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. (5 puntos)

¡Fernando Alonso desea ganar el próximo premio de Fórmula 1! Y para ello, ha decidido contar con la ayuda de los alumnos de la Universidad Carlos III de Madrid, para crear un sistema de simulación inteligente de carreras, tanto en el modo contrarreloj como en carrera individual contra otro piloto. En una primera aproximación, se supone que el circuito de carreras está constituido por celdas del mismo tamaño entre las que se mueven los vehículos si éstas están desocupadas, o contra las que pueden estrellarse, si aparecen rayadas en negro. Además, con el propósito de tener en cuenta el efecto de la aceleración sobre los vehículos, se considera que un coche puede pasar de las coordenadas (x_0, y_0) a una cualquiera de las posiciones del cuadrado inscrito entre las coordenadas $(x_r, y_r) = (x_0+d_x-2, y_0+d_y-2)$ y $(x_R, y_R) = (x_0+d_x+2, y_0+d_y+2)$, donde (d_x, d_y) es el vector de desplazamiento que llevó al monoplaza hasta la posición (x_0, y_0). Obviamente, $(d_x, d_y) = (0, 0)$ al inicio de la carrera para cualquier coche.

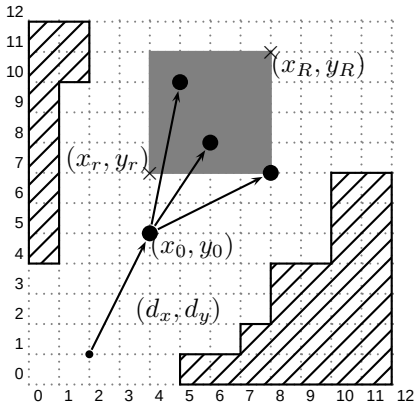


Figura 1: Modelo discreto de desplazamiento de coches

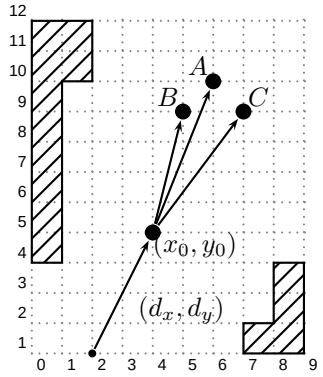
Para la aplicación contrarreloj, se desea encontrar la mejor secuencia de movimientos que lleven al monoplaza desde la línea de salida hasta la línea de meta **en el menor tiempo posible** (esto es, movimientos), en un circuito cualquiera descrito con celdas como las mencionadas anteriormente. Para ello, se pide:

1. **(0,5 puntos)** Formalizar el espacio de estados con el uso de marcos
2. **(0,5 puntos)** Modelizar los operadores del dominio

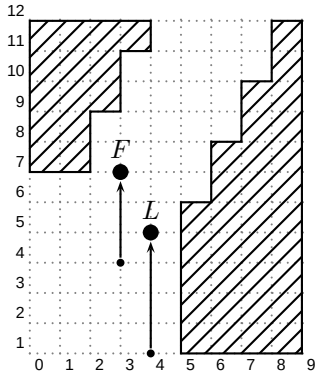
4. **(0,5 puntos)** Sin considerar los obstáculos, ¿cuáles son el valor mínimo y máximo del factor de ramificación en el árbol de búsqueda desarrollado a una profundidad arbitraria d?

5. **(1 punto)** Definir una función heurística $h(n)$ que sea *admisible* e *informada* explicando claramente su construcción

Para la construcción del sistema de simulación de carreras contra un único piloto, se asume que ambos jugadores toman decisiones alternadamente, decidiendo en cada turno cuál es la posición que ocuparán. Por otra parte, el cálculo de colisiones se ha simplificado notablemente, considerando que ambos monoplazas chocan sólo si después de que uno se mueve, llega a la misma posición que ocupa el otro. Por último, el conjunto de posiciones siguientes para esta segunda aplicación se ha restringido significativamente, y ahora sólo puede pasarse a una cualquiera de las tres posiciones siguientes: $A(x_0 + d_x + 0, y_0 + d_y + 1)$, $B(x_0 + d_x - 1, y_0 + d_y)$ y $C(x_0 + d_x + 1, y_0 + d_y)$, tal y como se muestra en la figura 2(a)



