Inteligencia Artificial: Formulación, análisis e implementación en Lisp-Scheme del problema "Misioneros y Caníbales"

Carlos Javier Tacón Fernández

14 de marzo de 2018

Índice

1. MC-1

Versión clásica del problema: encontrar con qué movimientos pueden atravesar un río tres misioneros y tres caníbales utilizando una barca de dos plazas, que al menos debe llevar un pasajero para remar, partiendo todos de una de las orillas y con la condición de que en ningún momento queden en ninguna de las orillas más caníbales que misioneros.

1.1. Formulación del problema

- **Morfología de un estado:** {Nº misioneros orilla izquierda, Nº caníbales orilla izquierda, Nº misioneros orilla derecha, Nº caníbales orilla derecha}. El estado será válido si no hay más caníbales que misioneros en cada una de las orillas, a no ser que haya solamente caníbales. Solo se contemplarán números naturales y la suma de elementos entre las dos orillas será constante (3 misioneros y 3 caníbales).
- **Estado inicial:** {0, 0, 3, 3}.
- **Estado meta:** {3, 3, 0, 0}.
- **Estados disponibles:** {0, 0, 3, 3}, {0, 1, 3, 2}, {0, 2, 3, 1}, {0, 3, 3, 0}, {1, 1, 2, 2}, {2, 2, 1, 1}, {3, 0, 0, 3}, {3, 1, 0, 2}, {3, 2, 0, 1}, {3, 3, 0, 0}.
- **Coste de las operaciones:** Suponemos que todas las operaciones tienen igual coste (1).
- **Operaciones disponibles:** Las operaciones serán hacia una orilla u otra dependiendo de dónde se encuentre la barca. Para ello utilizaremos la paridad de la profundidad del nodo a evaluar. La barca tiene máximo dos plazas pero siempre tendrá que haber una persona para dirigirla.

  1. Mover un misionero.
  2. Mover dos misioneros.
  3. Mover un caníbal.
  4. Mover dos caníbales.
  5. Mover un misionero y un caníbal.

1.2. Análisis de la solución

El problema puede resolverse mediante búsqueda desinformada, por los métodos de búsqueda en anchura, búsqueda en profundidad iterativa o búsqueda bidireccional. También podríamos pensar alguna heurística válida, como por ejemplo, si el objetivo es que todas las personas lleguen a la orilla izquierda, podrían tener más preferencia los estados donde haya más personas en esa orilla ({3, 0, 0, 3} mejor que {1, 1, 2, 2}) así podríamos intentar utilizar el método de búsqueda primero el mejor.

Para mejorar la eficiencia del algoritmo podemos guardar una lista de nodos cerrados, ya evaluados teniendo en cuenta el sentido de la operación, en este caso la paridad de la profundidad. También podemos hacer la implementación de tal manera que se descarten los estados generados inválidos (o ya evaluados anteriormente) para que no tengan que ser evaluados otra vez, ya que generarán los mismos descendientes y aumentará tanto el tiempo de procesamiento como la memoria necesaria para guardar la lista de abiertos.

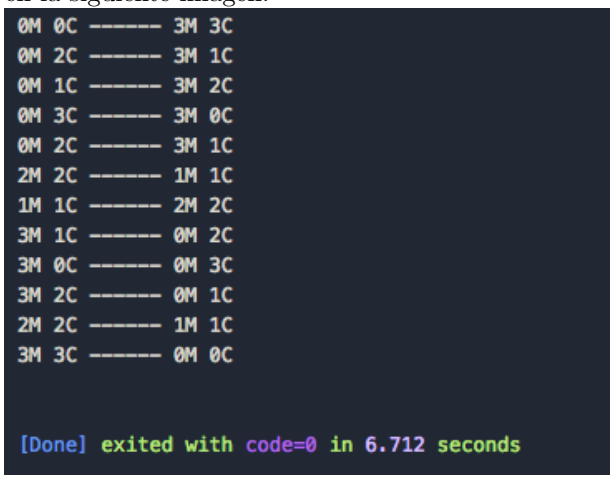
Desarrollamos el árbol de expansión realizando búsqueda en anchura, gestionando la lista de abiertos como FIFO, además no generamos estados inválidos, y no añadimos duplicados a la lista de abiertos. Tampoco desarrollamos los estados que ya hemos evaluado anteriormente [....] (teniendo en cuenta el sentido del viaje, o lo que es lo mismo, la paridad de la profundidad del nodo en el árbol). Las operaciones para generar los hijos se marcan con "Nº operación)" y la profundidad del nodo con [profundidad].

Estado actual	Hijos	Abiertos
-	-	[0]{0,0,3,3}
[0]{0,0,3,3}	3)[1]{0,1,3,2}, 4)[1]{0,2,3,1}, 5)[1]{1,1,2,2}	[1]{0,1,3,2}, [1]{0,2,3,1}, [1]{1,1,2,2}
[1]{0,1,3,2}	3)[2]{0,0,3,3}	[1]{0,2,3,1}, [1]{1,1,2,2}, [2]{0,0,3,3}
[1]{0,2,3,1}	3)[2]{0,1,3,2}, 4)[2]{0,0,3,3}	[1]{1,1,2,2}, [2]{0,0,3,3}, [2]{0,1,3,2}
[1]{1,1,2,2}	1)[2]{0,1,3,2}, 5)[2]{0,0,3,3}	[2]{0,0,3,3}, [2]{0,1,3,2}
[2]{0,0,3,3}	....	[2]{0,1,3,2}
[2]{0,1,3,2}	1)[3]{1,1,2,2}, 3)[3]{0,2,3,1}, 4)[3]{0,3,3,0}	[3]{1,1,2,2}, [3]{0,2,3,1}, [3]{0,3,3,0}
[3]{1,1,2,2}	....	[3]{0,2,3,1}, [3]{0,3,3,0}
[3]{0,2,3,1}	....	[3]{0,3,3,0}
[3]{0,3,3,0}	3)[4]{0,2,3,1}, 4)[4]{0,1,3,2}	[4]{0,2,3,1}, [4]{0,1,3,2}
[4]{0,2,3,1}	2)[5]{2,2,1,1}, 4)[5]{0,3,3,0}	[4]{0,1,3,2}, [5]{2,2,1,1}, [5]{0,3,3,0}
[4]{0,1,3,2}	....	[5]{2,2,1,1}, [5]{0,3,3,0}
[5]{2,2,1,1}	2)[6]{0,2,3,1}, 5)[6]{1,1,2,2}	[5]{0,3,3,0}, [6]{0,2,3,1}, [6]{1,1,2,2}
[5]{0,3,3,0}	....	[6]{0,2,3,1}, [6]{1,1,2,2}
[6]{0,2,3,1}	....	[6]{1,1,2,2}
[6]{1,1,2,2}	2)[7]{3,1,0,2}, 5)[7]{2,2,1,1}	[7]{3,1,0,2}, [7]{2,2,1,1}

