Departamento de Informática
Universidad Carlos III de Madrid

Ingeniería Informática

Inteligencia Artificial

Prueba de evaluación

Normas generales del examen

- El tiempo para realizar el examen es de **2 horas**
- No se responderá a ninguna pregunta sobre el examen
- Si se sale del aula, no se podrá volver a entrar durante el examen
- No se puede presentar el examen escrito a lápiz

Problema 1. (2 puntos)

Dados una cantidad arbitraria de bloques, N, cada uno de ellos con una altura diferente, n_i, se desean construir **dos pilas** cuya altura sea lo más parecida posible. Por ejemplo, si hubiera cuatro bloques ($N = 4$) cuyas alturas fueran $n_1 = 2$, $n_2 = 3$, $n_3 = 4$ y $n_4 = 5$, sería posible organizar una pila con los bloques 1 y 4, con una altura igual a $n_1 + n_4 = 2 + 5 = 7$ unidades, de modo que la segunda pila contendría los bloques 2 y 3, con una altura también igual a $n_2 + n_3 = 3 + 4 = 7$ unidades.

Se pide:

1. **(0,5 puntos)** Definir formalmente el espacio de estados
2. **(0,5 puntos)** Calcular el tamaño del espacio de estados para cualquier número de bloques, N
3. **(0,25 puntos)** ¿Es posible resolver este problema *óptimamente* con una estrategia del primero en profundidad? Razona tu respuesta
4. **(0,25 puntos)** ¿Es posible resolver este problema *óptimamente* con una estrategia del primero en amplitud? Razona tu respuesta
5. **(0,25 puntos)** ¿Es posible resolver este problema óptimamente con un algoritmo de búsqueda bidireccional? Razona tu respuesta
6. **(0,25 puntos)** Si en vez de construir dos pilas de bloques, se pidiera organizar una cantidad cualquiera (pero estrictamente menor que el número de bloques, N), ¿es preciso cambiar alguna de las consideraciones de los tres últimos apartados?

Problema 2. (3 puntos)

Una agencia de viajes ofrece un servicio inteligente de organización de viajes a través de Internet. Para ello, la agencia solicita las localizaciones de origen y destino y utiliza un algoritmo heurístico para determinar la mejor ruta entre ambos puntos usando, para ello, cualquiera de los medios de transporte que tiene a su disposición. Los medios de transporte incluyen aviones, trenes, barcos, coches de alquiler, etc. Por supuesto, las rutas pueden contener una cantidad cualquiera de puntos intermedios donde es posible, entonces, cambiar de medio de transporte.

Teniendo en cuenta que cada uno de los medios de transporte tiene una velocidad y coste diferente, y que los usuarios pueden solicitar o bien minimizar el tiempo de tránsito, o bien el coste total del viaje, se pide:

2. **(0,5 puntos)** ¿Qué algoritmo de búsqueda de fuerza bruta puede usarse para resolver *óptimamente* este problema?

3. **(0,5 puntos)** Definir una función heurística, $h(n)$, que sea *admisible* e *informada*, para los casos en los que el usuario desea minimizar el tiempo de viaje entre el inicio y destino final

4. **(0,5 puntos)** Definir una función heurística, $h(n)$, que sea *admisible* e *informada*, para los casos en los que el usuario decida minimizar el coste total del viaje. Asumir, en este caso, que el coste de un trayecto es siempre proporcional a la velocidad del medio de transporte —de modo que los más rápidos son los más caros

5. **(0,5 puntos)** Definir una función heurística, $h(n)$, que sea *admisible* e *informada*, para los casos en los que el usuario decida minimizar el coste total del viaje. Asumir, en este caso, que no tiene por qué haber ninguna relación entre la velocidad del medio de transporte y su coste

Departamento de Informática
Universidad Carlos III de Madrid

Ingeniería Informática

Inteligencia Artificial

Prueba de evaluación

Normas generales del examen

- El tiempo para realizar el examen es de **1.5 horas**
- No se responderá a ninguna pregunta sobre el examen
- Si se sale del aula, no se podrá volver a entrar durante el examen
- No se puede presentar el examen escrito a lápiz