(a) Posiciones permitidas en el modo de carrera



(b) Fernando Alonso (F) seguido por Lewis Hamilton (L)

Figura 2: Definiciones para el modo de carrera contra otro piloto

Se pide:

6. **(1 punto)** Aplicar el algoritmo MINIMAX a profundidad 2 al caso de la figura 2(b) para Fernando Alonso, usando una función de evaluación que devuelve $-\infty$ si el vehículo de Fernando Alonso (señalado con una F) choca y el de Lewis Hamilton (etiquetado con L) no; $+\infty$ si choca el de Hamilton y no el de Fernando; 0 si ambos chocan y, por último, la diferencia en Y entre el coche de Fernando Alonso y el de Lewis Hamilton en cualquier otro caso.

   Indicar claramente el valor MINIMAX del nodo raíz, así como la secuencia elegida de movimientos. Interpreta el resultado, ¿qué significa?

7. **(1 punto)** Definir una función de evaluación más realista que la anterior explicando claramente su construcción

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 2. (5 puntos)

La matrioska o matrioska son unas muñecas tradicionales rusas creadas en 1890, cuya originalidad consiste en que se encuentran huecas por dentro, de tal manera que en su interior albergan una nueva muñeca, y ésta a su vez a otra, y ésta a su vez otra, en un número variable que puede ir desde cinco hasta el número que se desee, aunque es raro que pasen de veinte. Se caracterizan por ser multicolores, o por la presencia de elementos decorativos en la pintura tales como jarrones o recipientes sostenidos por las muñecas, pero en la práctica pueden identificarse por su tamaño.

Las matrioskas pueden verse como un juego. Las matrioskas pueden estar abiertas (el cuerpo separado de la cabeza), o cerradas. Una matrioska puede contener a otra matrioska. Los estados inicial y final pueden ser cualquier configuración en la que todas las matrioskas están: (i) o bien cerradas en la mesa y/o unas dentro de otras, (ii) o bien abiertas encima de la mesa. Por tanto, unas matrioskas que inicialmente tienen la configuración que se muestra en la Figura 2 no tendrían una configuración correcta, puesto que hay unas matrioskas abiertas dentro de otras. Una matrioska sólo se puede abrir si está encima de la mesa. Una matrioska cerrada que está encima de la mesa puede meterse en otra si la segunda está abierta, encima de la mesa y es más grande que la primera.

1. (**0,5 puntos**) Formalizar el enunciado como un dominio de planificación. Se valorará que la formalización dada sea fácilmente escalable a cualquier número de matrioskas.

2. (**0,5 puntos**) Formalizar un problema de 5 matrioskas. En el estado inicial, la más grande está cerrada y vacía encima de la mesa, y las demás están todas cerradas dentro de la de tamaño inmediatamente superior de la que le corresponde (excepto para la cuarta más grande que está encima de la mesa, puesto que la más grande está vacía). En su configuración final están todas cerradas y dentro de la de tamaño inmediatamente superior.

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 formalmente el árbol de planificación.

4. (**0,5 puntos**) ¿Es el plan obtenido óptimo? ¿Garantiza el planificador utilizado que la solución sea óptima?

5. (**0,5 puntos**) ¿Qué literales habría que añadir al estado inicial si agregamos al problema 2 matrioskas más?

6. (**1,5 puntos**) ¿Cuál es el tamaño del espacio de estados para n matrioskas, si asumimos que los estados inicial y meta pueden ser cualquiera (que cumpla las especificaciones del dominio)?

Figura 3: Configuración incorrecta de Matrioskas

Solución al problema 1

1. Los conceptos relevantes del enunciado son los relativos a los vehículos y los circuitos donde se desenvuelven las carreras.

