Actividad 1

Resolución de problema mediante búsqueda heurística

Autor: Pedro Alejandro González Morales

Resumen

En este actividad utilizaremos la estrategia de búsqueda heurística A* para generar un plan de actuación que permita al robot de Amazon mover el inventario de un estado inicial a un estado final.

Objetivo

Nuestro objetivo es que un robot lleve las estanterías con la mercancía desde un punto de origen hasta un punto de destino donde un operario humano recogerá los productos.

El almacén podemos representarlo como una rejilla donde en cada casilla colocaremos *objetos*. En este caso serán: la mercancía (**M**), nuestro robot (**R**), o las paredes (**#**). Estos objetos los podemos localizar en todo momento mediante un sitema de coordenadas (**X**,**Y**) donde **X** corresponde a las filas e **Y** a las columnas, ver Fig. 1.

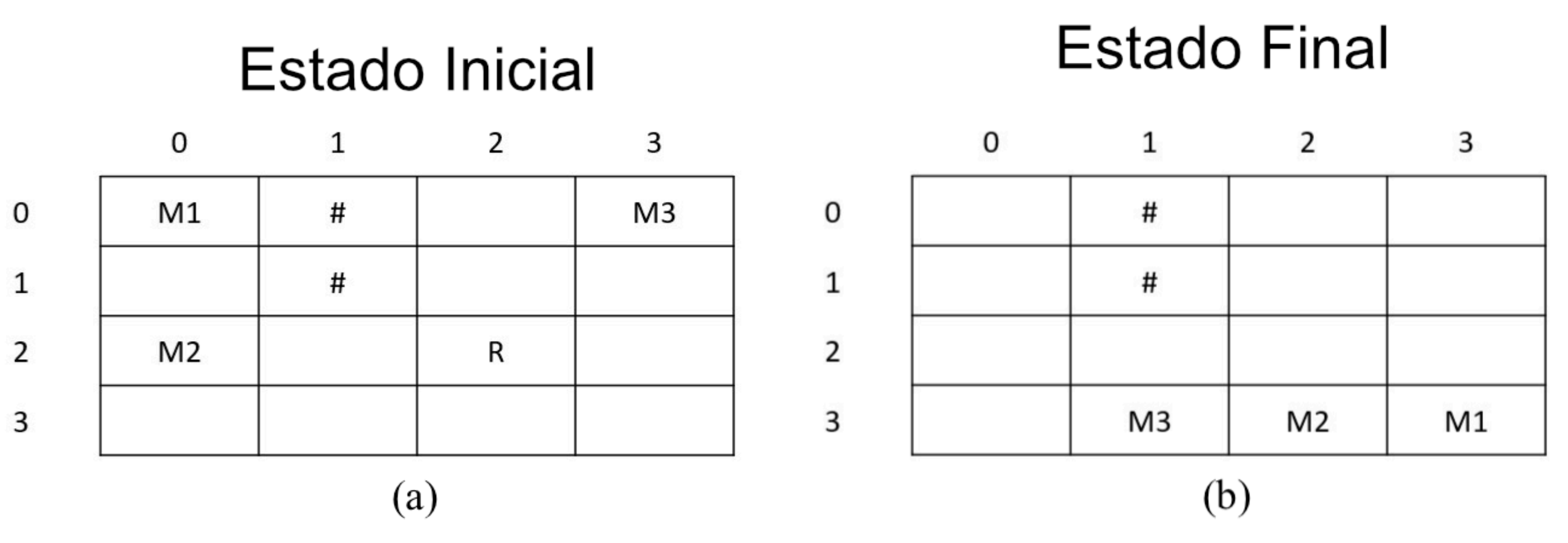

Figura 1. *Representación esquemática de nuestro almacén indicando el estado inicial (a) y el estado final (b).*

Con esta notación nuestras estanterías en el estado inicial estarán localizadas en las coordenadas M1=(0,0), M2=(2,0) y M3=(0,3) y el robot en R=(2,2), ver Fig 1a. El estado final vendrá dado por las posiciones M1=(3,3), M2=(3,2) y M3=(3,1), ver Fig 1b.

