

PRÁCTICA 3: USO DEL PLANIFICADOR FAST DOWNWARD PARA RESOLVER PROBLEMAS AMBIENTADOS EN LA TIERRA MEDIA DE EL SEÑOR DE LOS ANILLOS



**UNIVERSIDAD
DE GRANADA**

TÉCNICAS DE LOS SISTEMAS INTELIGENTES

Elena Torres Fernández
Curso académico 2024-2025

4 de junio de 2025

Ejercicio 1

En este ejercicio planteamos el modelo base de la práctica (viajar para desplazarse y trabajar para extraer recursos). Hago algunos comentarios al respecto:

- En cuanto a la representación del **mapa de conexiones entre ciudades**, he optado por definir el predicado (`conectado ?a - Localizacion ?b - Localizacion`) en el archivo del dominio y ponerlo duplicado para cada par de ciudades intercambiado la ciudad *a* por la *b* en el archivo del problema. Así, nos aseguramos que dentro de la acción de **Viajar** se pueda ir de un ?origen a un ?destino porque existe dicho camino en ese sentido concreto. No bastaría con ponerlo sólo una vez, ya que estaríamos permitiendo viajar en un único sentido entre cada par de variables.
- En cuanto al tratamiento de las **constantes**, no las he añadido las constantes en el archivo del problema dentro del apartado de los objetos, donde solo están las localizaciones y los personajes; las he usado directamente en las inicializaciones de los predicados (`personajeEs Enano1 Enano`) y (`recursoEn Mineral Moria`). Sí he definido las constantes en el archivo *dominio1.pddl* en el apartado especial de `:constant`, creadas como tipoPersonaje (subtipo de object) y Recurso (subtipo de movable). Además, **movible** es un subtipo de **object**. Hacemos esta división de objetos entre movibles y no movibles para que en el predicado (`en ?obj - movable ?x - Localizacion`) podamos diferenciar los objetos que ocupan posiciones (Personajes y Recursos) de los que no.
- El **objetivo** de este ejercicio es que haya algún personaje fabricando madera, sin especificar quién. No obstante, sabemos por las condiciones del problema que será el Enano1 el que fabricará madera, pues el Enano2 no está disponible y el Hobbit es perezoso; pero dejamos que eso lo deduzca el planificador. De hecho, el goal del ejercicio lo indicamos con el predicado (`alguienTrabajandoEn Madera`), sin especificar el personaje que está trabajando en dicho recurso. Además, incluimos el predicado (`trabajandoEn ?p - Personaje ?r - Recurso`), que sí incluye al personaje que está produciendo el recurso. Este segundo predicado se podría omitir sin problema porque no aparece en ninguna precondition ni es el objetivo del problema (igual ocurre en el ejercicio 5), pero se ha mantenido porque no supone ningún sobreesfuerzo computacional en este caso y sí vendrá bien en las próximas tareas.

La secuencia de acciones que nos devuelve el **planner** es la proporcionada en la figura 1. Vemos cómo ha deducido correctamente que el personaje Enano1 es el único posible para la tarea y que acaba recogiendo Madera en una ciudad que es, efectivamente, un nodo de Madera (Fangorn).

```
1(viajar enano1 tharbad helmsdeep)
2(viajar enano1 helmsdeep isengard)
3(viajar enano1 isengard fangorn)
4(extraerrecurso enano1 fangorn madera)
5; cost = 4 (unit cost)
```

Figura 1: Plan obtenido al ejecutar los archivos *dominio1.pddl* y *problema1.pddl* usando *seq-opt-lmcut*

Ejercicio 2

En este ejercicio ampliamos el modelo base para poder formar una Comunidad reducida y destruir el anillo. Hago algunos comentarios al respecto:

- Además de los **predicados** ya definidos en el anterior ejercicio, añadimos los siguientes para manejar correctamente a la comunidad: (`comunidad ?h - Personaje ?m - Personaje`), (`enComunidad ?p - Personaje`) y (`comunidadFormada`). También incluimos más predicados para gestionar la recolección de objetos y asegurar que el primero que se recoja sea el Anillo, los cuales son: (`(portadorDe ?p - Personaje ?rec - Recurso)`, (`lugarDestruccion ?loc - Localizacion`), (`anilloDestruido`), (`anilloRecogido`), (`mithrilRecogido`) y (`espadaRecogida`).
- Los **predicados sin argumentos** los uso a modo de variables bandera. Por ejemplo, uso (`comunidadFormada`) para que no se forme más de una comunidad, pues solo permitimos una.
- En cuanto a la gestión de la **disponibilidad** de los personajes, solo he considerado la precondition de `disponible` para la acción `formarComunidad`, en el resto de acciones `viajarComunidad`, `recogerObjeto` y `destruirAnillo` no la pongo porque ya va implícita al haberse creado la comunidad y pertenecer dichos personajes a la misma.

La secuencia de acciones que nos devuelve el **planner** es la proporcionada en la figura 2. Vemos cómo, efectivamente, la comunidad que se podía formar en un menor número de acciones era aquella constituida por el hobbit4 y el mago1. También verificamos cómo viajan conjuntamente hasta Orodruin para poder destruir el Anillo de forma consistente.

```
1 (viajar hobbit4 bree rivendell)
2 (formarcomunidad hobbit4 mago1 rivendell)
3 (recogerobjeto hobbit4 rivendell anillo)
4 (viajarcomunidad rivendell moria hobbit4 mago1)
5 (recogerobjeto hobbit4 moria mithril)
6 (viajarcomunidad moria lothlorien hobbit4 mago1)
7 (recogerobjeto hobbit4 lothlorien espada)
8 (viajarcomunidad lothlorien amonhen hobbit4 mago1)
9 (viajarcomunidad amonhen deadmarshes hobbit4 mago1)
10 (viajarcomunidad deadmarshes minasmorgul hobbit4 mago1)
11 (viajarcomunidad minasmorgul orodruin hobbit4 mago1)
12 (destruiranillo hobbit4 orodruin)
13 ; cost = 12 (unit cost)
```

Figura 2: Plan obtenido al ejecutar los archivos *dominio2.pddl* y *problema2.pddl* usando *seq-opt-lmcut*

Ejercicio 3

Este ejercicio es una variante del 2 considerando **varios tipos de comunidades**. Para cambiar el tipo de comunidad necesitamos modificar, dentro del fichero de dominio, el predicado (*comunidad ?p1 - Personaje ?p2 - Personaje*) añadiéndole tantos argumentos como Personajes nuevos queramos que tenga la comunidad (luego se especifica el tipo de Personaje en las acciones). El resto de predicados siguen iguales.

En cuanto a las acciones, solo necesitamos alterar *viajarComunidad* y *formarComunidad* para incluir a más personajes en los argumentos, todos ellos deben estar disponibles y ser del tipo que se especifica según la comunidad. Es buena señal que no necesitemos modificar la acción *recogerObjeto*, pues solo afecta a uno de los Hobbits de la comunidad, independientemente del tipo de comunidad (todas las comunidades tienen al menos 1 Hobbit).

Hechos estos comentarios, pasamos a construir la **tabla de tiempos y planes** para cada tipo de comunidad (tabla 1).

| Tamaño de la Comunidad | Tiempo de planificación ("Planner time") | Longitud del Plan encontrado ("Plan length") |
|----------------------------|---|---|
| 1 hobbit y 1 mago | 0.51s | 12 |
| 2 hobbits y 1 mago | 8.21s | 14 |
| 3 hobbits y 1 mago | 31.86s | 15 |
| 3 hobbits, 1 mago y 1 elfo | 495.13s | 18 |