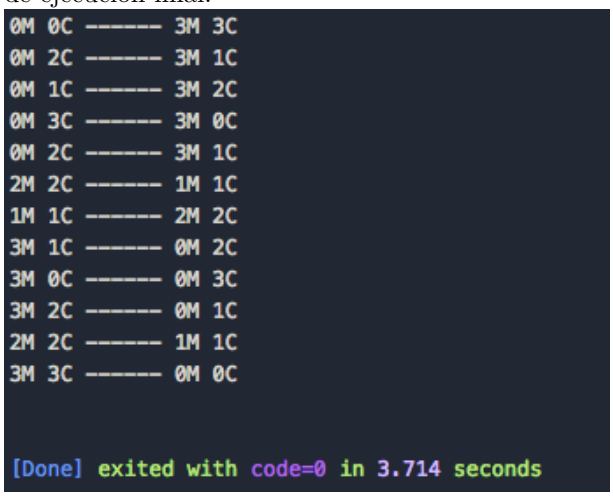
[7]{3,1,0,2}	2)[8]{1,1,2,2}, 3)[8]{3,0,0,3}	[7]{2,2,1,1}, [8]{1,1,2,2}, [8]{3,0,0,3}
[7]{2,2,1,1}	....	[8]{1,1,2,2}, [8]{3,0,0,3}
[8]{1,1,2,2}	....	[8]{3,0,0,3}
[8]{3,0,0,3}	3)[9]{3,1,0,2}, 4)[9]{3,2,0,1}	[9]{3,1,0,2}, [9]{3,2,0,1}
[9]{3,1,0,2}	....	[9]{3,2,0,1}
[9]{3,2,0,1}	1)[10]{2,2,1,1}, 3)[10]{3,1,0,2}, 4)[10]{3,0,0,3}	[10]{2,2,1,1}, [10]{3,1,0,2}, [10]{3,0,0,3}
[10]{2,2,1,1}	1)[11]{3,2,0,1}, 5)[11]{3,3,0,0}	[10]{3,1,0,2}, [10]{3,0,0,3}, [11]{3,2,0,1}, [11]{3,3,0,0}
[10]{3,1,0,2}	3)[11]{3,2,0,1}, 4)[11]{3,3,0,0}	[10]{3,0,0,3}, [11]{3,2,0,1}, [11]{3,3,0,0}
[10]{3,0,0,3}	....	[11]{3,2,0,1}, [11]{3,3,0,0}
[11]{3,2,0,1}	....	[11]{3,3,0,0}
[11]{3,3,0,0}	Éxito	-

El programa implementado en Lisp-Scheme Racket (Anexo A) ha sido escrito usando el algoritmo de búsqueda en anchura, sin generar nodos que sean inválidos. No se ha tenido en cuenta ninguna lista de nodos cerrados, tampoco se ha tenido en cuenta evitar añadir a abiertos nodos con estados duplicados ni se ha implementado una heurística válida. Esto no impide encontrar la respuesta al problema pero el programa consumirá más tiempo y memoria. Al ejecutar el programa podemos ver la respuesta que imprime y el tiempo que ha tardado en encontrarla. La misma respuesta que hemos encontrado con el procedimiento manual.

En un principio se hizo una implementación que generaba todos los nodos a partir de las operaciones sin tener en cuenta su validez. Se puede ver el resultado en la siguiente imagen.



Para intentar mejorar este tiempo de ejecución se implementó una manera de generación de descendientes que solo genera los descendientes que a priori son válidos. Con ese cambio se ha conseguido una mejora notable en el tiempo de ejecución final.



2. MC-2

Versión modificada del problema: encontrar con qué movimientos pueden atravesar un río cuatro misioneros y cuatro caníbales utilizando una barca de dos plazas, que al menos debe llevar un pasajero para remar, partiendo todos de una de las orillas y con la condición de que en ningún momento queden en ninguna de las orillas más caníbales que misioneros.

2.1. Formulación del problema

- **Morfología de un estado:** {Nº misioneros orilla izquierda, Nº caníbales orilla izquierda, Nº misioneros orilla derecha, Nº caníbales orilla derecha}. El estado será válido si no hay más caníbales que misioneros en cada una de las orillas, a no ser que haya solamente caníbales. Solo se contemplarán números naturales y la suma de elementos entre las dos orillas será constante (4 misioneros y 4 caníbales).
- **Estado inicial:** {0, 0, 4, 4}.
- **Estado meta:** {4, 4, 0, 0}.
- **Estados disponibles:** {0, 0, 4, 4}, {0, 1, 4, 3}, {0, 2, 4, 2}, {0, 3, 4, 1}, {0, 4, 4, 0}, {1, 1, 3, 3}, {2, 2, 2, 2}, {3, 3, 1, 1}, {4, 0, 0, 4}, {4, 1, 0, 3}, {4, 2, 0, 2}, {4, 3, 0, 1}, {4, 4, 0, 0}.
- **Coste de las operaciones:** Suponemos que todas las operaciones tienen igual coste (1).
- **Operaciones disponibles:** Las operaciones serán hacia una orilla u otra dependiendo de dónde se encuentre la barca. Para ello utilizaremos la paridad de la profundidad del nodo a evaluar. La barca tiene máximo dos plazas pero siempre tendrá que haber una persona para dirigirla.

  1. Mover un misionero.
  2. Mover dos misioneros.
  3. Mover un caníbal.
  4. Mover dos caníbales.
  5. Mover un misionero y un caníbal.

2.2. Análisis de la solución

Podemos observar que en esta versión del problema, al cambiar el número de misioneros y caníbales, obtenemos un espacio de estados posibles diferente, más amplio, además de cambiar el estado inicial y final. Las operaciones se mantienen exactamente igual que en el apartado anterior, ya que la capacidad de la barca se mantiene en 2.

Este problema no tiene solución, no se puede encontrar un camino para que todas las personas pasen de un lado a otro del río sin que los caníbales se coman a los misioneros ya que la limitación de la barca a 2 personas hace que, llagado un punto, no podamos avanzar en el espacio de estados. Vamos a observar qué pasa si intentamos desarrollar el árbol de expansión por el método de búsqueda en anchura, al igual que anteriormente, sin añadir duplicados a la lista de abiertos y sin desarrollar estados que ya hemos evaluado anteriormente [....] (teniendo en cuenta el sentido del viaje, o lo que es lo mismo, la paridad de la profundidad del nodo en el árbol).