Los posibles movimientos del robot estarán limitados por el diseño del almacén y las posiciones de nuestras estanterías. Por simplicidad vamos a considerar que:

1. El robot no podrá moverse a posiciones ocupadas por una pared.
2. Si el robot no tiene carga podrá moverse a posiciones ocupadas por estanterías para alcanzar la posición objetivo o realizar la carga de dicha estantería.
3. Si el robot está cargado no podrá moverse a una posición ocupada por una estantería.
4. El robot no podrá moverse en las direcciones diagonales debido a que las estanterías ocupan un volumen y podría chocar con otras estanterías (o paredes).

Código

A continuación se adjunta el código fuente propuespo para mover al robot así como una breve descripción de las funciones auxiliares definidas y clases. Este código está basado en el algoritmo de Swift (2017).

Clases

- **Nodo**: Clase con las propiedades de un nodo

Funciones auxiliares

- **size**: calcula el número de casillas que tiene la malla para usarlo como límite superior en el bucle encargado de la búsqueda del camino óptimo mediante el algoritmo A* y así evitar un bucle infinito en caso de encontrar solución.
- **updateMalla**: actualiza la malla colocando nuevos obstáculos, en nuestro caso, las estanterías en la posición de destino.
- **minimo**: dada una coordenada de origen, calcula el camino hasta varios destinos y devuelve el camino más corto como aquel en el que se realiza el menor número de pasos hasta el destino.
- **imprime**: imprime por pantalla los movimientos del robot de una forma legible por el usuario
- **coste**: calcula el coste del movimiento. La función por defecto calcula el coste como la distancia entre dos puntos. El parámetro opcional *metodo* permite calcular esta distancia mediante dos formas distintas. Si no se especifica, el valor por defecto es "*manhatan*" y la distancia es calculada mediante la norma-l_1 también conocida como distancia Manhattan. Si es especificado con el valor "*euclides*" el coste se calculará como la distancia euclidiana (**p**^2+**q**^2).
- **movPermitidpo**: devuelve una lista formada por tuplas. Estas tuplas contienen los movimientos permitidos desde el nodo padre hacia los hijos en coordenadas relativas al nodo padre. El parámetro *permit* ofrece la posibilidad de permitir movimientos en las diagonales o no. Por defecto *permit* tiene el valor "*NoDiagonal*" y la función devuelve los movimientos permitidos sin tener en cuenta las las diagonales. Si se especifica el parámetro *permit* como "*Diagonal*", las posiciones permitidas incluirán además a los vecinos situados en las diagonales.

In [1]:

```python
class Nodo():
    """Clase 'Nodo' para el buscador A-estrella"""

    def __init__(self, padre=None, posicion=None):
        self.padre    = padre
        self.posicion = posicion

        self.g = 0
        self.h = 0
        self.f = 0

    def __eq__(self, otro):
        return self.posicion == otro.posicion

def size(malla):
    """Calcula el numero de casillas en la malla"""
    filas = len(malla)
    colum = len(malla[0])
    return filas*colum

def updateMalla(malla,posicion):
    """ Actuliza la malla colocando los nuevos obstaculos"""
    x=posicion[0]
    y=posicion[1]
    malla[x][y] = 1
    return malla

def minimo(malla,inicio,fin):
    """Calcula el camino más corto desde una coordenada origen
    a varios destinos como el camino con menos pasos"""
    n = len(fin)
    n_mov = []
    for i in range(n):
        try:
            path = astar(malla, inicio, fin[i])
            n_mov.append(len(path))
        except RuntimeError:
            n_mov.append(1e10) # Si nos pasamos de iteraciones hacemos el camino muy grande
            continue

    result = n_mov.index(min(n_mov))
    if n_mov[result] == 1e10:
        raise RuntimeError("No encuentro mínimo")

    return result

def imprime(path):
    """Sacamos por pantalla el moviemiento del robot"""
    for pos in path[1:]:
        print(" - mover robot a fila %s, columna %s"%pos)

def coste(nodo_ini,nodo_fin,metodo='manhattan'):
    """Calculamos el coste del movimiento"""
    if metodo == 'manhattan':
        d = abs(nodo_ini.posicion[0] - nodo_fin.posicion[0]) + abs(nodo_ini.posicion[1] - nodo_fin.posicion[1]) # Manhattan
    elif metodo == 'euclides':
        d = (nodo_ini.posicion[0] - nodo_fin.posicion[0]) ** 2 + (nodo_ini.posicion[1] - nodo_fin.posicion[1]) ** 2 # Euclidea
    return d

def movPermitido(permit="NoDiagonal"):
    """Devuelve una lista con las posiciones permitidas en el movimiento"""
    if permit == "NoDiagonal":
        m = [(0, -1), (0, 1), (-1, 0), (1, 0)]  # Diagonales NO permitidas
    elif permit == "Diagonal":
        m = [(0, -1), (0, 1), (-1, 0), (1, 0),
            (-1, -1), (-1, 1), (1, -1), (1, 1)] # Diagonales permitidas
    return m
```