Cuadro 1: Tabla de tiempos y plan obtenido

Los archivos *dominio3.pddl* y *problema3.pddl* mandados cuentan con la implementación de la comunidad de mayor tamaño, cuya ejecución tarda 8.25 minutos.

```
(viajar hobbit4 bree rivendell)
(formarcomunidad hobbit4 mago1 rivendell)
(recogerobjeto hobbit4 rivendell anillo)
(viajarcomunidad rivendell moria hobbit4 mago1)
(recogerobjeto hobbit4 moria mithril)
(viajarcomunidad moria lothlorien hobbit4 mago1)
(recogerobjeto hobbit4 lothlorien espada)
(viajarcomunidad lothlorien amonhen hobbit4 mago1)
(viajarcomunidad amonhen deadmarshes hobbit4 mago1)
(viajarcomunidad deadmarshes minasmorgul hobbit4 mago1)
(viajarcomunidad minasmorgul orodruin hobbit4 mago1)
(destruiranillo hobbit4 orodruin)
; cost = 12 (unit cost)
```

(a) 1H + 1M

```
(viajar hobbit1 hobbiton bree)
(viajar hobbit1 bree rivendell)
(viajar hobbit4 bree rivendell)
(formarcomunidad hobbit1 hobbit4 mago1 rivendell)
(recogerobjeto hobbit1 rivendell anillo)
(viajarcomunidad rivendell moria hobbit1 hobbit4 mago1)
(recogerobjeto hobbit1 moria mithril)
(viajarcomunidad moria lothlorien hobbit1 hobbit4 mago1)
(recogerobjeto hobbit1 lothlorien espada)
(viajarcomunidad lothlorien amonhen hobbit1 hobbit4 mago1)
(viajarcomunidad amonhen deadmarshes hobbit1 hobbit4 mago1)
(viajarcomunidad deadmarshes minasmorgul hobbit1 hobbit4 mago1)
(viajarcomunidad minasmorgul orodruin hobbit1 hobbit4 mago1)
(destruiranillo hobbit1 orodruin)
; cost = 14 (unit cost)
```

(b) 2H + 1M

```
(viajar mago1 rivendell bree)
(viajar hobbit1 hobbiton bree)
(viajar hobbit2 hobbiton bree)
(formarcomunidad hobbit1 hobbit2 hobbit4 mago1 bree)
(viajarcomunidad bree rivendell hobbit1 hobbit2 hobbit4 mago1)
(recogerobjeto hobbit1 rivendell anillo)
(viajarcomunidad rivendell moria hobbit1 hobbit2 hobbit4 mago1)
(recogerobjeto hobbit1 moria mithril)
(viajarcomunidad moria lothlorien hobbit1 hobbit2 hobbit4 mago1)
(recogerobjeto hobbit1 lothlorien espada)
(viajarcomunidad lothlorien amonhen hobbit1 hobbit2 hobbit4 mago1)
(viajarcomunidad amonhen deadmarshes hobbit1 hobbit2 hobbit4 mago1)
(viajarcomunidad deadmarshes minasmorgul hobbit1 hobbit2 hobbit4 mago1)
(viajarcomunidad minasmorgul orodruin hobbit1 hobbit2 hobbit4 mago1)
(destruiranillo hobbit1 orodruin)
; cost = 15 (unit cost)
```

(c) 3H + 1M

```
(viajar elfo1 lothlorien moria)
(viajar hobbit1 hobbiton bree)
(viajar hobbit2 hobbiton bree)
(viajar hobbit1 bree rivendell)
(viajar hobbit2 bree rivendell)
(viajar hobbit4 bree rivendell)
(viajar elfo1 moria rivendell)
(formarcomunidad hobbit1 hobbit2 hobbit4 mago1 elfo1 rivendell)
(recogerobjeto hobbit1 rivendell anillo)
(viajarcomunidad rivendell moria hobbit1 hobbit2 hobbit4 mago1 elfo1)
(recogerobjeto hobbit1 moria mithril)
(viajarcomunidad moria lothlorien hobbit1 hobbit2 hobbit4 mago1 elfo1)
(recogerobjeto hobbit1 lothlorien espada)
(viajarcomunidad lothlorien amonhen hobbit1 hobbit2 hobbit4 mago1 elfo1)
(viajarcomunidad amonhen deadmarshes hobbit1 hobbit2 hobbit4 mago1 elfo1)
(viajarcomunidad deadmarshes minasmorgul hobbit1 hobbit2 hobbit4 mago1 elfo1)
(viajarcomunidad minasmorgul orodruin hobbit1 hobbit2 hobbit4 mago1 elfo1)
(destruiranillo hobbit1 orodruin)
; cost = 18 (unit cost)
```

