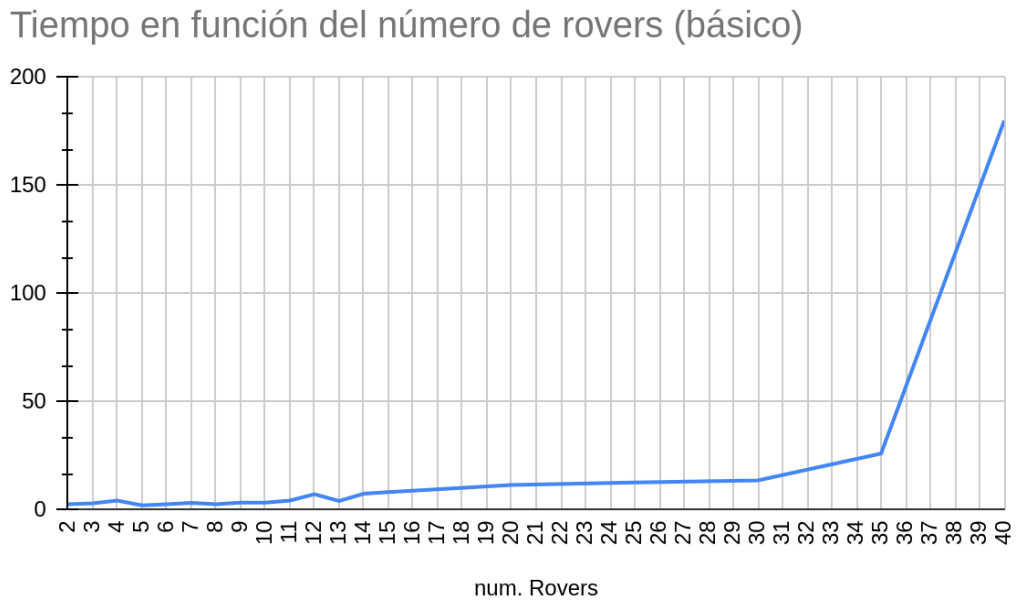
Para cada extensión hemos mirado de usar de 1 rover hasta 40. Hemos hecho todos los casos desde un rover hasta 15 y a partir de ahí hemos ido en intervalos de 5 en 5 (20, 25...).

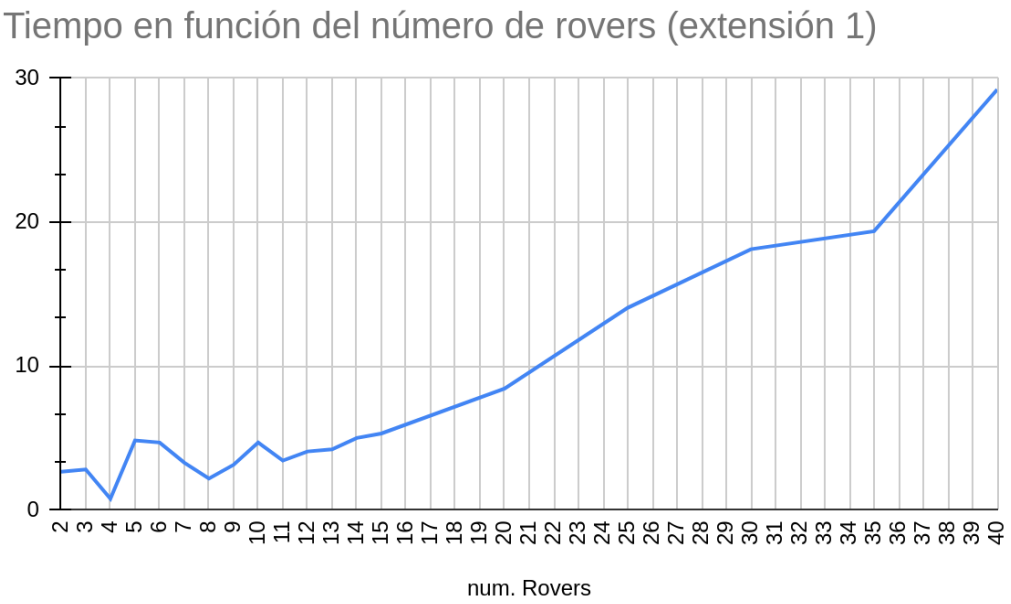
Al mantener el mismo valor para *seed* y *num* hacemos que el juego de pruebas sea idéntico y sólo varie el número de rovers. Esto hace que el tiempo obtenido sea más fiable, ya que cuando generábamos juegos aleatorios en ciertas ocasiones los juegos con más rovers realizaban más movimientos, ya que la posición inicial y las peticiones variaban respecto a los otros juegos y la solución podía tener un número de movimientos considerablemente mayor a las versiones con menos rovers. Realizamos la