   De una parte, la información más importante que debe mantenerse de los coches son tanto su posición $P(x, y)$, como su vector de dirección, $d(d_x, d_y)$. Puesto que ambos pueden almacenarse como un vector de dos componentes, lo más razonable es representar este concepto primero como sigue:

VECTOR			
es-un:			
Atributo	**Posibles valores/Valor**	**Valor omisión**	**Descripción**
x	entero		valor de abscisas
y	entero		valor de ordenadas

   de modo que ahora puede representarse eficientemente la información útil sobre los coches como sigue:

COCHE			
es-un:			
Atributo	**Posibles valores/Valor**	**Valor omisión**	**Descripción**
p	#VECTOR		posición del vehículo
d	#VECTOR		vector dirección del vehículo

   Por otra parte, el circuito podría representarse como la colección de casillas que pueden ser usadas por los vehículos:

CASILLA			
es-un:			
Atributo	**Posibles valores/Valor**	**Valor omisión**	**Descripción**
celda	#VECTOR		casilla del circuito

   Obsérvese que no hay ninguna necesidad de caracterizar las casillas como *rayadas* (o utilizables) o no, puesto que sólo se crearán instancias de aquellas que pueden ser usadas por los vehículos para permanecer en la carrera.

2. Resulta muy fácil advertir que hay un único operador en este problema, y que es el que se usa para llevar un coche `c` desde su posición actual (almacenada en el atributo `c.p`) hasta otra (x', y')[1].

   `Mover (c,x',y'):`

   SI $\exists v \in$ `COCHE` $\wedge$ *(obtener la instancia de coche)*
   $\exists c \in$ `CASILLA` $\wedge$ *(obtener la instancia de casilla)*
   $(v.p.\text{x}{=}p_x) \wedge (v.p.\text{y}{=}p_y) \wedge$ *(lectura del vector de posición)*
   $(v.d.\text{x}{=}d_x) \wedge (v.d.\text{y}{=}d_y) \wedge$ *(lectura del vector de dirección)*
   $(p_x + d_x - 2 \leq x') \wedge (p_x + d_x + 2 \geq x') \wedge$ *(comprobación del desplazamiento en X)*
   $(p_y + d_y - 2 \leq y') \wedge (p_y + d_y + 2 \geq y')$ *(comprobación del desplazamiento en Y)*
   ENTONCES $v.d.\text{x}{=}\ (v.p.\text{x}\text{-}x') \wedge$ *(actualización de la componente X de la dirección)*
   $v.d.\text{y}{=}\ (v.p.\text{y}\text{-}y') \wedge$ *(y de la componente Y)*
   $v.p.\text{x}{=}x' \wedge$ *(actualización de la posición en X)*
   $v.p.\text{y}{=}y' \wedge$ *(así como en Y)*
   $\text{k} = 1$ *(coste del operador)*

   Obsérvese que el coste del operador es siempre el mismo y, por lo tanto, éste es un problema de costes uniformes.

[1] En realidad, el operador tal y como se ha presentado, es algo simplista, puesto que no tiene en cuenta si en su movimiento desde su posición actual hasta (x', y') está cruzando o no por posiciones rayadas o ilegales, pero es suficiente para los propósitos del ejercicio.

primero en profundidad o amplitud y del primero en profundidad iterativa, ya sea ejecutados hacia delante, detrás o de modo bidireccional.

Puesto que en el último avance podría escogerse una cualquiera de las casillas del rectángulo que están más allá de la línea de meta, (¡de modo que existen varios estados finales!) se descarta la búsqueda hacia atrás y, con ella, cualquier forma de búsqueda bidireccional.

De entre los algoritmos que quedan, el algoritmo del primero en profundidad no es *completo* (y, por ello, menos aún *admisible*), de modo que nunca servirá para encontrar soluciones óptimas.

Por lo tanto, se recomienda:

Primero en amplitud: es un algoritmo completo (encuentra al menos una solución si ésta existe) y admisible (devuelve la solución óptima) cuando los costes son uniformes. Sin embargo, tiene un consumo de memoria exponencial. Considerando, además, que se están recomendando algoritmos de fuerza bruta, esta alternativa no serviría probablemente, ni siquiera para resolver problemas triviales.

