DECSAI
Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

Inteligencia Artificial en Investigación Operativa

Curso académico 2012/2013

Práctica 1

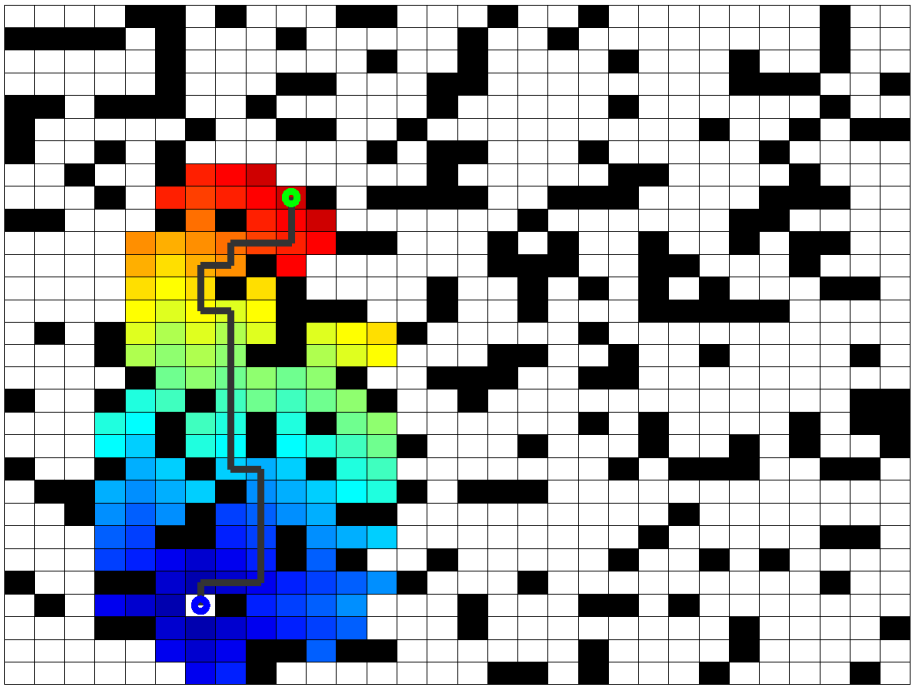
Técnicas de búsqueda heurística

ENTREGA DE LA PRÁCTICA
(a través del acceso identificado de DECSAI)
`https://decsai.ugr.es/`

```
resultados.m
initialState.m (x2)
nextState.m (x2)
sameState.m (x2)
finalState.m (x2)
heuristic.m (x2)
```

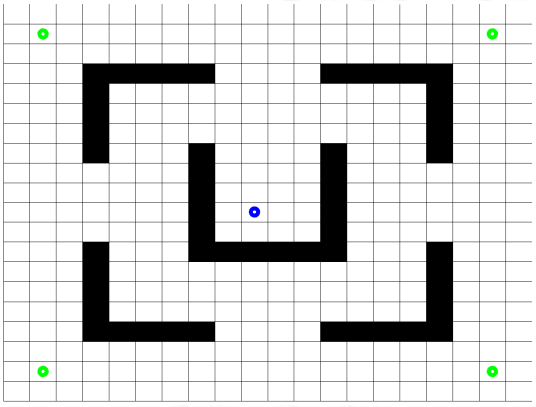
Técnicas de búsqueda heurística

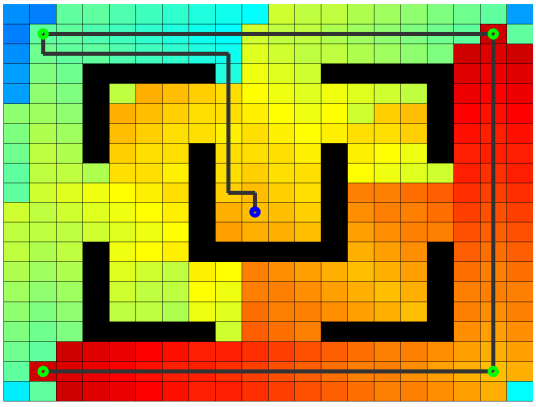
En esta práctica, analizaremos el funcionamiento de distintas técnicas de búsqueda a la hora de encontrar la ruta óptima que un robot debe seguir a través de un tablero para llegar desde una casilla inicial (marcada en azul) a una casilla final (marcada en verde) previamente establecidas. El tablero puede incluir obstáculos (casillas que el robot no puede atravesar), tal como se muestra en la siguiente figura:



En la primera parte de la práctica, deberá resolver un problema de búsqueda simple, en el que ha de llegar a la casilla marcada como objetivo.