Estado actual	Hijos	Abiertos
-	-	[0]{0,0,4,4}
[0]{0,0,4,4}	3)[1]{0,1,4,3}, 4)[1]{0,2,4,2}, 5)[1]{1,1,3,3}	[1]{0,1,4,3}, [1]{0,2,4,2}, [1]{1,1,3,3}
[1]{0,1,4,3}	3)[2]{0,0,4,4}	[1]{0,2,4,2}, [1]{1,1,3,3}, [2]{0,0,4,4}
[1]{0,2,4,2}	3)[2]{0,1,4,3}, 4)[2]{0,0,4,4}	[1]{1,1,3,3}, [2]{0,0,4,4}, [2]{0,1,4,3}
[1]{1,1,3,3}	1)[2]{0,1,4,3}, 5)[2]{0,0,4,4}	[2]{0,0,4,4}, [2]{0,1,4,3}
[2]{0,0,4,4}	....	[2]{0,1,4,3}
[2]{0,1,4,3}	1)[3]{1,1,3,3}, 3)[3]{0,2,4,2}, 4)[3]{0,3,4,1}	[3]{1,1,3,3}, [3]{0,2,4,2}, [3]{0,3,4,1}
[3]{1,1,3,3}	....	[3]{0,2,4,2}, [3]{0,3,4,1}
[3]{0,2,4,2}	....	[3]{0,3,4,1}
[3]{0,3,4,1}	3)[4]{0,2,4,2}, 4)[4]{0,1,4,3}	[4]{0,2,4,2}
[4]{0,2,4,2}	2)[5]{2,2,2,2}, 3)[5]{0,3,4,1}, 4)[5]{0,4,4,0}	[5]{2,2,2,2}, [5]{0,3,4,1}, [5]{0,4,4,0}
[4]{2,2,2,2}	2)[6]{0,2,4,2}, 5)[6]{1,1,3,3}	[5]{0,3,4,1}, [5]{0,4,4,0}, [6]{0,2,4,2}, [6]{1,1,3,3}
[5]{0,3,4,1}	....	[5]{0,4,4,0}, [6]{0,2,4,2}, [6]{1,1,3,3}
[5]{0,4,4,0}	3)[6]{0,3,4,1}, 4)[6]{0,2,4,2}	[6]{0,2,4,2}, [6]{1,1,3,3}, [6]{0,3,4,1}
[6]{0,2,4,2}	....	[6]{1,1,3,3}, [6]{0,3,4,1}
[6]{1,1,3,3}	5)[7]{2,2,2,2}	[6]{0,3,4,1}, [7]{2,2,2,2}
[6]{0,3,4,1}	3)[7]{0,4,4,0}	[7]{2,2,2,2}, [7]{0,4,4,0}
[7]{2,2,2,2}	5)[8]{1,1,3,3}	[7]{0,4,4,0}, [8]{1,1,3,3}
[7]{0,4,4,0}	....	[8]{1,1,3,3}
[8]{1,1,3,3}	....	....

En este caso llega un momento en que solo encontramos estados que ya hemos evaluado anteriormente, así que podríamos seguir evaluando, pero nunca saldremos del bucle ya que siempre se evaluarían los mismos estados. Si la implementación que hiciésemos llevase el conteo de una lista de cerrados, en el momento en el que la lista de abiertos quedase vacía y no se generasen más descendientes válidos, podríamos parar la ejecución y mostrar un mensaje de error porque no existe solución.

Por esta razón podemos afirmar que este enunciado no puede encontrar un camino válido. Con este tamaño de barca podemos mover como máximo 3 misioneros y 3 caníbales, si aumentáramos el tamaño de la barca sí podríamos encontrar una solución ya que podríamos mover más personas simultáneamente, como desarrollaremos el el siguiente apartado.

3. MC-3

Versión modificada del problema: encontrar con qué movimientos pueden atravesar un río cinco misioneros y cinco caníbales utilizando una barca de tres plazas, que al menos debe llevar un pasajero para remar, partiendo todos de una de las orillas y con la condición de que en ningún momento haya en ninguna de las orillas ni en la barca más caníbales que misioneros.

3.1. Formulación del problema

- **Morfología de un estado:** {Nº misioneros orilla izquierda, Nº caníbales orilla izquierda, Nº misioneros orilla derecha, Nº caníbales orilla derecha}. El estado será válido si no hay más caníbales que misioneros en cada una de las orillas, a no ser que haya solamente caníbales. Solo se contemplarán números naturales y la suma de elementos entre las dos orillas será constante (5 misioneros y 5 caníbales).
- **Estado inicial:** {0, 0, 5, 5}.
- **Estado meta:** {5, 5, 0, 0}.
- **Estados disponibles:** {0, 0, 5, 5}, {0, 1, 5, 4}, {0, 2, 5, 3}, {0, 3, 5, 2}, {0, 4, 5, 1}, {0, 5, 5, 0}, {1, 1, 4, 4}, {2, 2, 3, 3}, {3, 3, 2, 2}, {4, 4, 1, 1}, {5, 0, 0, 5}, {5, 1, 0, 4}, {5, 2, 0, 3}, {5, 3, 0, 2}, {5, 4, 0, 1}, {5, 5, 0, 0}.
- **Coste de las operaciones:** Suponemos que todas las operaciones tienen igual coste (1).
- **Operaciones disponibles:** Las operaciones serán hacia una orilla u otra dependiendo de dónde se encuentre la barca. Para ello utilizaremos la paridad de la profundidad del nodo a evaluar. La barca tiene máximo tres plazas pero siempre tendrá que haber una persona para dirigirla y tampoco se contemplará que haya más caníbales que misioneros en esta barca.

    1. Mover un misionero.
    2. Mover dos misioneros.
    3. Mover tres misioneros.
    4. Mover un caníbal.
    5. Mover dos caníbales.
    6. Mover tres caníbales.
    7. Mover un misionero y un caníbal.
    8. Mover dos misioneros y un caníbal.

3.2. Análisis de la solución

Podemos observar que esta versión del problema es similar a las anteriores, pero ahora tenemos que contar con más misioneros y caníbales, además de contar con una barca de tamaño 3, pudiendo llevar a más personas simultáneamente. El problema tiene solución ya que, aún siendo más personas que el ejercicio anterior, la barca también ha aumentado el tamaño.