Profundización iterativa: en todos los casos en los que el algoritmo de el primero en amplitud agota la memoria, es posible progresar iterativamente a profundidades incrementalmente mayores, de modo que se garantiza la admisibilidad u optimalidad de las soluciones encontradas. Si en la enumeración de alternativas a profundidades cada vez mayores se usa, además, el algoritmo del primero en amplitud, entonces el consumo de memoria sería lineal.

Probablemente el algoritmo de profundización iterativa tardaría mucho tiempo en resolver problemas, pero en la práctica resolvería muchos que el algoritmo de el primero en amplitud no puede.

4. En ausencia de obstáculos, lo más común será que el vehículo pueda desplazarse a una cualquiera de las posiciones del cuadrado inscrito entre las coordenadas $(x_r, y_r) = (x_0 + d_x - 2, y_0 + d_y - 2)$ y $(x_R, y_R) = (x_0 + d_x + 2, y_0 + d_y + 2)$ que están más allá de su posición actual. En total, hay hasta:

$$((x_0 + d_x + 2) - (x_0 + d_x - 2) + 1) \times ((y_0 + d_y + 2) - (y_0 + d_y - 2) + 1) = 5 \times 5 = 25$$

casillas alcanzables desde la posición actual[2]. Por supuesto, no es posible generar más descendientes que éstos después de cualquier expansión, de modo que el factor de ramificación máximo será $b_{max} = 25$.

Sin embargo, sí es posible generar menos descendientes: las veces que el cuadrado que inscribe las posiciones legales del desplazamiento ¡contiene a la posición actual del vehículo! Considerando únicamente el eje de abscisas X esta condición puede formularse como el sistema de ecuaciones:

$$\begin{cases} x_0 + d_x - 2 & \leq & x_0 \\ x_0 + d_x + 2 & \geq & x_0 \end{cases}$$

cuya solución son todos los valores de $d_x \in [-2, +2]$. Análogamente, la misma observación aplica a d_y. Por lo tanto, las veces que $d_x, d_y \in [-2, +2]$, el cuadrado resultante contendrá a la posición actual del vehículo y, por lo tanto, el movimiento se hará escogiendo entre alguna de las 25-1=24 posiciones restantes[3]. En ausencia de obstáculos no es factible generar menos descendientes que éstos y, por lo tanto, $b_{min} = 24$

5. Como de costumbre, se sugiere aplicar la técnica de *relajación de restricciones* para la obtención sistemática de funciones heurísticas. Aunque las restricciones del problema pueden presentarse de varias formas, lo más común es observarlas como elementos de condición del antecedente de los operadores. Por lo tanto, prestando atención al operador presentado en la sección 2, son restricciones del problema susceptibles de ser relajadas las siguientes:

   - La condición de que haya obstáculos en el circuito. Una vez que se relaja esta condición, ahora es posible mover el vehículo a cualquier posición cumpliendo o no con otras restricciones, susceptibles también de ser relajadas como sigue:

[2] Nótese que se ha sumado 1 en cada factor porque los extremos del cuadrado también son posiciones legales del desplazamiento

[3] Un ejemplo trivial en el que $d_x, d_y \in [-2, +2]$ es cuando el vehículo está aún en la línea de salida, en cuyo caso $-2 \leq d_x = d_y = 0 \leq +2$

$d_y - 2)$ y $(x_R, y_R) = (x_0 + d_x + 2, y_0 + d_y + 2)$. Si se elimina, entonces el coche podría desplazarse en cualquier momento hasta cualquier posición del circuito. En tal caso, el coste óptimo del problema relajado de esta manera es:

$$h_1 = 1$$

que es un valor constante y, por lo tanto, no informado, que convierte los algoritmos de búsqueda heurística en algoritmos de fuerza bruta.