Algoritmo A*

El algoritmo A* fue propuesto por Hart et al. (1968) como una extensión y mejora del algoritmo propuesto por Dijkstra (1959). Mientras que algoritmo Dijkstra realiza la búsqueda a coste uniforme, A* se basa en el cálculo del coste total F necesario para realizar un camino desde un nodo inicio hasta un nodo final haciendo uso de una *heurística*. Esta función F se puede escribir entonces como

$$F = G + H,$$

donde G es el coste necesario para moverse desde el punto de origen hasta el nodo actual y H es el coste necesario para moverse desde el nodo actual hasta el destino. Este coste H es la denominada *heurística* ya que no lo conocemos *a priori* pero lo podemos estimar de varias maneras. Un forma conveniente de estimar esta función heurística es mediante la distancia Manhattan o norma-l_1.

Norma-l_1 o distantcia Manhattan.

Esta métrica, también conocida como *Geometría del taxi* (2020), fue propuesta por Minkowski a finales del siglo XIX para medir la distancia entre dos puntos donde la geometría Euclidiana es reemplazada por la distancia l_1. Esta distancia viene dada como la suma de las longitudes proyectadas del segmento de linea entre dos puntos sobre los ejes coordenados, esto es

$$l_1(\mathbf{p},\mathbf{q}) = ||\mathbf{p}-\mathbf{q}||_1 = \sum_{i=1}^{n} |p_i - q_i|$$

donde $(\mathbf{p},\mathbf{q})$ son los vectores n-dimensionales $\mathbf{p} = (p_1, p_2, \ldots, p_n)$ y $\mathbf{q} = (q_1, q_2, \ldots, q_n)$. De esta manera, para nuestro algoritmo de búsqueda en un plano, la distancia entre dos puntos (p_1, p_2) y (q_1, q_2) vendrá dada por $|p_1 - q_1| + |p_2 - q_2|$.

Pseudocódigo

Los pasos del algoritmo los podemos resumir siguiendo las indicaciones de Lester (2005):

1. Añade el punto inicial a la lista abierta
2. Repetir lo siguiente:
    - Mira en la lista abierta la posición con menor coste F. Nos referimos a esta como posición actual.
    - Cámbialo a la lista cerrada.
    - Para cada uno de los 8 vecinos de la posición actual.
        - Si no es accesible o no está en la lista cerrada, ignoralo. De lo contrario:
            - Si no está en la lista abierta, añádelo a ella. Haz la posición actual "padre" de dicha posición. Guarda los valores de coste F, G y H.
            - Si está en la lista abierta, comprueba que el camino hasta la posisión es mejor usando el valor del coste G para ello. Un menor valor de G significa que es mejor camino. En este caso, cambia el "padre" de dicha posición a la posición actual y recalcula los valores de G y F.
    - Para cuando:
        - Se añade una posición a la lista cerrada, esto es, se encuentra el camino.
        - No encuentras una posición objetivo y la lista abierta queda vacía. En este caso no hay camino.
3. Salva el camino. Recorriéndolo en sentido inverso, esto es, desde el objetivo hasta el inicio, tendremos el camino.