Siguiendo el mismo procedimiento que en desarrollos anteriores, analizaremos el problema haciendo una búsqueda en anchura, y para simplificar no desarrollaremos nodos ya evaluados según su dirección (profundidad del nodo) y tampoco añadiremos nodos repetidos a la lista de abiertos.

Estado actual	Hijos	Abiertos
-	-	[0]{0,0,5,5}
[0]{0,0,5,5}	4)[1]{0,1,5,4}, 5)[1]{0,2,5,3}, 6)[1]{0,3,5,2}, 7)[1]{1,1,4,4}	[1]{0,1,5,4}, [1]{0,2,5,3}, [1]{0,3,5,2}, [1]{1,1,4,4}
[1]{0,1,5,4}	4)[2]{0,0,5,5}	[1]{0,2,5,3}, [1]{0,3,5,2}, [1]{1,1,4,4}, [2]{0,0,5,5}
[1]{0,2,5,3}	4)[2]{0,1,5,4}, 5)[2]{0,0,5,5}	[1]{0,3,5,2}, [1]{1,1,4,4}, [2]{0,0,5,5}, [2]{0,1,5,4}
[1]{0,3,5,2}	4)[2]{0,2,5,3}, 5)[2]{0,1,5,4}, 6)[2]{0,0,5,5}	[1]{1,1,4,4}, [2]{0,0,5,5}, [2]{0,1,5,4}, [2]{0,2,5,3}
[1]{1,1,4,4}	4)[2]{0,1,5,4}, 7)[2]{0,0,5,5}	[2]{0,0,5,5}, [2]{0,1,5,4}, [2]{0,2,5,3}
[2]{0,0,5,5}	....	[2]{0,1,5,4}, [2]{0,2,5,3}
[2]{0,1,5,4}	1)[3]{1,1,4,4}, 4)[3]{0,2,5,3}, 5)[3]{0,3,5,2}, 6)[3]{0,4,5,1}, 8)[3]{2,2,3,3}	[2]{0,2,5,3}, [3]{1,1,4,4}, [3]{0,2,5,3}, [3]{0,3,5,2}, [3]{0,4,5,1}, [3]{2,2,3,3}
[2]{0,2,5,3}	2)[3]{2,2,3,3}, 4)[3]{0,3,5,2}, 5)[3]{0,4,5,1}, 6)[3]{0,5,5,0}	[3]{1,1,4,4}, [3]{0,2,5,3}, [3]{0,3,5,2}, [3]{0,4,5,1}, [3]{2,2,3,3}, [3]{0,5,5,0}
[3]{1,1,4,4}	....	[3]{0,2,5,3}, [3]{0,3,5,2}, [3]{0,4,5,1}, [3]{2,2,3,3}, [3]{0,5,5,0}
[3]{0,2,5,3}	....	[3]{0,3,5,2}, [3]{0,4,5,1}, [3]{2,2,3,3}, [3]{0,5,5,0}
[3]{0,3,5,2}	....	[3]{0,4,5,1}, [3]{2,2,3,3}, [3]{0,5,5,0}

[3]{0,4,5,1}	4)[4]{0,3,5,2}, 5)[4]{0,2,5,3}, 6)[4]{0,1,5,4}	[3]{2,2,3,3}, [3]{0,5,5,0}, [4]{0,3,5,2}, [4]{0,2,5,3}, [4]{0,1,5,4}
[3]{2,2,3,3}	7)[4]{1,1,4,4}, 8)[4]{0,1,5,4}	[3]{0,5,5,0}, [4]{0,3,5,2}, [4]{0,2,5,3}, [4]{0,1,5,4}, [4]{1,1,4,4}
[3]{0,5,5,0}	4)[4]{0,4,5,1}, 5)[4]{0,3,5,2}, 6)[4]{0,2,5,3}	[4]{0,3,5,2}, [4]{0,2,5,3}, [4]{0,1,5,4}, [4]{1,1,4,4}, [4]{0,4,5,1}
[4]{0,3,5,2}	3)[5]{3,3,2,2}, 4)[5]{0,4,5,1}, 5)[5]{0,5,5,0}	[4]{0,2,5,3}, [4]{0,1,5,4}, [4]{1,1,4,4}, [4]{0,4,5,1}, [5]{3,3,2,2}, [5]{0,4,5,1}, [5]{0,5,5,0}
[4]{0,2,5,3}	....	[4]{0,1,5,4}, [4]{1,1,4,4}, [4]{0,4,5,1}, [5]{3,3,2,2}, [5]{0,4,5,1}, [5]{0,5,5,0}
[4]{0,1,5,4}	....	[4]{1,1,4,4}, [4]{0,4,5,1}, [5]{3,3,2,2}, [5]{0,4,5,1}, [5]{0,5,5,0}
[4]{1,1,4,4}	7)[5]{2,2,3,3}	[4]{0,4,5,1}, [5]{3,3,2,2}, [5]{0,4,5,1}, [5]{0,5,5,0}, [5]{2,2,3,3}
[4]{0,4,5,1}	4)[5]{0,5,5,0}	[5]{3,3,2,2}, [5]{0,4,5,1}, [5]{0,5,5,0}, [5]{2,2,3,3}
[5]{3,3,2,2}	3)[6]{0,3,5,2}, 7)[6]{2,2,3,3}	[5]{0,4,5,1}, [5]{0,5,5,0}, [5]{2,2,3,3}, [6]{0,3,5,2}, [6]{2,2,3,3}
[5]{0,4,5,1}	....	[5]{0,5,5,0}, [5]{2,2,3,3}, [6]{0,3,5,2}, [6]{2,2,3,3}
[5]{0,5,5,0}	....	[5]{2,2,3,3}, [6]{0,3,5,2}, [6]{2,2,3,3}
[5]{2,2,3,3}	2)[6]{0,2,5,3}, 7)[6]{1,1,4,4}, 8)[6]{0,1,5,4}	[6]{0,3,5,2}, [6]{2,2,3,3}, [6]{0,2,5,3}, [6]{1,1,4,4}, [6]{0,1,5,4}
[6]{0,3,5,2}	....	[6]{2,2,3,3}, [6]{0,2,5,3}, [6]{1,1,4,4}, [6]{0,1,5,4}
[6]{2,2,3,3}	3)[7]{5,2,0,3}, 7)[7]{3,3,2,2}	[6]{0,2,5,3}, [6]{1,1,4,4}, [6]{0,1,5,4}, [7]{5,2,0,3}, [7]{3,3,2,2}
[6]{0,2,5,3}	....	[6]{1,1,4,4}, [6]{0,1,5,4}, [7]{5,2,0,3}, [7]{3,3,2,2}
[6]{1,1,4,4}	....	[6]{0,1,5,4}, [7]{5,2,0,3}, [7]{3,3,2,2}
[6]{0,1,5,4}	....	[7]{5,2,0,3}, [7]{3,3,2,2}
[7]{5,2,0,3}	3)[8]{2,2,3,3}, 4)[8]{5,1,0,4}, 5)[8]{5,0,0,5}	[7]{3,3,2,2}, [8]{2,2,3,3}, [8]{5,1,0,4}, [8]{5,0,0,5}
[7]{3,3,2,2}	....	[8]{2,2,3,3}, [8]{5,1,0,4}, [8]{5,0,0,5}
[8]{2,2,3,3}	....	[8]{5,1,0,4}, [8]{5,0,0,5}
[8]{5,1,0,4}	4)[9]{5,2,0,3}, 5)[9]{5,3,0,2}, 6)[9]{5,4,0,1}	[8]{5,0,0,5}, [9]{5,2,0,3}, [9]{5,3,0,2}, [9]{5,4,0,1}