- La *dirección* del vector de dirección (d_x, d_y). En este caso, se sugiere resolver óptimamente el problema que resulta de relajar los obstáculos del circuito y, además, la condición relativa a la dirección en la que debe calcularse el cuadrado que delimita las posiciones legales de avance del coche, pero preservando la *magnitud* (o *módulo*) del vector de dirección (d_x, d_y).
  Por lo tanto, si el coche debe llegar desde su posición original (x_0, y_0) hasta otra final (x_k, y_k) en k pasos progresando por posiciones intermedias (x_1, y_1), (x_2, y_2), ..., resulta obvio que lo más rápido es progresar *acelerando* el vehículo.
  Considérese primero el eje de abscisas, X. En este caso, el coche puede llegar desde x_0 hasta $x_1 = x_0 + d_x + 2$, de modo que en este punto la nueva componente del vector d_x (que se denotará como $d_x^{(1}$) será: $d_x^{(1} = x_1 - x_0 = d_x + 2$. Por lo tanto, el siguiente punto será $x_2 = x_1 + d_x^{(1} + 2 = x_1 + (d_x + 2) + 2 = (x_0 + d_x + 2) + (d_x + 2) + 2 = x_0 + 2(d_x + 3)$. Con este desplazamiento, la segunda componente del vector de desplazamiento, $d_x^{(2}$ será igual a $d_x^{(2} = x_2 - x_1 = d_x^{(1} + 2 = (d_x + 2) + 2 = d_x + 4$, de tal suerte que $x_3 = x_2 + d_x^{(2} + 2 = (x_0 + 2(d_x^{(0} + 3)) + (d_x + 4) + 2 = x_0 + 3(d_x + 4)$. Y así sucesivamente, hasta que finalmente algún $x_k = x_0 + k(d_x + k + 1)$ excede el valor X de la meta, x_f o, en otras palabras:

$$x_0 + k(d_x + k + 1) \geq x_f$$

que, tomando la igualdad y desarrollando el miembro izquierdo, puede expresarse como la ecuación de segundo grado siguiente:

$$k^2 + k(d_x + 1) - (x_f - x_0) = 0 \tag{1}$$

cuya raíz positiva menor es la estimación admisible del número de pasos, en las abscisas, que hacen falta para alcanzar la meta x_f:

$$h_2 = \textit{mínima raíz positiva de la ecuación (1)}$$

Sin embargo, debe advertirse también que el mismo razonamiento aplica a las ordenadas, de modo que es posible obtener la misma ecuación para las ordenadas con el mismo razonamiento:

$$k^2 + k(d_y + 1) - (y_f - y_0) = 0 \tag{2}$$

de donde resulta una nueva función heurística:

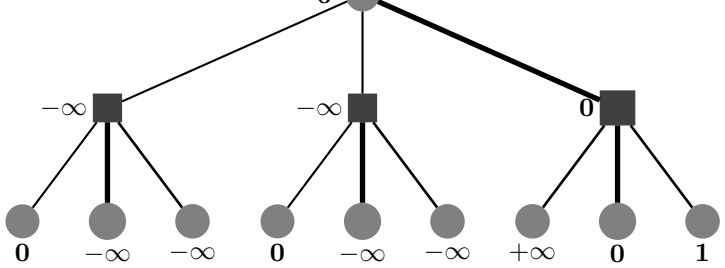
$$h_3 = \textit{mínima raíz positiva de la ecuación (2)}$$

Puesto que ahora se tienen dos funciones heurísticas diferentes (una para el progreso acelerando en X, y otra para la aceleración en Y), una función heurística que siempre será más informada que las dos anteriores ¡es precisamente el máximo de éllas!:

$$h_4 = \max\{h_2, h_3\}$$

- Una forma de obtener funciones heurísticas, mejor informadas aún que h_4, consiste en considerar los obstáculos en el rectángulo de posiciones legales alcanzables desde (x_0, y_0), calculando el valor h_4 para cada posición que no sea un obstáculo y tomando, finalmente, el mínimo de éllas.