Problema 3. (5 puntos)

Un robot se encuentra en una mina y tiene el objetivo de coger oro. La mina está organizada como un grafo, en el que cada nodo se asume que es una celda, y hay celdas que se conectan con otras celdas. Cada celda puede estar libre, tener roca frágil o tener roca dura. El oro se encuentra en una celda libre o con roca frágil. En una celda libre el robot se puede encontrar una caja de bombas o un cañón láser (ambos inagotables). El cañón láser puede destruir rocas frágiles y duras, pero las bombas sólo pueden destruir rocas frágiles. Sin embargo, el cañón también destruirá el oro si se usa sobre una celda en la que hay oro, por lo que no se debería disparar sobre celdas con oro. Las bombas no destruyen el oro. El robot sólo puede sujetar en cada momento o el láser, o la caja de bombas, pero no los dos, y sólo puede cogerlos si se encuentra en la misma celda. El robot sólo puede moverse a celdas adyacentes a la que se encuentre y que estén libres. Para destruir una roca, el robot debe estar en una celda adyacente a la de la roca. Para coger el oro, debe estar en la misma celda.

1. (**1 punto**) Formalizar el enunciado como un dominio de planificación.
2. (**0,5 puntos**) Formalizar el problema correspondiente con el mostrado en la Figura 1.
3. (**1,5 puntos**) Obtener un plan que resuelva el problema anterior, y justificar la solución obtenida. Para justificar el plan obtenido, es necesario elegir un planificador y desarrollar informalmente el árbol de planificación (describir el proceso por el que se llegaria a esa solución).
4. (**0,5 puntos**) ¿Es el plan obtenido óptimo? ¿Garantiza el planificador utilizado que la solución sea óptima?
5. (**1,5 puntos**) ¿Cuál es el tamaño del espacio de estados para el problema y dominio dados?

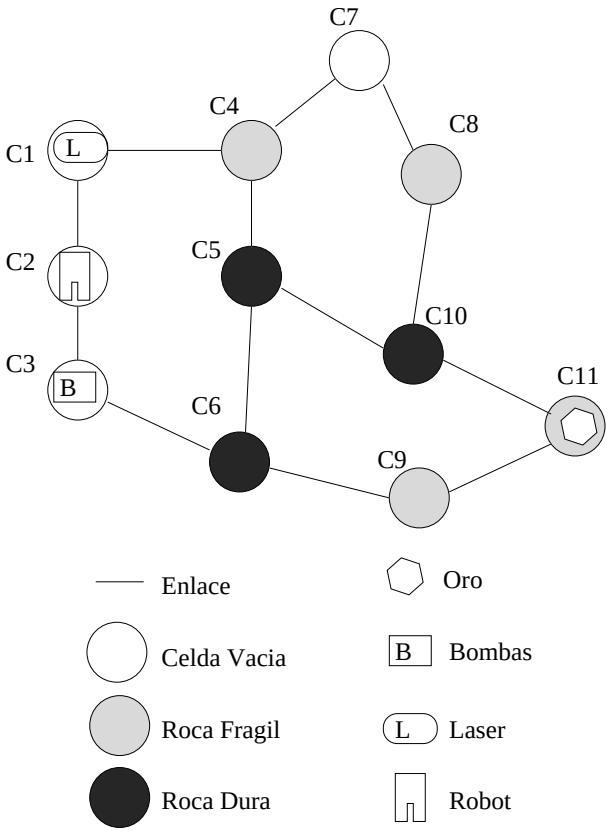

Figura 1: Configuración inicial del problema

Solución al problema 1

Este problema está dedicado al estudio de los modelos básicos de búsqueda no informada, o de fuerza bruta. Algunos de los apartados requerían razonar sobre algunos conceptos vistos en clase y algunas respuestas no eran tan obvias como podría parecer a primera vista.

Apartado 1.1

La formulación del espacio de estados consiste en definir, separadamente, los *estados* y *operadores*. Mientras que los estados sirven para distinguir configuraciones plausibles del problema (entre los que se distinguen el estado inicial y uno o más estados finales), los operadores son el modo más común para representar las transiciones entre éllos.