prueba en la extensión básica y en la 1 porque en las otras dos al usar métrica la ejecución tardaba mucho, por lo que no era viable seguir con más ejemplos ya que no llegaríamos a ninguna conclusión más.

Los juegos de prueba que hemos usado para el experimento tienen 15 lugares, n rovers, 8 personas y 8 suministros.

Los resultados de ambas extensiones quedan reflejadas en las siguientes gráficas, donde efectivamente se puede observar que al aumentar el número de rovers también aumenta el tiempo que tarda el programa. Aún así, hasta aproximadamente los 15 rovers no se aprecia una tendencia clara, ya que en todos los casos nos da una solución en poco menos de 10 segundos.





Podemos ver que en la primera extensión hay un gran incremento de 35 a 40. En el segundo pero, podemos observar una evolución más lineal.

7.2 Evolución del tiempo según el número de personas

Además de hacer pruebas con el número de rovers, hemos querido comprobar la evolución cambiando el número de personas. En este caso también hemos hecho juegos de prueba con 1..15 personas y luego 20, 25, 30, 35 y 40.

Podemos observar claramente una evolución exponencial. Creemos que esto se debe a que el rover tendrá más peticiones que podrá satisfacer, por lo que realizará más movimientos y el planificador tendrá que tomar más decisiones para decidir a qué lugar manda a cada persona.

8. Juegos de prueba

Para generar los juegos de prueba, hemos programado un código en Python, de manera que nos cree un estado inicial según los parámetros introducidos. Podemos determinar el número de lugares, rovers, de personas, suministros. Además, tendremos dos parámetros para randomizar los otros parámetros del problema, de manera que podamos replicar el experimento en cualquier momento. Uno será la seed que nos creará un mapa, con los lugares indicados y caminos aleatorios, según el número introducido. Y el otro que nos servirá para inicializar todo el resto de valores que se creen de manera aleatoria, como pueden ser: combustible restante, posición inicial de rovers, suministros y personas, prioridades, número de peticiones, entre otros.

Para cada extensión hemos generado un problema diferente, dado el estado inicial, nos ha salido un problema muy simple, y por consiguiente, una solución muy corta.

Nivel básico

```
0: COGER R0 P0 L0
1: COGER R0 S0 L0
2: DEJAR_SUMINISTRO_FINAL R0 S0 L0
3: MOVER R1 L0 L1
4: MOVER R1 L1 L2
5: DEJAR_PERSONA_FINAL R0 P0 L0
6: COGER R1 P1 L2
7: DEJAR_PERSONA_FINAL R1 P1 L2
8: REACH-GOAL
```

Extensión 1

```
0: COGER_PERSONA R0 P0 L0
1: COGER_SUMINISTRO R1 S0 L0
2: DEJAR_SUMINISTRO_FINAL R1 S0 L0
3: MOVER R1 L0 L1
4: DEJAR_PERSONA_FINAL R0 P0 L0
5: MOVER R1 L1 L2
6: COGER_PERSONA R1 P1 L2
7: DEJAR_PERSONA_FINAL R1 P1 L2
8: REACH-GOAL
```