(d) 3H + 1M + 1E

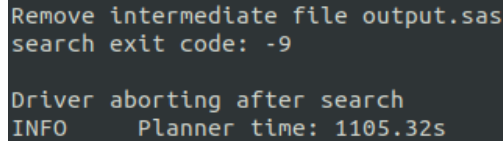
Figura 3: Planes encontrados para los distintos tipos de comunidades

Comentarios sobre la **complejidad computacional** de la planificación automática:

- Antes de nada, cabe mencionar qué es la **planificación automática**. Es una rama de la inteligencia artificial que busca generar una secuencia de acciones que lleve desde un estado inicial hasta un estado de meta. En estos problemas es muy importante el desacople de la lógica del dominio de la tarea concreta que queremos abordar. Se usa mucho en robótica, videojuegos y logística.
- En nuestro ejemplo, solo aumentamos la comunidad en un personaje cada vez y ello implica un aumento del plan solución en 1, 2, o 3 pasos como mucho. Sin embargo, el **tiempo crece exponencialmente** una barbaridad. Este problema no es nada escalable. Se abren demasiadas opciones a probar con cada nuevo integrante.
- En concreto, se trata de un problema **PESPACIO** y **NP-completo en tiempo**, como casi todos los problemas de la planificación automática. Dicho de otra forma, son problemas que requieren, en el peor de los casos, una cantidad de espacio que crece de forma polinomial con respecto al tamaño del problema; pero que toman un tiempo exponencial en probar todas las posibles combinaciones.

Ejercicio 4

No he llegado a obtener un plan para este ejercicio porque, debido a que **mi implementación es bastante ineficiente** y el planificador desiste de encontrar un plan tras agotar todos los recursos de memoria y tiempo disponibles. Este es el código de error que me proporcionó:



```
Remove intermediate file output.sas
search exit code: -9

Driver aborting after search
INFO      Planner time: 1105.32s
```

Figura 4: Error obtenido al ejecutar los archivos *dominio4.pddl* y *problema4.pddl* usando *seq-sat-lama-2011*

Había pensado en usar las funciones que expongo en el fragmento de código siguiente, pero me daban múltiples errores. Investigando el error, uno de los posibles fallos puede estar en que no se puede usar la función `increase` dentro de un `when`, si no que el `increase` debe estar en la “primera línea” de los efectos.

```
1 en problem:
2 ; inicializamos las funciones definidas en el dominio
3 (= (total-hobbits) 4)           ; tenemos 4 hobbits
4 (= (hobbits-produciendo) 0)     ; inicialmente hay 0 hobbits produciendo Alimento
5
6 en domain:
7 definirla como:
8 (:functions
9   (total-hobbits)           ; Numero total de hobbits
10  (hobbits-produciendo)     ; Hobbits actualmente produciendo
11 )
12
13 y sustituir esto:
14 ; cuando todos los hobbits esten produciendo alimento,
15 ; activamos el predicado (todosHobbitsProduciendo)
16 (when
17   (forall (?h - Personaje)
18     (imply (personajeEs ?h Hobbit)
19       (trabajandoEn ?h Alimento)
20     )
21   )
22   (todosHobbitsProduciendo)
23 )
24
25 por:
26 (when (esAlimento ?rec) (increase (hobbits-produciendo) 1))
27 (when (= (hobbits-produciendo) (total-hobbits)) (todosHobbitsProduciendo))
```

Listing 1: Intento fallido de optimizar el código proporcionado para el ejercicio 4

Ejercicio 5

En este problema, tomando como base el código del ejercicio 1, experimentamos con la idea de que **las acciones pueden tener un coste no unitario**. En concreto, queremos que la acción `Viajar` cueste 3 para ciertos caminos y siga valiendo 1 para el resto.