La definición de los estados es realmente muy fácil. Basta con advertir que cada bloque está, o bien asignado a una pila (ya sea la número 1, o la número 2), o está sin asignar. Si el último caso se describe con el número 0, el estado de todos los bloques podría especificarse como un vector (que llamaremos `asignación`) con tantos elementos como bloques haya, donde la posición i-ésima vale 0, 1 ó 2 según esté sin asignar, asignado a la primera pila, o a la segunda, respectivamente. Por ejemplo, el estado inicial típico de un problema con cuatro bloques sería `asignacion` $= (0, 0, 0, 0)$. De la misma manera, el estado final presentado en el enunciado se representaría como `asignacion` $= (1, 2, 2, 1)$. Además, por supuesto, de cada bloque es preciso mantener su altura. Esto podría hacerse en un segundo vector (que ahora llamaremos `altura-bloque`), cuya componente i-ésima es la altura del bloque i-ésimo. Por ejemplo, el caso de los cuatro bloques discutido en el enunciado se representaría como $\texttt{altura} - \texttt{bloque} = (2, 3, 4, 5)$. Por último, para calcular la altura de una pila, en vez de sumar las alturas de todos los bloques cuyo vector `asignación` sea precisamente la pila deseada, es más eficiente mantener en otro vector, `altura-pila`, la altura de cada pila en una componente separada.

La definición de los operadores es también trivial. La única acción relevante en este problema consiste en `asignar` un bloque b a una pila p[1].

`Asignar (b,p)`:

SI $\texttt{asignacion}[b] = 0 \wedge$ *(comprobación de que el bloque no está asignado)*
ENTONCES $\texttt{altura} - \texttt{pila}[p]$ += $\texttt{altura} - \texttt{bloque}[b]$ *(actualización de la altura de la pila)*

Apartado 1.2

La definición de los estados mostrado en el apartado anterior, sugiere que para un mismo problema, habrá tantos estados como asignaciones diferentes de bloques a pilas. Puesto que cada bloque puede estar en uno de tres estados de asignación diferente (0, 1 ó 2), y hay hasta N bloques, el espacio de estados es necesariamente 3^N.

Nótese que en este problema, el número de soluciones que pueden generarse es también muy alto. Puesto que cualquier solución consiste en la asignación completa a alguna pila de los N bloques (esto es, que cada componente tome valores exclusivamente iguales a 1 ó 2), el número de soluciones es 2^N.

Apartado 1.3

¡Por supuesto que se puede! En este problema en particular no es posible caer en ramas infinitas, de modo que no hay ninguna necesidad de establecer valores de profundidad máxima como se hace habitualmente para garantizar la *consistencia* del algoritmo. El motivo es que en cada nivel del árbol de búsqueda se asigna un bloque nuevo a cada pila, por lo que el árbol de búsqueda desarrollado en profundidad jamás puede exceder N niveles de profundidad, donde N es el número de bloques considerado.

Para encontrar la solución óptima, basta únicamente con mantener en una variable auxiliar la solución con la menor diferencia de altura entre las pilas encontrada hasta ese momento. Si inicialmente se inicializa esta variable a $+\infty$, y se actualiza cada vez que se llega a una solución nueva con una diferencia menor que la obtenida hasta ese momento, necesariamente el algoritmo acabará con la solución óptima.

[1] ¡Pero nunca desapilarlas! Si se programaran simultaneamente operadores para apilar y desapilar bloques entonces cualquier algoritmo de búsqueda podría caer en bucles infinitos.

Efectivamente. El algoritmo del primero en amplitud también serviría para verificar todas las 2^N soluciones diferentes que, como en el caso anterior, se generarían a profundidad N. Como antes, si se guarda en una variable auxiliar la mejor solución encontrada hasta ese momento, el algoritmo necesariamente acabará con la solución óptima.

Nótese, sin embargo, que este algoritmo tiene una ocupación de memoria exponencial, meintras que el algoritmo del primero en profundidad del apartado anterior no tiene esos problemas y servirá, también, para encontrar la mejor solución.