Extensión 2

```
0: COGER_PERSONA R0 P0 L0
1: COGER_SUMINISTRO R1 S0 L0
2: DEJAR_PERSONA_FINAL R0 P0 L0
3: MOVER R0 L0 L1
4: MOVER R0 L1 L2
5: COGER_PERSONA R0 P1 L2
6: DEJAR_PERSONA_FINAL R0 P1 L2
7: DEJAR_SUMINISTRO_FINAL R1 S0 L0
8: REACH-GOAL
```

Extensión 3

```
0: COGER_PERSONA R0 P0 L0
1: MOVER R1 L0 L1
2: MOVER R1 L1 L2
3:  DEJAR_PERSONA_FINAL  R0  P0  L0
ALTA
4: COGER_SUMINISTRO R0 S0 L0
5: DEJAR_SUMINISTRO_FINAL R0 S0 L0
ALTA
6: COGER_PERSONA R1 P1 L2
7:  DEJAR_PERSONA_FINAL  R1  P1  L2
ALTA
8: REACH-GOAL
```

Podemos comprobar que, aunque la solución sea muy simple, el programa cumple con su funcionalidad de repartir los pedidos. Un rover coge un suministro o una persona y lo deja en un lugar donde se haya pedido un móvil de su tipo. En estos juegos de prueba la

ejecución acaba por falta de recursos en vez de por haber cumplido con todos los pedidos.

Para agilizar los juegos de prueba, y demostrar más claramente que los dominios funcionan, en vez de crear problemas con nuestro generador hasta encontrar uno con el que podamos ver ciertos comportamientos, hemos creado unos a mano para comprobar diferentes funcionalidades.

También usaremos un juego de pruebas creado con el programa generador para ver cómo se comporta el planificador al enfrentarse al mismo juego (con cambios mínimos por las necesidades añadidas de cada extensión) y así poder ver qué variaciones hay.

8.1 Nivel básico

Con esta versión, tenemos pocas restricciones, y por lo tanto, comprobaremos las funcionalidades más básicas.

Experimento 1 (exp1_basico)

En este experimento queremos comprobar el correcto funcionamiento, con un ejemplo muy básico. Dado un camino, que un rover lleve un objeto de una punta a otra. Tenemos tres lugares L0-L1-L2, un rover en L0, una persona en L2, y L0 hace una petición de una persona.

```
0: MOVER R0 L0 L1
1: MOVER R0 L1 L2
2: COGER R0 P0 L2
3: MOVER R0 L2 L1
4: MOVER R0 L1 L0
5: DEJAR_PERSONA_FINAL R0 P0 L0
6: REACH-GOAL
```

Obtenemos el resultado esperado, el rover va a buscar a la persona en L2, la coge y la deja en L0.

Experimento 2 (exp2_basico)

En este segundo experimento, comprobamos que se cumplan las condiciones del goal. Para ello, agregamos dos personas más en L2.

```
0: MOVER R0 L0 L1
1: MOVER R0 L1 L2
2: COGER R0 P2 L2
3: MOVER R0 L2 L1
4: MOVER R0 L1 L0
```

```
5: DEJAR_PERSONA_FINAL R0 P2 L0
6: REACH-GOAL
```

Las acciones siguen siendo las mismas pero acaba en el momento en el que hemos cumplido con todas las peticiones. En el experimento hecho con el generador tenemos un ejemplo de finalización de ejecución por falta de recursos.

Experimento generador

En este experimento usaremos el generador con los mismos parámetros para todas las extensiones, así podremos ver como van evolucionando los resultados dadas las entradas. El comando que hemos usado para generar este problema ha sido:

```
python generator3.py --lugares=6 --rovers=2 --suministros=1 --num=0 --personas=2
--seed=3
```

Como estado inicial tendremos el un rover en L1, un rover en L4, dos personas en L0 y un suministro en L1.