[8]{5,0,0,5}	4)[9]{5,1,0,4}, 5)[9]{5,2,0,3}, 6)[9]{5,3,0,2}	[9]{5,2,0,3}, [9]{5,3,0,2}, [9]{5,4,0,1}, [9]{5,1,0,4}
[9]{5,2,0,3}	....	[9]{5,3,0,2}, [9]{5,4,0,1}, [9]{5,1,0,4}
[9]{5,3,0,2}	4)[10]{5,2,0,3}, 5)[10]{5,1,0,4}, 6)[10]{5,0,0,5}	[9]{5,4,0,1}, [9]{5,1,0,4}, [10]{5,2,0,3}, [10]{5,1,0,4}, [10]{5,0,0,5}
[9]{5,4,0,1}	1)[10]{4,4,1,1}, 4)[10]{5,3,0,2}, 5)[10]{5,2,0,3}, 6)[10]{5,1,0,4}, 8)[10]{3,3,2,2}	[9]{5,1,0,4}, [10]{5,2,0,3}, [10]{5,1,0,4}, [10]{5,0,0,5}, [10]{4,4,1,1}, [10]{5,3,0,2}, [10]{3,3,2,2}
[9]{5,1,0,4}	4)[10]{5,0,0,5}	[10]{5,2,0,3}, [10]{5,1,0,4}, [10]{5,0,0,5}, [10]{4,4,1,1}, [10]{5,3,0,2}, [10]{3,3,2,2}
[10]{5,2,0,3}	4)[11]{5,3,0,2}, 5)[11]{5,4,0,1}, 6)[11]{5,5,0,0}	[10]{5,1,0,4}, [10]{5,0,0,5}, [10]{4,4,1,1}, [10]{5,3,0,2}, [10]{3,3,2,2}, [11]{5,3,0,2}, [11]{5,4,0,1}, [11]{5,5,0,0}
[10]{5,1,0,4}	....	[10]{5,0,0,5}, [10]{4,4,1,1}, [10]{5,3,0,2}, [10]{3,3,2,2}, [11]{5,3,0,2}, [11]{5,4,0,1}, [11]{5,5,0,0}
[10]{5,0,0,5}	....	[10]{4,4,1,1}, [10]{5,3,0,2}, [10]{3,3,2,2}, [11]{5,3,0,2}, [11]{5,4,0,1}, [11]{5,5,0,0}
[10]{4,4,1,1}	1)[11]{5,4,0,1}, 7)[11]{5,5,0,0}	[10]{5,3,0,2}, [10]{3,3,2,2}, [11]{5,3,0,2}, [11]{5,4,0,1}, [11]{5,5,0,0}
[10]{5,3,0,2}	5)[11]{5,4,0,1}, 6)[11]{5,5,0,0}	[10]{3,3,2,2}, [11]{5,3,0,2}, [11]{5,4,0,1}, [11]{5,5,0,0}
[10]{3,3,2,2}	7)[11]{4,4,1,1}, 8)[11]{5,4,0,1}	[11]{5,3,0,2}, [11]{5,4,0,1}, [11]{5,5,0,0}, [11]{4,4,1,1}
[11]{5,3,0,2}	....	[11]{5,4,0,1}, [11]{5,5,0,0}, [11]{4,4,1,1}
[11]{5,4,0,1}	....	[11]{5,5,0,0}, [11]{4,4,1,1}
[11]{5,5,0,0}	Éxito	[11]{4,4,1,1}

La implementación de este programa Lisp-Scheme Racket (Anexo B) es una variación de la implementación MC-1, modificando los estados inicial y final además de añadir las nuevas operaciones disponibles. El procedimiento de búsqueda es el mismo, búsqueda en anchura teniendo en cuenta la generación solamente de nodos válidos.

Podemos ver así cómo se dispara en tiempo de ejecución para encontrar el camino solución con respecto a MC-1, en una primera implementación donde no se comprobaba si el nodo generado era válido antes de generarlo (7 segundos vs. 20139 segundos).

```
0M 0C ------ 5M 5C
0M 2C ------ 5M 3C
0M 1C ------ 5M 4C
0M 4C ------ 5M 1C
0M 3C ------ 5M 2C
3M 3C ------ 2M 2C
2M 2C ------ 3M 3C
5M 2C ------ 0M 3C
5M 1C ------ 0M 4C
5M 3C ------ 0M 2C
5M 2C ------ 0M 3C
5M 5C ------ 0M 0C


[Done] exited with code=0 in 20138.833 seconds
```

Después también se actualizó el algoritmo de generación de descendientes para generar solo los nodos válidos, quedando así el resultado de su ejecución, donde se puede observar la mejora en tiempo respecto a la implementación anterior. Si se hubiesen aplicado otras formas de mejora del algoritmo, como una lista de nodos cerrados o una heurística válida, este tiempo debería ser aún menor.

```
0M 0C ------ 5M 5C
0M 2C ------ 5M 3C
0M 1C ------ 5M 4C
0M 4C ------ 5M 1C
0M 3C ------ 5M 2C
3M 3C ------ 2M 2C
2M 2C ------ 3M 3C
5M 2C ------ 0M 3C
5M 1C ------ 0M 4C
5M 3C ------ 0M 2C
5M 2C ------ 0M 3C
5M 5C ------ 0M 0C


[Done] exited with code=0 in 6437.424 seconds
```