Apartado 1.5

No, no es posible, puesto que para poder aplicar la búsqueda bidireccional es preciso disponer de un estado final. ¡Precisamente un estado final es lo que se buscaba en este problema! de modo que difícilmente se podría hacer la búsqueda hacia atrás.

Apartado 1.6

No, en absoluto. Todas las consideraciones anteriores se refieren al dominio de un modo muy general y, por lo tanto, son independientes del número de pilas consideradas.

Solución al problema 2

Este problema está dedicado, fundamentalmente, al reconocimiento de las clases más típicas de funciones heurísticas: aquellas basadas en la distancia euclídea u otras más generales, basadas en cálculos hechos sobre tablas arbitrarias[2].

Apartado 2.1

La formalización del espacio de estados se hace describiendo, por una parte los estados y, por la otra, los operadores que sirven para transitar entre ellos. No forma parte de la formalización del espacio de estados la descripción de un estado inicial y final que, en su lugar, sirven para describir un problema particular.

En primer lugar, puede distinguirse un `viaje`, cuyos atributos será el `origen` y `destino`, para los que se calculará un `coste` y que tendrá una `distancia` final:

VIAJE			
es-un:			
Atributo	**Posibles valores/Valor**	**Valor omisión**	**Descripción**
origen	#LOCALIZACIÓN		Origen del viaje
destino	#LOCALIZACIÓN		Destino del viaje
ruta	VECTOR #LOCALIZACIÓN		Ruta completa
distancia	NÚMERO		Distancia final
tiempo	NÚMERO		Tiempo de viaje
coste	NÚMERO		Precio final

Como puede verse, un viaje consiste en una secuencia de localizaciones (cada una de las cuales se representa con instancias diferentes del marco clase `localización`, a continuación), que empiezan a partir del `origen` y que se almacenan en el atributo `ruta` del marco clase `viaje`. Cada una de las localizaciones puede describirse, simplemente, con un marco como el siguiente:

LOCALIZACIÓN			
es-un:			
Atributo	**Posibles valores/Valor**	**Valor omisión**	**Descripción**
nombre	SÍMBOLO		Nombre de la localización
transporte	VECTOR #MEDIO-TRANSPORTE		Transporte disponible

[2] El tercer y último tipo de función heurística típica sería la distancia de Manhattan.

cuales se describe a continuación como sigue:

MEDIO-TRANSPORTE			
es-un:			
Atributo	**Posibles valores/Valor**	**Valor omisión**	**Descripción**
identificador	SÍMBOLO		Identificador del transporte
origen	#LOCALIZACIÓN		Localización inicial
destino	#LOCALIZACIÓN		Localización final
distancia	NÚMERO		Distancia entre las localizaciones
tiempo	NÚMERO		Tiempo que tarda
coste	NÚMERO		Coste
velocidad	NÚMERO		Velocidad

Como hay tantas instancias de medios de transporte como destinos haya desde un mismo origen, este diseño sirve para representar y acceder toda la red de transportes de la compañía. Por supuesto, estos medios de transporte podrían *especializarse* después en aviones, coches, trenes, etc. Nótese que se ha definidio un atributo de `velocidad` que será útil para el diseño de las funciones heurísticas en los siguientes apartados.

Por otra parte, hay un único operador, que es `Desplazar` un viajero desde una localización inicial hasta otra final, en un determinado medio de transporte:

`Desplazar (inicio, fin, medio)`:

SI $\exists v \in$ `VIAJE` $\wedge$ *(obtener la instancia de viaje)*
  $\exists t \in$ `MEDIO-TRANSPORTE` $\wedge$ *(obtener la instancia de transporte)*
  (t.origen=`inicio`) $\wedge$ (t.destino=`fin`) *(comprobación del trayecto)*
ENTONCES v.ruta += `fin` $\wedge$ *(actualización de la ruta)*
  v.coste += t.coste $\wedge$ *(actualización del coste)*
  v.distancia += t.distancia $\wedge$ *(actualización de la distancia)*
  v.tiempo += t.tiempo $\wedge$ *(así como del tiempo)*

El coste del operador no se ha especificado explícitamente y es que depende del parámetro que deba tenerse en cuenta en cada caso —unas veces el tiempo y otras veces el coste, como se verá a continuación.