6. El siguiente árbol MINIMAX representa las alternativas para el primer piloto, Fernando Alonso (con círculos, será el jugador MAX) y, a continuación, las alternativas para el segundo piloto, Lewis Hamilton —con cuadrados, el nodo MIN. En todos los casos, los nodos se han expandido de izquierda a derecha como se muestra en los casos B, A y C de la figura 2(a). Debajo de los nodos terminales del árbol de búsqueda desarrollado a profundidad 2 se muestra el valor de la función de evaluación en cada caso:



Los descendientes que son elegidos para dar la puntuación a cada nodo no terminal están marcados con un arco más grueso. Como puede verse, la *variante principal* del árbol resulta ser que Fernando Alonso moverá C, que es su única alternativa para evitar un choque, mientras que Lewis Hamilton elegirá entonces A, chocándose con Fernando Alonso, ¡aunque jugando también C habría evitado el choque! pero en ese caso, su contrincante le habría adelantado.

7. Obviamente, el resultado del apartado anterior es irracional. El motivo es que la función de evaluación no es razonable —ningún piloto *racional* arriesgaría su propia vida por ser adelantado en una curva.

   Por lo tanto, otras funciones de evaluación más razonables tendrán ahora en cuenta la posición de los coches. Si sólo tuvieran en cuenta la posición actual, podrían no darse cuenta que uno de ellos tiene una aceleración mayor que el otro. Por lo tanto, esta función de evaluación puede mejorarse significativamente comparando los vectores de dirección (sentido y magnitud) para comparar quien tiene más posibilidades de ir delante del otro en el siguiente turno. Este cómputo puede hacerse, sencillamente, comparando los cuadrados de posiciones legales a los que accede cada jugador:

$$f(x) = \frac{\textit{Área por delante de Alonso no solapada}}{25}$$

   donde el *área por delante de Alonso no solapada* es la cantidad del cuadrado de Alonso inscrito en $(x_r, y_r) = (x_0 + d_x - 2, y_0 + d_y - 2)$ y $(x_R, y_R) = (x_0 + d_x + 2, y_0 + d_y + 2)$ que está por delante del cuadrado de su oponente —esto es, que no se solapa con el de Hamilton y, de hecho, está por delante. Si Fernando Alonso no dispone de ninguna posición por delante de la de Lewis Hamilton, obtendrá 0, pero en el mejor de los casos podrá llegar a tener sus 25 casillas por delante de Hamilton y, con ello, su valor será 1. Por lo tanto $f(x) \in [0, 1]$

Solución al problema 2

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 *MATRIOSKA* que hacer referencia a las figuritas rusas. En cuanto a los predicados, tenemos los siguientes predicados:

   (closed $< m >$**)** : indica que la matrioska $< m >$ está cerrada

   (on-table $< m >$**)** : indica que la matrioska $< m >$ está encima de la mesa

   (bigger $< m2 > < m1 >$**)** : indica que la matrioska $< m2 >$ es mayor que la matrioska $< m1 >$

   (empty $< m >$**)** : indica que la matrioska $< m >$ no contiene ninguna otra matrioska

   (in $< m1 > < m2 >$**)** : indica que la matrioska $< m1 >$ está dentro de la matrioska $< m2 >$

   En cuanto a los operadores, son necesarios 4 operadores:

   (OPEN $< m >$**)** : permite abrir una matrioska que está cerrada y encima de la mesa. Como efecto, la matrioska pasa a estar abierta

   (CLOSE $< m >$**)** : permite cerrar una matrioska que está abierta y encima de la mesa. Como efecto, la matrioska pasa a estar cerrada

   (PUT-IN $< m1 > < m2 >$**)** : introduce la matrioska $< m1 >$, que debe estar cerrada y encima de la mesa, en la matrioska $< m2 >$, que debe estar abierta, vacía y encima de la mesa. Además, la matrioska $< m1 >$ debe ser menor que $< m2 >$

abierta y encima de la mesa. Obviamente, la matrioska $< m1 >$ debe estar inicialmente dentro de la matrioska $< m2 >$, de forma que $< m1 >$ pasa a estar encima de la mesa, y $< m2 >$ se queda vacía.