En la segunda parte de la práctica, implementará un agente de búsqueda que sea capaz de encontrar la ruta óptima que visita todas las casillas marcadas como objetivos a partir de una casilla inicial determinada (al estilo de un juego tipo "comecocos"):





Ficheros de MATLAB

Junto con el guión de la práctica, tiene a sus disposición una serie de ficheros en los que ya están implementadas algunas estrategias de búsqueda y un conjunto de plantillas que deberá rellenar para conseguir que su implementación funcione correctamente. Además de estos ficheros, también ha de rellenar el fichero `resultados.m` con sus respuestas a los distintos ejercicios propuestos en este guión.

A continuación se describe la función de los distintos ficheros de los que dispone:

- Los ficheros `*Board.m` crean distintas configuraciones de tableros sobre los que efectuar el proceso de búsqueda.

- El fichero `actions.m` define las distintas acciones que puede aplicar su agente de búsqueda (en este caso: izquierda [`L`], derecha [`R`], arriba [`U`] y abajo [`D`]).

- Los ficheros `initialState.m`, `nextState.m`, `finalState.m` y `sameState.m` son los ficheros que hemos de implementar para representar correctamente nuestro espacio de búsqueda.

- El fichero `heuristic.m` es el fichero que debe rellenar para implementar correctamente distintas heurísticas para el problema de búsqueda planteado.

- Los ficheros `dfs.m`, `bfs.m`, `ucs.m` y `astar.m` implementan los algoritmos de búsqueda en profundidad, en anchura, de coste uniforme y A*, respectivamente.

- Los ficheros `createFigure.m` y `path.m` son ficheros internos que se utilizan para visualizar el tablero y reconstruir el camino desde el nodo inicial hasta la solución.

- El fichero `demo.m` sirve de demostración del uso de las distintas funciones implementadas en esta práctica: cómo crear un tablero de un tipo determinado y cómo lanzar el proceso de búsqueda.

- El fichero `test.m` incluye una batería de casos de prueba que una implementación correcta de la práctica debería superar con éxito (su calificación final dependerá [parcialmente] de su grado de éxito a la hora de superar esta batería de prueba).

- El fichero `resultados.m` deberá rellenarlo con sus respuestas a las distintas cuestiones y ejercicios propuestos en este guión (junto con los casos de prueba de `test.m`, determinará su calificación de la práctica).

Representación de un problema de búsqueda

Las funciones definidas en los ficheros `*Board.m` se utilizan para crear distintas configuraciones de tableros sobre los que luego se efectuará el proceso de búsqueda:

`board = randomBoard(n,w);`
crea un tablero aleatorio de tamaño `nxn` en el que el `w`% de las casillas son obstáculos (en ocasiones, los obstáculos harán que sea imposible resolver el problema de búsqueda, al no existir un camino desde la casilla inicial hasta la casilla de salida).

`board = frontierBoard(n);`
crea un tablero sencillo de tamaño `nxn`, sin obstáculos, para que pueda ver la forma que adquiere la frontera de búsqueda al utilizar distintos algoritmos de búsqueda.

`board = clearBoard(n);`
crea un tablero de tamaño `nxn`, sin obstáculos, con las casillas inicial y final en esquinas opuestas del tablero.

`board = impossibleBoard(n);`
crea un tablero en el que es imposible encontrar la salida, al no existir ésta.

`board = pacmanBoard(n);`
crea un tablero de tamaño `nxn` con una configuración similar al del conocido videojuego del comecocos (`pacman` en inglés).

`board = zigzagBoard(n);`
crea un tablero de tamaño `nxn` con paredes que impiden encontrar un camino recto hacia la salida, lo que nos obliga a ir haciendo zigzag.

`board = costBoard(n);`
crea un tablero de tamaño `nxn` en el que el coste de recorrer las casillas inferiores es superior al coste de las casillas superiores, lo que nos empujará siempre a ir hacia arriba si queremos encontrar la ruta óptima.