In [2]:

```python
def astar(malla, inicio, fin):
    """Devuelve una lista de tuplas con el camino desde un punto
    de origen a un destino en la malla dada"""

    # Movimientos permitidos para el desplazamiento
    # el argumento puede ser "NoDiagonal" o "Diagonal"
    movimientos = movPermitido("NoDiagonal")

    # Creamos el nodo inicial y final
    nodo_ini = Nodo(None, inicio)
    nodo_ini.g = nodo_ini.h = nodo_ini.f = 0
    nodo_fin = Nodo(None, fin)
    nodo_fin.g = nodo_fin.h = nodo_fin.f = 0

    # Iniciamos las listas abierta y cerrada
    lista_abierta = []
    lista_cerrada = []

    # Añadimos el nodo de inicio a la lista abierta
    lista_abierta.append(nodo_ini)

    # Bucle principal. Se ejecuta hasta que encuentre un destino o
    # hasta que el número de pasos sea mayor que
    # el número de posiciones en la malla.
    count = 0
    max_itera=size(malla)*size(malla)
    while len(lista_abierta) > 0:

        #print(size(malla),count,len(lista_cerrada))

        # Si alcanzamos un número de iteraciones máximo
        # salimos del bucle.
        if count > max_itera:
            raise RuntimeError("Demasiadas iteraciones")

        # Obtenermos el nodo actual
        nodo_actual = lista_abierta[0]
        indice_actual = 0
        for ind, item in enumerate(lista_abierta):
            if item.f < nodo_actual.f:
                nodo_actual = item
                indice_actual = ind

        # Sacamos el nodo actual de la lista abierta y lo añadimos
        # a la lista cerrada
        lista_abierta.pop(indice_actual)
        lista_cerrada.append(nodo_actual)

        # Reconstruimos el camino en el caso de haber llegado al nodo final
        if nodo_actual == nodo_fin:
            path = []
            actual = nodo_actual
            while actual is not None:
                path.append(actual.posicion)
                actual = actual.padre
            return path[::-1] # Retornamos el camino en orden inverso

        # Generamos los hijos, esto es las posibles posiciones de movimiento
        hijos = []
        for index,posicion_nueva in enumerate(movimientos):

            # Tupla con la posición del Nodo hijo
            posicion_nodo = (nodo_actual.posicion[0] + posicion_nueva[0],
                             nodo_actual.posicion[1] + posicion_nueva[1])

            # Comprobamos que el nodo está dentro del rango de movimientos, esto es,
            # que está dentro de la malla.
            if (posicion_nodo[0] > (len(malla) - 1) or
                posicion_nodo[0] < 0 or
                posicion_nodo[1] > (len(malla[len(malla)-1]) -1) or
                posicion_nodo[1] < 0 ):
                continue

            # Nos aseguramos que la posición es accesible, es decir, no está
            # ocupada por paredes o estanterías
            if malla[posicion_nodo[0]][posicion_nodo[1]] != 0:
                continue

            # Creamos el nuevo nodo
            nodo_nuevo = Nodo(nodo_actual, posicion_nodo)

            # Añadimos el nuevo nodo a la lista de hijos
            hijos.append(nodo_nuevo)


        # Barremos la lista de hijos
        for hijo in hijos:

            # Comprobamos si el Hijo está en la lista cerrada.
            for hijo_cerrado in lista_cerrada:
                if hijo == hijo_cerrado:
                    continue

            # Calculamos los valores de coste F, G, y H
            hijo.g = nodo_actual.g + 1
            hijo.h = coste(hijo,nodo_fin,metodo="manhattan")
            hijo.f = hijo.g + hijo.h

            # Comprobamos si el Hijo está en la lista abierta
            for nodo_abierto in lista_abierta:
                if hijo == nodo_abierto and hijo.g > nodo_abierto.g:
                    continue

            # Si el hijo no esta en la lista cerrada, esta en la lista abierta
            # y tiene el menor coste G lo añadimos a la lista abierta
            lista_abierta.append(hijo)

        # Aumentamos en uno el contador de vueltas
        count=count+1
    #return None
    #raise ValueError('No encuentro camino')
```

Llamada principal

El código requiere definir unas variables:

- **malla**: lista con la configuración del almacen sin las estanterías. *0* denota espacio libre y *1* denota una pared.
- **robot**: tupla con la posición inicial del robot dentro del almacén en el formato (fila,columna).
- **carga_ini**: lista con las coordenadas del estado inicial para cada estantería en el formato (fila,columna).
- **carga_fin**: lista con las coordenadas del estado final para cada estantería en el formato (fila,columna).