Apartado 2.2

Puesto que se trata de un problema de optimización con costes no unitarios, la mejor alternativa sería un algoritmo `de ramificación y acotación en profundidad`.

Apartado 2.3

Obviamente, el tiempo depende de la distancia a recorrer desde un nodo (o localización cualquiera) n hasta el nodo meta, t. Por lo tanto, si se relaja el hecho de que para llegar hasta la localización final es preciso transitar por las localizaciones contempladas por la agencia de viajes y que, en su lugar, es posible viajar en línea recta, resulta la distancia euclídea o aérea entre dos puntos, n y t. Sin embargo, para obtener una estimación admisible (esto es, conservadora que no sobreestime el esfuerzo para llegar hasta la meta), es preciso dividir esa distancia por el medio de transporte más rápido que haya disponible en el nodo n.

Por lo tanto, la función heurística sugerida es:

$$h_1(n,t) = \frac{d_E(n,t)}{v_{max}(n)}$$

donde $d_E(n,t)$ es, simplemente, la distancia euclídea entre las localizaciones n y t y $v_{max}(n)$ es la velocidad del medio de trasnporte más veloz disponible en n.

Apartado 2.4

Obviamente, si el coste es proporcional a la velocidad de los medios de transporte, y estos a la estimación de la distancia (como se ha visto en el apartado anterior), entonces es suficiente con usar la misma función heurística del apartado anterior. Ahora bien, puesto que en este caso se minimiza el coste, la función heurística debe estimar el coste y, por ello, debe multiplicarse por el coste del medio de transporte más barato disponible en n:

$$h_2(n,t) = h_1(n,t) \times c_{min}(n)$$

donde $c_{min}(n)$ es el medio de transporte más barato disponible en el nodo n.

Apartado 2.5

En este último caso no hay mucho que pueda hacerse puesto que se supone que ahora los costes están distribuidos arbitrariamente. En este caso, la única relajación factible consiste en asumir que se puede llegar en un único salto hasta el nodo final o meta y que, además, esto puede hacerse con el medio de transporte más barato disponible en el nodo origen, n:

$$h_3(n,t) = c_{min}(n)$$

Solución al problema 3

Apartado 3.1

Para describir el dominio, es necesario describir los tipos de los objetos, los predicados y los operadores del dominio. En este caso, tenemos un único objeto LOC que hace referencia a las celdas. En cuanto a los predicados, tenemos los siguientes predicados:

(robot-at ?x - LOC) : indica que el robot está en la posición $?x$

(bomb-at ?x - LOC) : indica que el robot está en la posición $?x$

(laser-at ?x - LOC) : indica que el láser está en la posición $?x$

(soft-rock-at ?x - LOC) : indica que hay roca frágil en la posición $?x$

(hard-rock-at ?x - LOC) : indica que hay roca dura en la posición $?x$

(gold-at ?x - LOC) : indica que hay oro en la posición $?x$

(arm-empty) : indica que el robot no está transportando nada

(holds-bomb) : indica que el robot está transportando la caja de bombas

(holds-laser) : indica que el robot está transportando el láser

(holds-laser) : indica que el robot está transportando oro

(clear ?x - LOC) : indica que la posición $?x$ está libre

(connected ?x - LOC ?y - LOC) : indica que se puede ir de la celda $?x$ a la celda $?y$

En cuanto a los operadores, son necesarios 4 operadores:

move (?x - LOC ?y - LOC) : permite desplazarse de la posición $?x$ a la posición $?y$

pickup-laser (?x - LOC) : permite recoger el láser de una posición $?x$

pickup-bomb (?x - LOC) : permite recoger una bomba de una posición $?x$

pick-gold (?x - LOC) : permite recoger oro de una posición $?x$

pick-up-laser(?x - LOC) : recoge el láser de la posición $?x$

pick-up-bomb(?x - LOC) : recoge las bombas de la posición $?x$

put-down-laser(?x - LOC) : deja el láser de la posición $?x$

put-down-bomb(?x - LOC) : deja las bombas de la posición $?x$

detonate-bomb(?x - LOC ?y - LOC) : detona una bomba en posición $?y$ desde la posición $?x$