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

```
(create-problem-space 'matrioska :current t)

(ptype-of MATRIOSKA :top-type)


(OPERATOR  OPEN
  (params <m>)
  (preconds
    ((<m> MATRIOSKA))
    (and (closed <m>) (on-table <m>)))
  (effects
    ()
    ((del (closed <m>)))))

(OPERATOR CLOSE
  (params <m>)
  (preconds
    ((<m> MATRIOSKA))
    (and (~ (closed <m>)) (on-table <m>)))
  (effects
    ()
    ((add (closed <m>)))))


(OPERATOR PUT-IN
  (params <m1> <m2>)
  (preconds
    ((<m1> MATRIOSKA) (<m2> MATRIOSKA))
    (and (~ (closed <m2>)) (closed <m1>)
 (on-table <m2>) (on-table <m1>)
 (bigger <m2> <m1>) (empty <m2>)))
  (effects
   ()
   ((del (on-table <m1>))
    (add (in <m1> <m2>))
    (del (empty <m2>))
       )))


(OPERATOR  TAKE-OUT
  (params <m1> <m2>)
  (preconds
    ((<m1> MATRIOSKA) (<m2> MATRIOSKA))
    (and
     (~ (closed <m2>)) (in <m1> <m2>) (on-table <m2>)))
  (effects
   ()
   ((del (in <m1> <m2>))
   (add (empty <m2>)))))
```

2. Para describir un problema de 5 matrioskas, sólo hay que tener en cuenta:

- Hay que definir las relaciones de tamaño entre todas las matrioskas, una a una. Es decir, si la matrioska 3 es mayor que la 2, y la 2 es mayor que la 1, eel sistema no deducirá que la 3 es mayor que la 1. Por tanto, hay que definir la relación de orden entre todas las matrioskas.

El problema queda formalizado del siguiente modo:

```
(setf (current-problem)
      (create-problem
       (name prob1)
       (objects
(matri1 matri2 matri3 matri4 matri5 MATRIOSKA))
       (state
(and
 (on-table matri4)
 (on-table matri5)
 (in matri1 matri2)
 (in matri2 matri3)
 (in matri3 matri4)
 (empty matri1)
 (empty matri4)
 (empty matri5)
 (closed matri1)
 (closed matri2)
 (closed matri3)
 (closed matri4)
 (closed matri5)
 (bigger matri2 matri1)
 (bigger matri3 matri1)
 (bigger matri4 matri1)
 (bigger matri5 matri1)
 (bigger matri3 matri2)
 (bigger matri4 matri2)
 (bigger matri5 matri2)
 (bigger matri4 matri3)
 (bigger matri5 matri3)
 (bigger matri5 matri4)))
       (goal
(and
 (in matri1 matri2)
 (in matri2 matri3)
 (in matri3 matri4)
 (in matri4 matri5)
 (closed matri5)))))
```

Cabe destacar que en esta meta sólo se especifica como meta que esté cerrada la matrioska *matri*5. Dado que el resto de matrioskas están unas dentro de otras, por las características del dominio se cumplirá que todas están cerradas.

3. Al ejecutar Prodigy, se genera el siguiente árbol de búsqueda:

```
(done) 2 0
 *finish* 3 0
  (*finish*) 4 0
   (in matri4 matri5) 5 0
    put-in 6 0
     (put-in matri4 matri5) 7 0
      (~(closed matri5)) 8 0"
```

```
(open matri5) 10 0
 OPEN MATRI5  11 0
  PUT-IN MATRI4 MATRI5  12 2
   (closed matri5) 13 1
    close 14 1
     (close matri5) 15 1
      CLOSE MATRI5  16
```