Las dimensiones de las listas *cargas_ini* y *cargas_fin* deben coincidir.

In [3]:

```python
def main(cargas_ini,cargas_fin,robot,malla):

    if len(carga_ini) != len(carga_fin):
        raise ValueError("Revisa las dimensiones de las listas de cargas")

    # --< Bucle principal >--
    #
    # - Comenzamos buscando el camino más corto desde el robot a las estanterías.
    # - Desplazamos el robot hasta la estantería más próxima y la cargamos.
    # - Desplazamos la carga hasta el destino.
    # - Si hay más estanterías a desplazar repitimos el bucle.
    #
    malla0 = malla.copy()
    cargas = []
    for i in range(len(carga_ini)):
        cargas.append("M%s" % (i+1))

    while len(cargas) > 0:
        # Intentamos encontrar el camino más corto hasta la carga
        try:
            m = minimo(malla0,robot,carga_ini)
        except RuntimeError:
            print("No encuentro camino a las cargas")
            break
        # Si encontramos camino movemos el robot
        path = astar(malla0, robot, carga_ini[m])
        print("Mueve Robot a la carga %s:" % cargas[m])
        #print(path)
        imprime(path)
        print("Carga %s"%cargas[m])
        # Intentamos mover el robot al destino
        robot = carga_ini[m]
        try:
            path = astar(malla, robot, carga_fin[m])
        except RuntimeError:
            print("No encuentro camino a destino %s: "%cargas[m])
            # Actualizamos variables: Sacamos esa carga de la lista
            carga_ini.pop(m)
            carga_fin.pop(m)
            cargas.pop(m)
            continue
        # Si hemos podido mover el robot a destino continuamos
        print("Mueve Robot a destino %s:"%cargas[m])
        #print(path)
        imprime(path)
        print("Descarga %s"%cargas[m])
        # Actualizamos variables
        robot = carga_fin[m]
        malla = updateMalla(malla,carga_fin[m])
        carga_ini.pop(m)
        carga_fin.pop(m)
        cargas.pop(m)
    print("Fin")
```

Ejecución

A continuación se ofrecen varios ejemplos de ejecución para diferentes configuaraciones.

Caso 1: configuración problema

In [4]:

```python
# Configuracion del almacén.
malla = [[0, 1, 0, 0],
         [0, 1, 0, 0],
         [0, 0, 0, 0],
         [0, 0, 0, 0]]
# Posición inicial del robot
robot = (2,2)
# Estados inicial y final
carga_ini = [(0,0),(2,0),(0,3)]
carga_fin = [(3,3),(3,2),(3,1)]
# Llamada principal
main(carga_ini,carga_fin,robot,malla)
```

```
Mueve Robot a la carga M2:
 - mover robot a fila 2, columna 1
 - mover robot a fila 2, columna 0
Carga M2
Mueve Robot a destino M2:
 - mover robot a fila 2, columna 1
 - mover robot a fila 2, columna 2
 - mover robot a fila 3, columna 2
Descarga M2
Mueve Robot a la carga M3:
 - mover robot a fila 3, columna 3
 - mover robot a fila 2, columna 3
 - mover robot a fila 1, columna 3
 - mover robot a fila 0, columna 3
Carga M3
Mueve Robot a destino M3:
 - mover robot a fila 0, columna 2
 - mover robot a fila 1, columna 2
 - mover robot a fila 2, columna 2
 - mover robot a fila 2, columna 1
 - mover robot a fila 3, columna 1
Descarga M3
Mueve Robot a la carga M1:
 - mover robot a fila 3, columna 0
 - mover robot a fila 2, columna 0
 - mover robot a fila 1, columna 0
 - mover robot a fila 0, columna 0
Carga M1
Mueve Robot a destino M1:
 - mover robot a fila 1, columna 0
 - mover robot a fila 2, columna 0
 - mover robot a fila 2, columna 1
 - mover robot a fila 2, columna 2
 - mover robot a fila 2, columna 3
 - mover robot a fila 3, columna 3
Descarga M1
Fin
```