Anexos

A. Implementación MC-1 en Lisp-Scheme

```
;;; ======================================== ;;;
;;; problema clásico de misioneros y caníbales ;;;
;;; ======================================== ;;;
;;; se encuentran tres misioneros y tres caníbales en el lado
;;; derecho de un río, necesitan cruzar todos al lado izquierdo
;;; del mismo, para ello hay una barca de dos plazas. cómo hacer
;;; que pasen todos al otro lado sin que haya nunca más caníbales
;;; que misioneros en el mismo lado


#lang racket

;;; ================= ;;;
;;; métodos auxiliares ;;;
;;; ================= ;;;

;;; construye la estructura de un nodo a partir de los datos
;;; necesarios [profundidad en el árbol, estado y camino]
;;; el estado se compone de un par de pares, el primer par
;;; representa el lado derecho del río, el segundo par el lado
;;; izquierdo el camino es una lista con el histórico de estados.
;;; estructura: '('((profundidad).(estado)).'(camino))
(define (construir_nodo profundidad estado camino)
    (cons (cons profundidad estado) camino)
)

;;; devuelve el estado a partir de la estructura de un nodo
(define (get_estado nodo) (cdr (car nodo)))
;;; devuelve la profundidad en el árbol a partir de un nodo
(define (get_profundidad nodo) (car (car nodo)))
;;; devuelve la lista de los estados recorridos sin el actual
(define (get_camino_recorrido nodo) (cdr nodo))

;;; devuelve la parte que simboliza la orilla derecha del río
(define (get_orilla_drcha estado) (car estado))
;;; devuelve la parte que simboliza la orilla izquierda del río
(define (get_orilla_izqui estado) (cdr estado))

;;; devuelve el número que representa a los misioneros
(define (get_misioneros orilla) (car orilla))
;;; devuelve el número que representa a los caníbales
(define (get_canibales  orilla) (cdr orilla))
```

```
;;; método recursivo para imprimir un estado del camino
(define (imprimir_camino camino)
    (if (empty? camino) (display "\n")
        (and
            (display (get_misioneros
                (get_orilla_izqui (car camino))))
            (display "M␣")
            (display (get_canibales
                (get_orilla_izqui (car camino))))
            (display "C␣")
            (display "------␣")
            (display (get_misioneros
                (get_orilla_drcha (car camino))))
            (display "M␣")
            (display (get_canibales
                (get_orilla_drcha (car camino))))
            (displayln "C")

            (imprimir_camino (cdr camino)))
    )
)

;;; ================================== ;;;
;;; generación de estados, inicial y meta ;;;
;;; ================================== ;;;
;;; se definen las dos orillas como (cons num_misioneros
;;; num_canibales) y con ellas se construyen los estados como
;;; (cons orilla_drcha orilla_izqui)

; tres misioneros y tres caníbales en la derecha
(define estado_inicial (cons (cons 3 3) (cons 0 0)))
; tres misioneros y tres caníbales en la izquierda
(define estado_meta (cons (cons 0 0) (cons 3 3)))

; se comprueba que el estado actual sea igual al estado meta
(define (es_meta nodo) (equal? (get_estado nodo) estado_meta))

; el estado no es válido si hay más caníbales o
; si se han generado numeros negativos
(define (estado_invalido estado)
    (or
        ;;; no puede haber caníbales o misioneros negativos
        (< (get_canibales  (get_orilla_drcha estado)) 0)
        (< (get_misioneros (get_orilla_drcha estado)) 0)
        (< (get_canibales  (get_orilla_izqui estado)) 0)
        (< (get_misioneros (get_orilla_izqui estado)) 0)

        ;;; si hay más caníbales que misioneros en cualquier
        ;;; orilla el estado no es válido, si hay misioneros
        (< 0 (get_misioneros (get_orilla_drcha estado))
```

```
            (get_canibales (get_orilla_drcha estado)))
        (< 0 (get_misioneros (get_orilla_izqui estado))
            (get_canibales (get_orilla_izqui estado)))
    )
)

;;; ====================== ;;;
;;; operaciones disponibles ;;;
;;; ====================== ;;;
;;; 1 - mover un misionero
;;; 2 - mover dos misioneros
;;; 3 - mover un caníbal
;;; 4 - mover dos caníbales
;;; 5 - mover un misionero y un caníbal
;;; *** dependiendo de la paridad de la profundidad del nodo a
;;;     evaluar, el movimiento será de derecha a izquierda si la
;;;     profundidad es par, al contrario si es impar
;;; *** suponemos que los costes de transiciones son iguales

; método de generación de estados basado en la suma en una orilla
; a costa de la resta de personas en la otra
(define (generar_estado nodo num_misioneros num_canibales)
    ; si es par se mueve hacia la izquierda, si no a la derecha
    (if (even? (get_profundidad nodo))
        (cons
            (cons
                (- (get_misioneros (get_orilla_drcha
                    (get_estado nodo))) num_misioneros)
                (- (get_canibales (get_orilla_drcha
                    (get_estado nodo))) num_canibales))
            (cons
                (+ (get_misioneros (get_orilla_izqui
                    (get_estado nodo))) num_misioneros)
                (+ (get_canibales (get_orilla_izqui
                    (get_estado nodo))) num_canibales))
        )
        (cons
            (cons
                (+ (get_misioneros (get_orilla_drcha
                    (get_estado nodo))) num_misioneros)
                (+ (get_canibales (get_orilla_drcha
                    (get_estado nodo))) num_canibales))
            (cons
                (- (get_misioneros (get_orilla_izqui
                    (get_estado nodo))) num_misioneros)
                (- (get_canibales (get_orilla_izqui
                    (get_estado nodo))) num_canibales))
        )
    )
)
```

```
; método para generar todos los hijos posibles de un nodo,
; enumeración de movimientos posibles, prioridad según posición
; en la lista generada. teniendo en cuenta la eliminación
; de nodos inválidos para menor tiempo de ejecución
(define (generar_hijos nodo)
    (remove* (list 0) (list
        (if (estado_invalido (generar_estado nodo 1 0)) 0
            (construir_nodo (+ 1 (get_profundidad nodo))
                (generar_estado nodo 1 0) (cons (get_estado nodo)
                (get_camino_recorrido nodo))))
        (if (estado_invalido (generar_estado nodo 2 0)) 0
            (construir_nodo (+ 1 (get_profundidad nodo))
                (generar_estado nodo 2 0) (cons (get_estado nodo)
                (get_camino_recorrido nodo))))
        (if (estado_invalido (generar_estado nodo 0 1)) 0
            (construir_nodo (+ 1 (get_profundidad nodo))
                (generar_estado nodo 0 1) (cons (get_estado nodo)
                (get_camino_recorrido nodo))))
        (if (estado_invalido (generar_estado nodo 0 2)) 0
            (construir_nodo (+ 1 (get_profundidad nodo))
                (generar_estado nodo 0 2) (cons (get_estado nodo)
                (get_camino_recorrido nodo))))
        (if (estado_invalido (generar_estado nodo 1 1)) 0
            (construir_nodo (+ 1 (get_profundidad nodo))
                (generar_estado nodo 1 1) (cons (get_estado nodo)
                (get_camino_recorrido nodo))))
    ))
)

;;; ======================================== ;;;
;;; recorrido en anchura de árbol de expansión ;;;
;;; ======================================== ;;;
;;; búsqueda desinformada utilizando el algoritmo de búsqueda
;;; en anchura para encontrar siempre (algoritmo completo)
;;; el camino óptimo

; método recursivo que tiene por parámetros el nodo a
; evaluar además de la lista de nodos abiertos
(define (buscar_solucion nodo_actual abiertos)
    (cond
        ; cuando el nodo a evaluar es meta, se imprime la solución,
        ; que es el camino recorrido más el nodo actual, reverso
        ; ya que se almacena como una pila
        ((es_meta nodo_actual)
            (imprimir_camino (reverse (cons (get_estado nodo_actual)
                (get_camino_recorrido nodo_actual)))))

        ; si el estado actual es inválido, se descarta el nodo,
        ; se busca la solución en el siguiente de la lista de
```

```
        ; abiertos
        ((estado_invalido (get_estado nodo_actual))
            (buscar_solucion (car abiertos) (cdr abiertos)))

        ; si el nodo sí es válido, pero no solución, se expandirá
        ; el árbol generando los hijos de ese nodo y añadiéndolos
        ; a abiertos, pasasndo al siguiente
        (else
            (buscar_solucion
                (if (empty? abiertos)
                    ; si la lista de abiertos está vacía,
                    ; evaluaremos el primer hijo
                    (car (generar_hijos nodo_actual))
                    ; si no, seguimos adelante con el primer
                    ; nodo en abiertos
                    (car abiertos)
                )
                (if (empty? abiertos)
                    ; si la lista de abiertos está vacía, hemos
                    ; pasado como actual el primer hijo, como
                    ; abiertos el resto
                    (cdr (generar_hijos nodo_actual))
                    ; si no concatenamos los hijos generados, a
                    ; la lista abiertos actual, con el esquema fifo
                    (append (cdr abiertos) (generar_hijos nodo_actual))
                )
            )
        )
    )
)

;;; ================== ;;;
;;; programa principal ;;;
;;; ================== ;;;
;;; llamada a buscar solución construyendo el nodo raíz, con
;;; profundidad 0, estado inicial, camino vacío y lista vacía
;;; de abiertos
(buscar_solucion (construir_nodo 0 estado_inicial '()) '())
```