Inicialmente, hay una única meta que cumplir, (in matri4 matri5), puesto que todas las demas se satisfacen en el estado inicial. Por tanto, Prodigy selecciona esa meta y la intenta resolver. El único operador que hace que se cumpla la meta es el operador *put-in*, por lo que lo selecciona. En el siguiente nivel, selecciona los *bindings* de dicho operador, generando un nodo sucesor que contiene el operador instanciado *(put-in matri4 matri5)*. El operador no es ejecutable en ese momento, puesto que la precondición *((closed matri5))* no se satisface en el estado. De hecho, es la única que no se cumple, por lo que intenta satisfacerla. Esa meta se satisface con el operador *open*, por lo que lo selecciona, y genera un nodo sucesor con la única instanciación del operador que consigue cumplir *((closed matri5))*, que es *(open matri5)*. En ese momento, no hay metas pendientes, por lo que Prodigy decide ejecutar el operador *OPEN MATRI5*. Cabe destacar que en este momento, una meta del problema inicial que se satisfacía inicialmente, *(closed matri5)*, deja de satisfacerse, por lo que pasará al conjunto de metas pendientes. Por tanto, Prodigy puede, o bien buscar un operador que la satisfaga, o bien ejecutar un operador pendiente. Prodigy se decanta por la segunda opción, y ejecuta *PUT-IN MATRI4 MATRI5*. En ese momento, no quedan ya operadores pendientes, pero sí la meta *(closed matri5)*, por lo que intenta satisfacerla con el operador *close*, que instancia a *(close matri5)*. En ese momento, se cumplen todas las precondiciones del operador, por lo que no quedan metas pendientes, y lo ejecuta. En ese momento, todas las metas están satisfechas y no queda ningún operador pendiente de ejecutar, por lo que termina la ejecución, y Prodigy devuelve el siguinte plan:

```
OPEN MATRI5
PUT-IN MATRI4 MATRI5
CLOSE MATRI5
```

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.

5. Suponemos que incluimos dos matrioskas mas, *matri*6 y *matri*7. En ese caso, al estado inicial hay que añadirle los siguientes predicados:

   - Sobre la posición de las matrioskas: indicar si están encima de la mesa o dentro de otra matrioska
   - Sobre si están cerradas o abiertas: indicar si están cerradas
   - Sobre la relación con las otras matrioskas: hay que incluir los predicados que muestran la relación de orden o tamaño entre las matrioskas.

   Si asumimos que las matrioskas están cerradas y encima de la mesa, habría que añadir los siguientes literales:

```
(on-table matri6) (on-table matri7)
(closed matri6) (closed matri7)
(bigger matri6 matri5) (bigger matri6 matri4) (bigger matri6 matri3)
(bigger matri6 matri2) (bigger matri6 matri1) (bigger matri7 matri6)
(bigger matri7 matri5) (bigger matri7 matri4) (bigger matri7 matri3)
(bigger matri7 matri2) (bigger matri7 matri1)
```

6. Para calcular el tamaño de estados se puede hacer el siguiente razonamiento. Dadas las n matrioskas, las numeramos por tamaño ascendente desde m_1 hasta m_n. Podemos tomar la primera, m_1. m_1 podría estar en $n+1$ situaciones distintas: abierta encima de la mesa, cerrada encima de la mesa, y cerrada en una de las restantes $n-1$ matrioskas ($n+1 = 1+1+(n-1)$ posiciones). Si asumimos que m_1 está encima de la mesa, m_2 podría estar en n posiciones distintas, abierta encima de la mesa, cerrada encima de la mesa, y cerrada en una de las restantes $n-2$ matrioskas ($n = 1+1+(n-2)$ posiciones).

en las que se pueden encontrar las matrioskas es $(n+1)!$, que viene de multiplicar las $(n+1)$ posiciones en las que se puede colocar la primera matrioska por las n de la segunda, las (n-1) de la tercer, etc. Sin embargo, es fácil ver que este valor no es real, sino una cota superior. Esto es debido a que para calcular el número de posibles posiciones de la matrioska m_2 hemos supuesto que m_1 estaba encima de la mesa. Sin embargo, si m_1 dentro de otra matrioska cualquiera, digamos m_i, m_2 ya no podría estar dentro de m_i, por lo que sólo tendría $n-1$ posibles posiciones.

La cota inferior del número de estados está en 2^n, dado que cada matrioska tiene asegurada dos posibles situaciones: abierta encima de la mesa y cerrada encima de la mesa.

Por tanto, podemos asegurar que el tamaño del espacio de estados están entre 2^n y $(n+1)!$. El cálculo del valor exacto requiere de una contabilización ad-hoc.