La implementación de todos estos tableros tiene una estructura similar e incluye información que le resultará necesaria para su implementación:

- `board.n` nos indica el tamaño del lado del tablero
- `board.start.x` y `board.start.y` nos dan las coordenadas de la casilla inicial.
- `board.goal.x` y `board.goal.y` nos proporcionan la posición de la salida.
- `board.cost(x,y)` nos indica el coste que supone atravesar la casilla (x,y).

Representación de los estados del espacio de búsqueda

Su primera tarea consiste en definir adecuadamente la forma de representar los estados del problema de búsqueda que consiste en, partiendo de la posición inicial (dada por `board.start.x` y `board.start.y`), encontrar el camino óptimo hasta la salida (marcada por `board.goal.x` y `board.goal.y`).

En este caso, bastará con mantener la posición actual de nuestro robot (sus coordenadas x e y, que guardaremos en `state.x` y `state.y`) junto con el coste que supone haber llegado hasta el estado actual (valor que almacenaremos en `state.cost`, donde `state` representa el estado en el que nos encontramos). Internamente, para reconstruir el camino hasta la solución, también se utiliza una cuarta variable [`state.prev`] que **NO** deberá modificar bajo ningún concepto si desea que su implementació funcione correctamente.

Para representar el problema de búsqueda y poder ejecutar los distintos algoritmos de búsqueda estudiados en clase, deberá proporcionar una implementación válida de las siguientes funciones en MATLAB:

- `initialState(board)` crea el estado inicial de la búsqueda a partir de los datos proporcionados por el tablero sobre el que se efectuará la búsqueda.

- `sameState(this, other)` nos indica si dos estados, `this` y `other`, representan el mismo estado de búsqueda o no (algo que resulta necesario para evitar ciclos al realizar la búsqueda, p.ej. cuando se llega a estados ya visitados).

- `finalState (board, state)` nos indica si el estado `state` corresponde a uno de los posibles estados finales para el problema de búsqueda definido por el tablero `board`.

- `nextState (board, state, action)` es la función que utilizaremos para obtener el estado al que se llega cuando, estando en el estado `state`, aplicamos la acción `action` en el problema de búsqueda definido por el tablero `board`. En esta función deberá calcular correctamente las coordenadas de nuestro robot (sin salirse del tablero) y calcular el coste que supone llegar al nuevo estado a partir del actual.

En nuestro problema, las acciones disponibles son las definidas por la función `actions`: `L` (moverse a la casilla de la izquierda), `R` (moverse a la casilla de la derecha), `U` (moverse a la casilla superior) y `D` (moverse a la casilla inferior).

Antes de pasar al siguiente apartado, asegúrese de que su implementación de las funciones anteriores funciona correctamente diseñando distintos casos de prueba.

Búsqueda sin información

Una vez implementadas las funciones adecuadas para representar nuestro problema de búsqueda, podemos probar el funcionamiento de las distintas técnicas de búsqueda

DFS: Búsqueda en profundidad

La función definida en `dfs.m` implementa una búsqueda en profundidad.

EJERCICIO 1: DFS [5%]
Compruebe el funcionamiento de la búsqueda en profundidad para las siguientes configuraciones de tableros e indique en `resultados.m` el número de nodos expandidos en la búsqueda junto con la longitud del camino encontrado hasta la solución (0 en caso de que no exista tal solución):

1. `frontierBoard`, de tamaño 20x20, para ver la frontera de búsqueda.
2. `clearBoard`, de tamaño 20x20
3. `impossibleBoard`, de tamaño 20x20
4. `pacmanBoard`, de tamaño 20x20
5. `zigzagBoard`, de tamaño 20x20
6. `costBoard`, de tamaño 20x20
7. El tablero aleatorio creado con `randomBoard()`, de tamaño 20x20, que se encuentra en el fichero `randomBoard.mat`.

BFS: Búsqueda en anchura

La función definida en `bfs.m` implementa una búsqueda en anchura.

EJERCICIO 2: BFS [5%]
Compruebe el funcionamiento de la búsqueda en anchura y observe las diferencias con respecto a los resultados obtenidos con DFS para las siguientes configuraciones de tableros e indique en `resultados.m` el número de nodos expandidos en la búsqueda junto con la longitud del camino encontrado hasta la solución (0 en caso de que no exista tal solución):

1. `frontierBoard`, de tamaño 20x20.
2. `clearBoard`, de tamaño 20x20.
3. `impossibleBoard`, de tamaño 20x20.
4. `pacmanBoard`, de tamaño 20x20.
5. `zigzagBoard`, de tamaño 20x20.
6. `costBoard`, de tamaño 20x20.
7. El tablero aleatorio creado con `randomBoard()`, de tamaño 20x20, que se encuentra en el fichero `randomBoard.mat`.