Las peticiones de personas son :

L0 -> 1, L1 -> 2, L2 -> 0, L3 -> 3, L4 -> 3, L5 -> 1

Las peticiones de suministros son:

L0 -> 2, L1 -> 1, L2 -> 2, L3 -> 1, L4 -> 2, L5 -> 2

El resultado ha sido:

```
0: COGER R0 S0 L1
1: MOVER R0 L1 L0
2: DEJAR_SUMINISTRO_FINAL R0 S0 L0
3: COGER R0 P1 L0
4: COGER R0 P0 L0
5: DEJAR_PERSONA_FINAL R0 P0 L0
6: MOVER R0 L0 L1
7: DEJAR_PERSONA_FINAL R0 P1 L1
8: REACH-GOAL
```

En este experimento la ejecución se acaba en el momento en el que nos quedamos sin recursos. Podemos ver como el rover coge a la persona que está en su misma posición y la lleva a L0. Lo siguiente que hace es coger a las dos personas de L0, la primera persona la deja en el mismo lugar para cumplir la petición, y para la segunda se vuelve a L1 para cumplir otra petición.

8.2 Extensión 1

En esta extensión tenemos la limitación de capacidad de un rover.

Experimento 1 (exp1_1)

En este experimento comprobaremos el límite de capacidad de un rover, para ello, tendremos un camino de tres nodos L0-L1-L2, una persona en L0, un suministro en L1, y finalmente una petición de una persona y un suministro de L2

```
0: COGER_PERSONA R0 P0 L0
1: MOVER R0 L0 L1
2: MOVER R0 L1 L2
3: DEJAR_PERSONA_FINAL R0 P0 L2
4: MOVER R0 L2 L1
5: COGER_SUMINISTRO R0 S0 L1
6: MOVER R0 L1 L2
7: DEJAR_SUMINISTRO_FINAL R0 S0 L2
8: REACH-GOAL
```

Podemos ver que el rover coge a la persona, pasa por L1 sin coger el suministro, ya que no le queda espacio, deja a la persona en L2 y se vuelve a buscar el suministro.

Experimento generador

Matenemos el mismo estado inicial que en el nivel básico. Pero le añadimos la capacidad libre de cada rover, dejando dos espacios.

```
0: COGER_SUMINISTRO R0 S0 L1
1: MOVER R0 L1 L0
2: DEJAR_SUMINISTRO_FINAL R0 S0 L0
3: COGER_PERSONA R0 P1 L0
4: COGER_PERSONA R0 P0 L0
5: DEJAR_PERSONA_FINAL R0 P0 L0
6: MOVER R0 L0 L1
7: DEJAR_PERSONA_FINAL R0 P1 L1
8: REACH-GOAL
```

Vemos que el resultado es igual que en el nivel básico. Esto tiene sentido ya que en ningún momento tenemos la necesidad de coger dos suministros o un suministro con una persona, por lo tanto, no entra en conflicto.

8.3 Extensión 2

Este dominio añade la restricción del límite de combustible. Para los juegos de prueba hemos usado la versión que tiene en cuenta la minimización de combustible, ya que ambas versiones son bastante similares y la segunda es más completa.

Experimento 1 (exp1_2)

En este experimento comprobaremos como dos rovers se ayudan para llevar una persona a un lugar, con un combustible limitado. Tendremos un camino de cuatro nodos, en L0 tendremos un rover con dos de combustible y en L3 otro rover con dos de combustible, además, tendremos una persona en L1 y L3 hará una petición de una persona.

```
0: MOVER R0 L0 L1
1: COGER_PERSONA R0 P0 L1
2: MOVER R0 L1 L2
3: DEJAR_PERSONA_NO_FINAL R0 P0 L2
4: MOVER R1 L3 L2
5: COGER_PERSONA R1 P0 L2
6: MOVER R1 L2 L3
7: DEJAR_PERSONA_FINAL R1 P0 L3
8: REACH-GOAL
```

En este ejemplo el primer rover deja el objeto en L2 ya que se ha quedado sin gasolina, y viene el segundo rover a coger a la persona y llevarla a L4.