Caso 2: bloqueo tras mover carga

In [5]:

```python
# Configuracion del almacén.
malla = [[0, 1, 0, 0],
         [0, 1, 0, 0],
         [0, 1, 0, 0],
         [0, 0, 0, 0]]
# Posición inicial del robot
robot = (2,2)
# Estados inicial y final
carga_ini = [(0,0),(2,0),(0,3)]
carga_fin = [(3,3),(3,2),(3,1)]
# Llamada principal
main(carga_ini,carga_fin,robot,malla)
```

```
Mueve Robot a la carga M3:
 - mover robot a fila 2, columna 3
 - mover robot a fila 1, columna 3
 - mover robot a fila 0, columna 3
Carga M3
Mueve Robot a destino M3:
 - mover robot a fila 0, columna 2
 - mover robot a fila 1, columna 2
 - mover robot a fila 2, columna 2
 - mover robot a fila 3, columna 2
 - mover robot a fila 3, columna 1
Descarga M3
Mueve Robot a la carga M2:
 - mover robot a fila 3, columna 0
 - mover robot a fila 2, columna 0
Carga M2
No encuentro camino a destino M2:
Mueve Robot a la carga M1:
 - mover robot a fila 1, columna 0
 - mover robot a fila 0, columna 0
Carga M1
No encuentro camino a destino M1:
Fin
```

Caso 3: dos cargas bloqueadas una libre

In [6]:

```python
# Configuracion del almacén.
malla = [[0, 1, 0, 0],
         [0, 1, 0, 0],
         [0, 1, 0, 0],
         [0, 1, 0, 0]]
# Posición inicial del robot
robot = (2,2)
# Estados inicial y final
carga_ini = [(0,0),(2,0),(0,3)]
carga_fin = [(3,1),(3,2),(3,3)]
# Llamada principal
main(carga_ini,carga_fin,robot,malla)
```

```
Mueve Robot a la carga M3:
 - mover robot a fila 2, columna 3
 - mover robot a fila 1, columna 3
 - mover robot a fila 0, columna 3
Carga M3
Mueve Robot a destino M3:
 - mover robot a fila 1, columna 3
 - mover robot a fila 2, columna 3
 - mover robot a fila 3, columna 3
Descarga M3
No encuentro camino a las cargas
Fin
```

Caso 4: cargas bloqueadas

In [7]:

```python
# Configuracion del almacén.
malla = [[0, 1, 0, 0],
         [0, 1, 0, 0],
         [0, 1, 0, 0],
         [0, 1, 0, 0]]
# Posición inicial del robot
robot = (2,2)
# Estados inicial y final
carga_ini = [(0,0),(2,0)]
carga_fin = [(3,1),(3,2)]
# Llamada principal
main(carga_ini,carga_fin,robot,malla)
```

```
No encuentro camino a las cargas
Fin
```

Dificultades encontradas

El código planteado para resolver el problema podría mejorarse implementando la posibilidad de que el robot se pueda mover en diagonal si tanto la posición horizontal como la vertical, en la dirección del movimiento óptimo, están vacías simultáneamente.

Otra mejora a tener en cuenta son los posibles bloqueos en la colocación de estanterías. Por ejemplo nuestro código no es capaz de resolver el Caso 2, es decir, no contempla mover temporalmente M3 para llegar a la solución o diseñar una estado intermedio donde almacenar las estanterías para luego llegar a la solución.

Referencias

Dijikstra, E. W. (1959). *A Note on Two Problems in Connexion with Graphs*. Numerische Mathematik, (1):269–271.

Geometría del taxista. (2020). Wikipedia, la enciclopedia libre. Recuperado el 27 de abril de 2021 de https://es.wikipedia.org/wiki/Geometr%C3%ADa_del_taxista

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). *A formal basis for the heuristic determination of minimum cost paths.* IEEE Transactions on Systems Science and Cybernetics, 4(2):100–107.

Lester, P. (2005). *A$* $PathfindingforBeginners$. Recuperado el 27 de abril de 2021 de http://csis.pace.edu/~benjamin/teaching/cs627/webfiles/Astar.pdf.

Swift, N. (2017). *A$$* pathfinding algorithm.* Recuperado el 27 de abril de 2021 de https://gist.github.com/Nicholas-Swift/003e1932ef2804bebef2710527008f44