Búsqueda heurística

Para utilizar técnicas de búsqueda heurística, lo primero que tiene que hacer es implementar las distintas funciones heurísticas que utilizaremos en esta práctica en el fichero `heuristics.m`. En particular, debe implementar las siguientes heurísticas:

- `'ucs'` para el algoritmo de búsqueda de coste uniforme (que se diferencia del algoritmo A* en que no utiliza el valor heurístico de un nodo para determinar el siguiente nodo que se expandirá, sólo el coste para llegar al nodo en cuestión).

- `'manhattan'` para emplear la distancia de Manhattan (de la posición correspondiente al estado evaluado con respecto a la posición del objetivo).

- `'chessboard'` para emplear la distancia de Chebyshev, también conocida como distancia del tablero de ajedrez (de la posición correspondiente al estado evaluado con respecto a la posición del objetivo).

- `'euclidean'` para emplear la distancia euclídea (de la posición correspondiente al estado evaluado con respecto a la posición del objetivo).

- `'euclidean2'` para emplear el cuadrado de la distancia euclídea (de la posición correspondiente al estado evaluado con respecto a la posición del objetivo).

UCS: Búsqueda de coste uniforme

La función definida en `ucs.m` implementa una búsqueda de coste uniforme.

EJERCICIO 3: UCS [10%]
Compruebe el funcionamiento de la búsqueda de coste uniforme para las distintas configuraciones de tableros e indique en `resultados.m` el número de nodos expandidos en la búsqueda, la longitud del camino encontrado hasta la solución y su coste. Los tableros sobre los que ha de probar el funcionamiento de UCS son:

1. `frontierBoard`, de tamaño 20x20.
2. `clearBoard`, de tamaño 20x20.
3. `impossibleBoard`, de tamaño 20x20.
4. `pacmanBoard`, de tamaño 20x20.
5. `zigzagBoard`, de tamaño 20x20.
6. `costBoard`, de tamaño 20x20.
7. El tablero aleatorio creado con `randomBoard()`, de tamaño 20x20, que se encuentra en el fichero `randomBoard.mat`.

El algoritmo A*

La función definida en `astar.m` implementa el algoritmo de búsqueda A* utilizando la heurística que se le indique como segundo parámetro (el mismo parámetro que recibe la función que ya debe tener implementada en `heuristics.m`).

Los siguientes ejercicios se han de realizar, una vez más, sobre las siguientes configuraciones de tableros:

1. `frontierBoard`, de tamaño 20x20.
2. `clearBoard`, de tamaño 20x20.
3. `impossibleBoard`, de tamaño 20x20.
4. `pacmanBoard`, de tamaño 20x20.
5. `zigzagBoard`, de tamaño 20x20.
6. `costBoard`, de tamaño 20x20.
7. El tablero aleatorio creado con `randomBoard()`, de tamaño 20x20, que se encuentra en el fichero `randomBoard.mat`.

EJERCICIO 4: A* DISTANCIA DE MANHATTAN [10%]
Compruebe el funcionamiento del algoritmo A* utilizando como heurística la distancia de Manhattan e indique, en `resultados.m`, el número de nodos expandidos por el algoritmo A*, la longitud del camino encontrado hasta la solución y su coste total.

EJERCICIO 5: A* DISTANCIA DEL TABLERO DE AJEDREZ [10%]
Compruebe el funcionamiento del algoritmo A* utilizando como heurística la distancia del tablero de ajedrez e indique, en `resultados.m`, el número de nodos expandidos por el algoritmo A*, la longitud del camino encontrado hasta la solución y su coste total.

EJERCICIO 6: A* DISTANCIA EUCLÍDEA [10%]
Compruebe el funcionamiento del algoritmo A* utilizando como heurística la distancia euclídea e indique, en `resultados.m`, el número de nodos expandidos por el algoritmo A*, la longitud del camino encontrado hasta la solución y su coste total.