La descripción completa del dominio en el lenguaje PDDL se muestra a continuación:

```
(define (domain gold-miner-typed)
(:requirements :typing)
(:types LOC)

(:predicates
(robot-at ?x - LOC)
(bomb-at ?x - LOC )
(laser-at ?x - LOC)
(soft-rock-at ?x - LOC)
(hard-rock-at ?x - LOC)
(gold-at ?x - LOC)
(arm-empty)
(holds-bomb)
                (holds-laser)
(holds-gold)
(clear ?x - LOC)
(connected ?x - LOC ?y - LOC)
)


(:action move
  :parameters (?x - LOC ?y - LOC)
  :precondition (and (robot-at ?x) (connected ?x ?y) (clear ?y))
  :effect (and (robot-at ?y) (not (robot-at ?x)))
)

(:action pickup-laser
  :parameters (?x - LOC)
  :precondition (and (robot-at ?x) (laser-at ?x) (arm-empty))
  :effect (and (not (arm-empty)) (holds-laser) (not (laser-at ?x)) )
)


(:action pickup-bomb
  :parameters (?x - LOC)
  :precondition (and (robot-at ?x) (bomb-at ?x) (arm-empty))
  :effect (and (not (arm-empty)) (holds-bomb) (not (bomb-at ?x)) )
)

(:action putdown-laser
  :parameters (?x - LOC)
  :precondition (and (robot-at ?x) (holds-laser))
  :effect (and (arm-empty) (not (holds-laser)) (laser-at ?x))
)

(:action putdown-bomb
  :parameters (?x - LOC)
  :precondition (and (robot-at ?x) (holds-bomb))
  :effect (and (arm-empty) (not (holds-bomb)) (bomb-at ?x))
)


(:action detonate-bomb
  :parameters (?x - LOC ?y - LOC)
```

```
                    (connected ?x ?y) (soft-rock-at ?y))
  :effect (and (not (holds-bomb)) (arm-empty) (clear ?y) (not (soft-rock-at ?y)))
)

(:action fire-laser
  :parameters (?x - LOC ?y - LOC)
  :precondition (and (robot-at ?x) (holds-laser)
                    (connected ?x ?y))
  :effect (and (clear ?y) (not (soft-rock-at ?y)) (not (gold-at ?y))
               (not (hard-rock-at ?y)))
)

(:action pick-gold
  :parameters (?x - LOC)
  :precondition (and (robot-at ?x) (arm-empty) (gold-at ?x))
  :effect (and (not (arm-empty)) (holds-gold))
)
)
```

Cabe destacar que otras represnetaciones son posibles. Por ejemplo, podríamos haber definido un único predicado (holds ?x), que definiera qué objeto está sujetando el robot. Eso requeriría definir las bombas, el láser y el oro como objetos, para poder usarlos en los predicados. El resto de la formulación debería cambiarse en función de esto, y habría que variar los operadores.

Apartado 3.2

Para describir el problema, sólo hay que tener en cuenta:

- Hay que definir todas las celdas como instancias del tipo LOC
- Hay que definir todas las conexiones entre las distintas celdas, en indicar qué hay en cada una de ellas.