Eso lo he implementado añadiendo un nuevo predicado (`conectado3 ?a - Localizacion ?b - Localizacion`) al que ya teníamos (`conectado ?a - Localizacion ?b - Localizacion`). Entonces, en el fichero del problema inicializamos cada conexión con el predicado correspondiente según su coste (unitario o 3).

Además, ponemos una acción `Viajar` por cada coste de camino diferente que haya. En este caso, definimos `ViajarUnitario` para los caminos con coste unitario y `ViajarEspecial` para los de coste especial. La primera tendrá como precondition `conectado` y la segunda, `conectado3`, entre ambas ciudades `?origen` y `?destino`. Además, consideramos un desacople del problema respecto a la lógica del domino al definir la función (`camino-especial`), con la que indicamos el coste del camino especial (distinto de 1) y que se inicializa en el archivo del problema como (`= (camino-especial) 3`). Así, la acción `ViajarEspecial` sumará, dentro de los efectos, el valor de la función/variable numérica (`camino-especial`) y `ViajarUnitario` sumará 1.

Por cierto, la acción `ExtraerRecurso` también suma 1 al coste total. Es decir, todas las acciones suman 1 menos la del viaje especial, que suma lo que indique la función (`camino-especial`). Este recuento del coste total lo representamos con la función (`total-cost`), la cual es inicializada en el fichero del problema a 0 mediante (`= (total-cost) 0`).

Otra novedad de este problema es que el objetivo no es sólo que alguien esté trabajando en Madera como en el ejercicio 1, si no que ese plan se consiga en el menor coste según la función (`total-cost`). Esto lo indicamos añadiendo (`:metric minimize (total-cost)`) a continuación del `:goal`.

Por defecto, un planificador (enfocado a optimización, con heurística admisible) intenta minimizar el número de acciones de un plan, pero esto no nos interesa ahora. Entonces, al definir dicha métrica, estamos forzando al planificador a explorar nuevas soluciones para cumplir con el requisito, aunque no sean soluciones de pasos mínimos. En definitiva, en el ejercicio 5 **no queremos minimizar el número de acciones, si no su costo**.

```
1(viajarunitario enano1 tharbad bree)
2(viajarunitario enano1 bree rivendell)
3(viajarunitario enano1 rivendell highpass)
4(viajarunitario enano1 highpass mirkwood)
5(extraerrecurso enano1 mirkwood madera)
6; cost = 5 (general cost)|
```

Figura 5: Plan obtenido al ejecutar los archivos *dominio5.pddl* y *problema5.pddl* usando *seq-opt-lmcut*

Como vemos en la figura 5, el plan obtenido en el ejercicio 5 es de 5 acciones, mientras que ese mismo objetivo lo alcanzábamos en el ejercicio 1 con 4 acciones. La diferencia radica en que ahora, dicho camino más corto en nodos es más costoso en acciones respecto al proporcionado. Por eso, el planificador ha preferido extraer Madera en Mirkwood en lugar de en Fangorn.

Planteémonos ahora la siguiente pregunta conceptual. Imaginemos que tenemos un planificador que no nos asegura optimalidad en el número de acciones del plan. Es decir, devolverá un plan válido, pero no necesariamente óptimo. ¿De qué modo podríamos implementar un método que garantice, o al menos nos aproxime a, que el plan devuelto sea óptimo (en número de pasos)?

Podemos añadir en el archivo del problema, debajo del `:goal`, la métrica (`:metric minimize (total-cost)`) e incorporar en todas las acciones definidas el efecto (`increase (total-cost) 1`). Así, todas las acciones valen lo mismo (podría ser cualquier otro número positivo). De esta forma, al ejecutar el planner estaríamos haciendo que se busque el objetivo en el número mínimo de pasos, que coincide con el número mínimo de acciones ejecutadas, pues todas ellas valen lo mismo. De este modo, en caso de encontrar una solución válida, en el guion se afirma que se encuentra, será la óptima siempre.