EJERCICIO 7: A* DISTANCIA EUCLÍDEA AL CUADRADO [10%]
Compruebe el funcionamiento del algoritmo A* utilizando como heurística la distancia euclídea al cuadrado e indique, en `resultados.m`, el número de nodos expandidos por el algoritmo A*, la longitud del camino encontrado hasta la solución y su coste total.

¿Observa algo raro al utilizar esta última heurística? ¿A qué se debe?

EJERCICIO 8: IDENTIFICACIÓN DE ALGORITMOS DE BÚSQUEDA [15%]

Las siguientes figuras corresponden a la ejecución de distintos algoritmos de búsqueda para resolver el mismo problema. Indique en `resultados.m`, mediante un número entero del 1 al 6, la figura a la que corresponde cada uno de los siguientes algoritmos:

a. Búsqueda en profundidad (DFS).
b. Búsqueda en anchura (BFS).
c. Algoritmo A* (distancia de Manhattan).
d. Algoritmo A* (distancia del tablero de ajedrez).
e. Algoritmo A* (distancia euclídea).
f. Algoritmo A* (distancia euclídea al cuadrado).

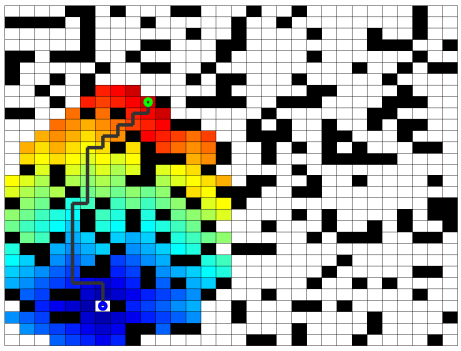


Figura 1

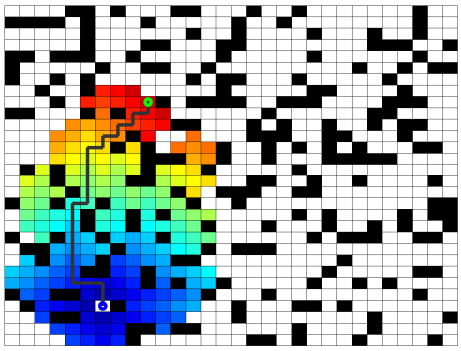


Figura 2

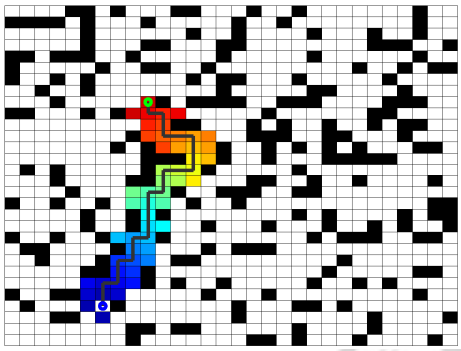


Figura 3

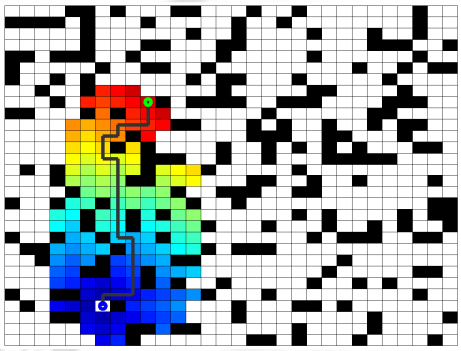


Figura 4

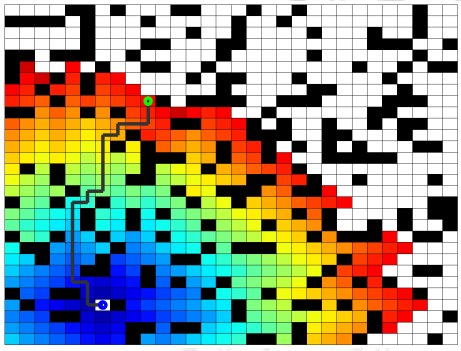


Figura 5

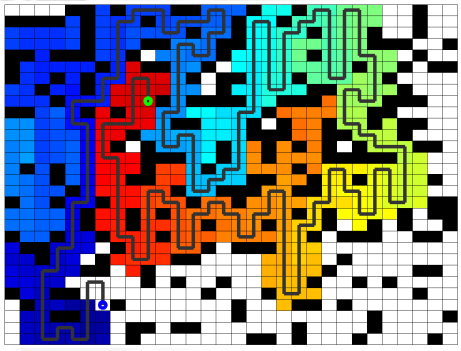
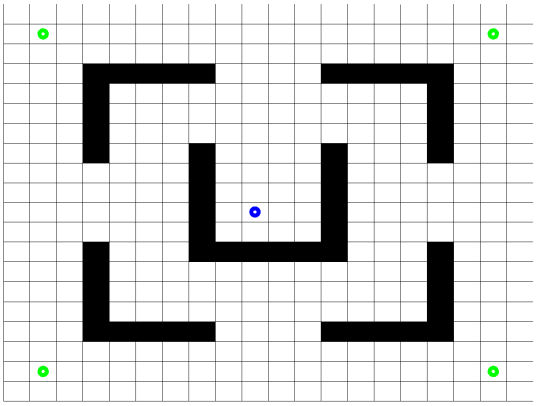


Figura 6

Otro problema de búsqueda: "Las 4 esquinas"

En la carpeta `multi` encontrará un problema de búsqueda diferente que deberá resolver realizando los mismos pasos que utilizó para resolver el problema anterior.

En este caso, implementará un agente de búsqueda que sea capaz de encontrar la ruta óptima que visita todas las casillas marcadas como objetivo a partir de una casilla inicial determinada (al estilo de un juego tipo "comecocos"):



Igual que sucedía anteriormente, las funciones definidas en los ficheros `*Board.m` se utilizan para crear distintas configuraciones de tableros sobre los que luego se efectuará el proceso de búsqueda "multiobjetivo":

`board = multiClearBoard(n);`
crea un tablero de tamaño `n`x`n`, sin obstáculos, con la casillas inicial en el centro y los objetivos que han de visitarse en las 4 esquinas del tablero.

`board = multiImpossibleBoard(n);`
crea un tablero en el que es imposible encontrar un camino que visite objeticos, ya que estos no existen en el tablero.

`board = multiPacmanBoard(n);`