B. Implementación MC-3 en Lisp-Scheme

```
;;; ========================================== ;;;
;;; problema modificado de misioneros y caníbales ;;;
;;; ========================================== ;;;
;;; se encuentran cinco misioneros y cinco caníbales en el lado
;;; derecho de un río, necesitan cruzar todos al lado izquierdo
;;; del mismo, para ello hay una barca de tres plazas. cómo hacer
;;; que pasen todos al otro lado sin que haya nunca más caníbales
;;; que misioneros en el mismo lado o en la barca


#lang racket

;;; ================= ;;;
;;; métodos auxiliares ;;;
;;; ================= ;;;

;;; construye la estructura de un nodo a partir de los datos
;;; necesarios [profundidad en el árbol, estado y camino]
;;; el estado se compone de un par de pares, el primer par
;;; representa el lado derecho del río, el segundo par el lado
;;; izquierdo el camino es una lista con el histórico de estados.
;;; estructura: '('((profundidad).(estado)).'(camino))
(define (construir_nodo profundidad estado camino)
    (cons (cons profundidad estado) camino)
)

;;; devuelve el estado a partir de la estructura de un nodo
(define (get_estado nodo) (cdr (car nodo)))
;;; devuelve la profundidad en el árbol a partir de un nodo
(define (get_profundidad nodo) (car (car nodo)))
;;; devuelve la lista de los estados recorridos sin el actual
(define (get_camino_recorrido nodo) (cdr nodo))

;;; devuelve la parte que simboliza la orilla derecha del río
(define (get_orilla_drcha estado) (car estado))
;;; devuelve la parte que simboliza la orilla izquierda del río
(define (get_orilla_izqui estado) (cdr estado))

;;; devuelve el número que representa a los misioneros
(define (get_misioneros orilla) (car orilla))
;;; devuelve el número que representa a los caníbales
(define (get_canibales  orilla) (cdr orilla))

;;; método recursivo para imprimir un estado del camino
(define (imprimir_camino camino)
    (if (empty? camino) (display "\n")
```

```
        (and
            (display (get_misioneros
                (get_orilla_izqui (car camino))))
            (display "M␣")
            (display (get_canibales
                (get_orilla_izqui (car camino))))
            (display "C␣")
            (display "------␣")
            (display (get_misioneros
                (get_orilla_drcha (car camino))))
            (display "M␣")
            (display (get_canibales
                (get_orilla_drcha (car camino))))
            (displayln "C")

            (imprimir_camino (cdr camino)))
    )
)

;;; ================================== ;;;
;;; generación de estados, inicial y meta ;;;
;;; ================================== ;;;
;;; se definen las dos orillas como (cons num_misioneros
;;; num_canibales) y con ellas se construyen los estados como
;;; (cons orilla_drcha orilla_izqui)

; tres misioneros y tres caníbales en la derecha
(define estado_inicial (cons (cons 5 5) (cons 0 0)))
; tres misioneros y tres caníbales en la izquierda
(define estado_meta (cons (cons 0 0) (cons 5 5)))

; se comprueba que el estado actual sea igual al estado meta
(define (es_meta nodo) (equal? (get_estado nodo) estado_meta))

; el estado no es válido si hay más caníbales o
; si se han generado numeros negativos
(define (estado_invalido estado)
    (or
        ;;; no puede haber caníbales o misioneros negativos
        (< (get_canibales  (get_orilla_drcha estado)) 0)
        (< (get_misioneros (get_orilla_drcha estado)) 0)
        (< (get_canibales  (get_orilla_izqui estado)) 0)
        (< (get_misioneros (get_orilla_izqui estado)) 0)

        ;;; si hay más caníbales que misioneros en cualquier
        ;;; orilla el estado no es válido, si hay misioneros
        (< 0 (get_misioneros (get_orilla_drcha estado))
            (get_canibales (get_orilla_drcha estado)))
        (< 0 (get_misioneros (get_orilla_izqui estado))
            (get_canibales (get_orilla_izqui estado)))
```

```
    )
)

;;; ====================== ;;;
;;; operaciones disponibles ;;;
;;; ====================== ;;;
;;; 1 - mover un misionero
;;; 2 - mover dos misioneros
;;; 3 - mover tres misioneros
;;; 4 - mover un caníbal
;;; 5 - mover dos caníbales
;;; 6 - mover tres caníbales
;;; 7 - mover un misionero y un caníbal
;;; 8 - mover dos misioneros y un caníbal
;;; *** dependiendo de la paridad de la profundidad del nodo a
;;;     evaluar, el movimiento será de derecha a izquierda si la
;;;     profundidad es par, al contrario si es impar
;;; *** suponemos que los costes de transiciones son iguales

; método de generación de estados basado en la suma en una orilla
; a costa de la resta de personas en la otra
(define (generar_estado nodo num_misioneros num_canibales)
    ; si es par se mueve hacia la izquierda, si no a la derecha
    (if (even? (get_profundidad nodo))
        (cons
            (cons
                (- (get_misioneros (get_orilla_drcha
                    (get_estado nodo))) num_misioneros)
                (- (get_canibales (get_orilla_drcha
                    (get_estado nodo))) num_canibales))
            (cons
                (+ (get_misioneros (get_orilla_izqui
                    (get_estado nodo))) num_misioneros)