El problema queda formalizado del siguiente modo:

```
(define (problem gold-miner-bootstrap-3x3-01-typed)
(:domain gold-miner-typed)
(:requirements :typing)
(:objects
        c1 c2 c3 c4 c5 c6
        c7 c8 c9 c10 c11  - LOC
)
(:init
  (arm-empty)
  (connected c1 c2) (connected c2 c1)
  (connected c1 c4) (connected c4 c1)
  (connected c2 c3) (connected c3 c2)
  (connected c3 c6) (connected c6 c3)
  (connected c4 c5) (connected c5 c4)
  (connected c4 c7) (connected c7 c4)
  (connected c5 c6) (connected c6 c5)
  (connected c5 c10) (connected c10 c5)
  (connected c6 c9) (connected c9 c6)
  (connected c7 c8) (connected c8 c7)
  (connected c8 c10) (connected c10 c8)
  (connected c9 c11) (connected c11 c9)
  (connected c10 c11)  (connected c11 c10)
```

```
  (robot-at c2)
  (bomb-at c3)
  (soft-rock-at c4)
  (hard-rock-at c5)
  (hard-rock-at c6)
  (clear c7)
  (soft-rock-at c8)
  (soft-rock-at c9)
  (hard-rock-at c10)
  (soft-rock-at c11)
  (gold-at c11)
)
(:goal
  (holds-gold)
)
)
```

Apartado 3.3

La única meta que existe es ($holds - gold$). Para cumplir esta meta, sólo existe un operador, que es *pick-gold* $?x$. Prodigy instanciaría este operador con todos los *bindings* posibles, que son todos los objetos de clase LOC. Supongamos que, de todos ellos, Prodigy seleccionara la instanciación *(pick-gold c1)*. Entre las precondiciones de este operador, está la precondición *(gold-at c1)*, que es imposible de satisfacer, puesto que no hay ningún operador que tenga dicho añadido en sus efectos. Eso ocurriría para todos los operadores instanciados, hasta que eligiera *(gold-at c11)*. Para ejecutar ese operador, se deben cumplir *(robot-at c11) (arm-empty) (gold-at c11)*, de las que en el estado inicial únicamente no se cumple *(robot-at c11)*. Por tanto, Prodigy seleccionaría esta nueva meta para trabajar en ella. Para conseguir *(robot-at c11)*, el único operador útil es *move ?x ?y*. La variable $?y$ se instanciaría con *c11*, pero $?x$ podría tomar cualquier valor, desde *c1* hasta *c11*. Si se sigue el razonamiento, un posible plan sería el siguiente:

```
(pick-gold)
(putdown-laser)
(move c9 c11)
(detonate-bomb c9 c11)
(move c6 c9)
(detonate-bomb c6 c9)
(move c3 c6)
(pick-up-bomb c6)
(put-down-laser c6)
(fire-laser c3 c6)
(move c2 c3)
(move c1 c2)
(pick-up-laser)
(move c2 c1)
```

Apartado 3.4

El plan obtenido es óptimo. Sin embargo, Prodigy es un planificador no óptimo, es decir, que no garantiza que los planes generados sean óptimos. Por tanto, se puede decir que el haber obtenido un plan óptimo ha sido casual.

Apartado 3.5

En primer lugar, habría que analizar cuántos estados surgen de las posibles rocas que puede haber en cada celda. Cada celda sólo puede estar libre, con roca frágil, y con roca Dura. La que están libres permanecerán siempre así, mientras que las otros puede que estén así, o puede que se destruyen la roca, pasen a estar libres. Por tanto, hay 4 celdas que siempre están libres, y 7 que estarán libres o con roca, en función de si destruyen la roca

cualquier combinación de las anteriores, o nada. Básicamente, eso es equivalente a, dadas las celdas libres, cuántas combinaciones de colocar el robot, el láser y la bomba hay. Si asumimos que hay estados en los que las 11 celdas podrían estar libres, eso da una cota superior de 11^3. Se dice que es una cota superior, porque este valor asume que todas las celdas están libres, pero esto no es verdad siempre, puesto que inicialmente hay rocas. Por tanto, la cota superior del tamaño del espacio de estados es $(2^7) \times (11^3) = 170368$ posibles estados. Hay que tener en cuenta que, en verdad, el número de estados posible en este problema es menor, puesto que se han contabilizado estados que nunca se podrían llegar a dar por ser imposibles de cumplir. Por ejemplo, no se podría dar un estado que resultara de cambiar en el estado inicial que *c8* pase a estar libre, puesto que no hay ningún plan capaz de transitar desde el estado inicial a ese hipotético estado (para destruir la roca de *c8*, antes deberíamos romper otras rocas).