crea un tablero de tamaño `n`x`n` con una configuración similar al del conocido videojuego del comecocos (`pacman` en inglés), salvo que ahora tenemos cuatro objetivos diferentes, como se puede apreciar en la figura de arriba.

La implementación de estos tableros tiene una estructura similar a la de los tableros con los que ya hemos trabajado, salvo que ahora, `board.goal` es un vector que puede incluir varios objetivos diferentes, a cuyas coordenadas puede acceder mediante las propiedades `board.goal(i).x` y `board.goal(i).y`.

Implemente las funciones necesarias para resolver el problema de búsqueda planteado. Para ello, deberá encontrar una forma adecuada de representar los estados de la búsqueda y proporcionar implementaciones válidas para las funciones:

- `initialState(board), que` crea el estado inicial de la búsqueda a partir de los datos proporcionados por el tablero `board`.
- `sameState(this, other),` que nos indicará si dos estados, `this` y `other`, representan el mismo estado de búsqueda o no.
- `finalState (board, state),` que utilizaremos para determinar si hemos alcanzado el final de la búsqueda al llegar al estado `state`.
- `nextState (board, state, action)`, la función clave que nos permite explorar el espacio de búsqueda obteniendo el estado al que se llega cuando, a partir del estado `state`, aplicamos la acción `action` en el problema de búsqueda definido por el tablero `board`.

A continuación, deberá implementar distintas funciones heurísticas **admisibles** que se puedan utilizar para resolver el problema planteado de búsqueda con múltiples objetivos:

- `'ucs'` para el algoritmo de búsqueda de coste uniforme.
- `'manhattan',` que se base en el uso de la distancia de Manhattan.
- `'chessboard'`, que utilice para emplear la distancia de Chebyshev, también conocida como distancia del tablero de ajedrez (de la posición correspondiente al estado evaluado con respecto a la posición de la casilla de salida).
- `'euclidean'` para emplear la distancia euclídea (de la posición correspondiente al estado evaluado con respecto a la posición de la casilla de salida).

Incluya su código en los ficheros `initialState.m, sameState.m, finalState.m, nextState.m` y `heuristics.m` (siempre dentro de la carpeta `multi`, para no interferir con las soluciones del problema anterior).

NOTA: Puede utilizar sus implementaciones anteriores como base para desarrollar esta parte de la práctica.