                (+ (get_canibales (get_orilla_izqui
                    (get_estado nodo))) num_canibales))
        )
        (cons
            (cons
                (+ (get_misioneros (get_orilla_drcha
                    (get_estado nodo))) num_misioneros)
                (+ (get_canibales (get_orilla_drcha
                    (get_estado nodo))) num_canibales))
            (cons
                (- (get_misioneros (get_orilla_izqui
                    (get_estado nodo))) num_misioneros)
                (- (get_canibales (get_orilla_izqui
                    (get_estado nodo))) num_canibales))
        )
    )
)
```

```
; método para generar todos los hijos posibles de un nodo,
; enumeración de movimientos posibles, prioridad según posición
; en la lista generada. teniendo en cuenta la eliminación
; de nodos inválidos para menor tiempo de ejecución
(define (generar_hijos nodo)
    (remove* (list 0) (list
        (if (estado_invalido (generar_estado nodo 1 0)) 0
            (construir_nodo (+ 1 (get_profundidad nodo))
                (generar_estado nodo 1 0) (cons (get_estado nodo)
                (get_camino_recorrido nodo))))
        (if (estado_invalido (generar_estado nodo 2 0)) 0
            (construir_nodo (+ 1 (get_profundidad nodo))
                (generar_estado nodo 2 0) (cons (get_estado nodo)
                (get_camino_recorrido nodo))))
        (if (estado_invalido (generar_estado nodo 3 0)) 0
            (construir_nodo (+ 1 (get_profundidad nodo))
                (generar_estado nodo 3 0) (cons (get_estado nodo)
                (get_camino_recorrido nodo))))
        (if (estado_invalido (generar_estado nodo 0 1)) 0
            (construir_nodo (+ 1 (get_profundidad nodo))
                (generar_estado nodo 0 1) (cons (get_estado nodo)
                (get_camino_recorrido nodo))))
        (if (estado_invalido (generar_estado nodo 0 2)) 0
            (construir_nodo (+ 1 (get_profundidad nodo))
                (generar_estado nodo 0 2) (cons (get_estado nodo)
                (get_camino_recorrido nodo))))
        (if (estado_invalido (generar_estado nodo 0 3)) 0
            (construir_nodo (+ 1 (get_profundidad nodo))
                (generar_estado nodo 0 3) (cons (get_estado nodo)
                (get_camino_recorrido nodo))))
        (if (estado_invalido (generar_estado nodo 1 1)) 0
            (construir_nodo (+ 1 (get_profundidad nodo))
                (generar_estado nodo 1 1) (cons (get_estado nodo)
                (get_camino_recorrido nodo))))
        (if (estado_invalido (generar_estado nodo 2 1)) 0
            (construir_nodo (+ 1 (get_profundidad nodo))
                (generar_estado nodo 2 1) (cons (get_estado nodo)
                (get_camino_recorrido nodo))))
    ))
)

;;; ======================================== ;;;
;;; recorrido en anchura de árbol de expansión ;;;
;;; ======================================== ;;;
;;; búsqueda desinformada utilizando el algoritmo de búsqueda
;;; en anchura para encontrar siempre (algoritmo completo)
;;; el camino óptimo

; método recursivo que tiene por parámetros el nodo a
```

```
; evaluar además de la lista de nodos abiertos
(define (buscar_solucion nodo_actual abiertos)
    (cond
        ; cuando el nodo a evaluar es meta, se imprime la solución,
        ; que es el camino recorrido más el nodo actual, reverso
        ; ya que se almacena como una pila
        ((es_meta nodo_actual)
            (imprimir_camino (reverse (cons (get_estado nodo_actual)
                (get_camino_recorrido nodo_actual)))))

        ; si el estado actual es inválido, se descarta el nodo,
        ; se busca la solución en el siguiente de la lista de
        ; abiertos
        ((estado_invalido (get_estado nodo_actual))
            (buscar_solucion (car abiertos) (cdr abiertos)))

        ; si el nodo sí es válido, pero no solución, se expandirá
        ; el árbol generando los hijos de ese nodo y añadiéndolos
        ; a abiertos, pasasndo al siguiente
        (else
            (buscar_solucion
                (if (empty? abiertos)
                    ; si la lista de abiertos está vacía,
                    ; evaluaremos el primer hijo
                    (car (generar_hijos nodo_actual))
                    ; si no, seguimos adelante con el primer
                    ; nodo en abiertos
                    (car abiertos)
                )
                (if (empty? abiertos)
                    ; si la lista de abiertos está vacía, hemos
                    ; pasado como actual el primer hijo, como
                    ; abiertos el resto
                    (cdr (generar_hijos nodo_actual))
                    ; si no concatenamos los hijos generados, a
                    ; la lista abiertos actual, con el esquema fifo
                    (append (cdr abiertos) (generar_hijos nodo_actual))
                )
            )
        )
    )
)

;;; ================== ;;;
;;; programa principal ;;;
;;; ================== ;;;
;;; llamada a buscar solución construyendo el nodo raíz, con
;;; profundidad 0, estado inicial, camino vacío y lista vacía
;;; de abiertos
(buscar_solucion (construir_nodo 0 estado_inicial '()) '())
```