Experimento generador

Mantenemos el mismo estado inicial que en la primera extensión

```
0: MOVER R0 L1 L0
1: COGER_PERSONA R0 P0 L0
2: COGER_PERSONA R0 P1 L0
3: MOVER R0 L0 L1
4: DEJAR_PERSONA_FINAL R0 P0 L1
5: DEJAR_PERSONA_FINAL R0 P1 L1
6: COGER_SUMINISTRO R0 S0 L1
7: DEJAR_SUMINISTRO_FINAL R0 S0 L1
8: REACH-GOAL
```

Respecto a la versión anterior vemos que cambia el orden en el que se realizan las acciones. Ahora primero coge a las personas para cumplir la petición y luego coge el suministro. Vemos que realiza el mismo número de movimientos porque está transportando todos los móviles disponibles (personas y suministros) pero en la versión anterior en ningún caso estaba cargando más de dos personas o un suministro a la vez, por lo que, pese a tener esta limitación en la extensión 2, no repercute en la planificación.

8.4 Extensión 3

En la última extensión tenemos prioridades.

Experimento 1 (exp1_3)

En este experimento comprobamos que un rover entregará un suministro al lugar con una petición con prioridad más alta. Para ello crearemos un grafo estrella, con un rover y un suministro en L0. Tendremos L0 conectado a L1, que a la vez lo estará con L2, L3 y L4 (de manera que L1 es el centro de la estrella); cada lugar tendrá una petición de un suministro, de prioridad baja a alta respectivamente.

```
0: COGER_SUMINISTRO R0 S0 L0
1: MOVER R0 L0 L1
2: MOVER R0 L1 L4
3: DEJAR_SUMINISTRO_FINAL R0 S0 L4 ALTA
4: REACH-GOAL
```

Podemos comprobar que el rover dejará el suministro en L4, lugar con la petición de mayor prioridad.

Experimento generador

```
0: COGER_SUMINISTRO R0 S0 L1
1: MOVER R1 L4 L1
2: MOVER R1 L1 L0
3: COGER_PERSONA R1 P0 L0
4: DEJAR_SUMINISTRO_FINAL R0 S0 L1 ALTA
5: COGER_PERSONA R1 P1 L0
6: DEJAR_PERSONA_FINAL R1 P0 L0 BAJA
7: DEJAR_PERSONA_FINAL R1 P1 L0 MEDIA
8: REACH-GOAL
```

Si analizamos en detalle los juegos de prueba adjuntos a la práctica (este en concreto es el de la carpeta Joc Proves Complet -> problem roverPro3_3_0_2.pddl), veremos que los dos rovers tienen combustible de sobras para realizar muchos más movimientos y hay peticiones de prioridad alta.

Peticiones personas:

- L0: Baja 1, Media 2, Alta 0
- L1: Baja 3, Media 3, Alta 1
- L2: Baja 0, Media 0, Alta 0
- L3: Baja 3, Media 4, Alta 2
- L4: Baja 0, Media 1, Alta 4
- L5: Baja 4, Media 2, Alta 2

Peticiones suministros:

- L0: Baja 2, Media 1, Alta 2
- L1: Baja 1, Media 2, Alta 2
- L2: Baja 2, Media 2, Alta 0
- L3: Baja 1, Media 0, Alta 2
- L4: Baja 0, Media 0, Alta 0
- L5: Baja 1, Media 1, Alta

Como en nuestra métrica damos el doble de importancia al combustible restante que a la prioridad de las peticiones servidas, teniendo en cuenta que cada movimiento gasta 1

de combustible y cada petición nos da una ganancia de 3 si es Alta, 2 si Media o 1 si es Baja, a menos que con muy pocos movimientos consigamos satisfacer una prioridad Alta, siempre elegirá satisfacer peticiones de menor prioridad pero que aumenten el ahorro de combustible. En este caso, una petición baja y media de personas. En cambio, la petición de suministro sí que se hace una de prioridad Alta porque solo hace falta gasta uno de combustible y ganamos 3 puntos por la prioridad Alta.