EJERCICIO 9: "LAS 4 ESQUINAS" [20%]

Resuelva el problema de las 4 esquinas para los siguientes tableros utilizando distintos algoritmos de búsqueda:

- `multiClearBoard`, de tamaño 10x10 (para apreciar las diferencias entre los distintos algoritmos de búsqueda sin tener que explorar un espacio de búsqueda demasiado grande);

- `multiImpossibleBoard`, de tamaño 20x20 (para comprobar que se explora por completo el espacio de búsqueda);

- `multiPacmanBoard`, de tamaño 20x20, para ver en la práctica cómo se realiza la exploración en un tablero similar al de un videojuego

Para cada uno de los tableros anteriores, deberá ejecutar los siguientes algoritmos de búsqueda:

- DFS (búsqueda en profundidad).

- BFS (búsqueda en anchura).

- UCS (búsqueda de coste uniforme).

- A* usando la distancia de Manhattan.

- A* usando la distancia de Chebyshev.

- A* usando la distancia euclídea.

Para cada una de las combinaciones resultantes, indique, en su fichero de resultados, `resultados.m`, el número de nodos expandidos por el algoritmo de búsqueda, la longitud del camino encontrado hasta la solución y su coste total.

EJERCICIO 10: IDENTIFICACIÓN DE ALGORITMOS DE BÚSQUEDA [5%]

Las siguientes figuras corresponden a la ejecución de distintos algoritmos de búsqueda para resolver el mismo problema de búsqueda "multiobjetivo". Indique en `resultados.m`, mediante un número entero, la figura a la que corresponde cada uno de los siguientes algoritmos:

a. Búsqueda en profundidad (DFS).

b. Algoritmo A* (distancia de Manhattan).

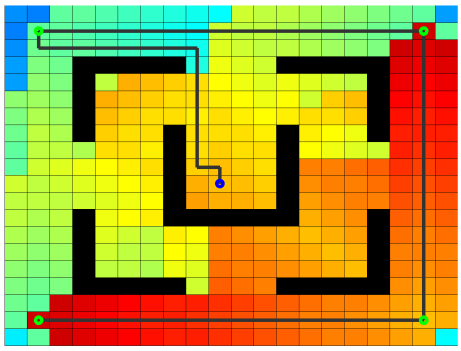


Figura 1

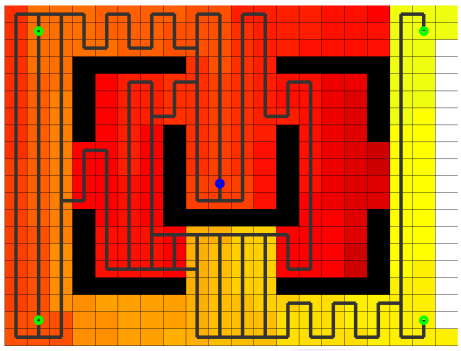


Figura 2

EVALUACIÓN DE LA PRÁCTICA

Para comprobar el funcionamiento de su implementación de la práctica, ejecute `test`, que lanza una batería automatizada de casos de prueba sobre las funciones que haya definido.

Tras remitir su fichero de resultados, se realizará una evaluación automática de los valores indicados en su fichero de resultados de la práctica, `resultados.m`.

El 50% de su calificación corresponderá a las respuestas que indique en `resultados.m`, mientras que el 50% restante provendrá de hasta qué punto satisfaga los casos de prueba definidos en `test.m`, que a su vez se reparten de la siguiente forma:

- 10% por las funciones necesarias para manejar los estados del espacio de búsqueda (`initialState`, `finalState`, `sameState`, `nextState`).
- 20% por la implementación de las distintas funciones heurísticas utilizadas en esta práctica (`ucs`, `euclidean`, `euclidean2`, `manhattan` y `chessboard`).
- 10% por el correcto funcionamiento de los algoritmos de búsqueda sin información (en profundidad, DFS, y en anchura, BFS).
- 10% por el correcto funcionamiento del algoritmo de búsqueda por coste uniforme (UCS).
- 30% por el correcto funcionamiento del algoritmo A*, que será evaluado sobre múltiples tableros de distintos tamaños y configuraciones.
- 20% final por resolver correctamente el problema de búsqueda de las 4 esquinas (o, en general, cualquier problema en el que tengamos que visitar una serie determinada de casillas, al estilo del problema del viajante de comercio) .