Planeación de rutas para la exploración en Marte

Juan Manuel Hernández Solano, José Carlos Yamuni Contreras, Miguel Steven Nguyen

March 14, 2023

1 Planteamiento del problema

[1]:
```
import numpy as np
import math
from simpleai.search import SearchProblem, breadth_first, depth_first,␣
 ↪uniform_cost, astar
```

Despues de procesar el archivo `mars_map.IMG`, obtenemos un archivo de numpy, que podemos leer de forma más facil.

[2]:
```
mars_map = np.load('mars_map.npy')
NR, NC = mars_map.shape
SCALE = 10.0174
```

Aquí, definimos algunas funciones que facilitan la converción de coordenadas (x, y) a coordenadas dentro de la matríz.

[3]:
```
def get_row(nr: int, y: int, scale: float) -> int:
    """
    Returns the row number inside the mars map for a given 'y'
    coordinate.
    ---
    nr: total number of rows inside the mars map
    y: the y coordinate we want to find
    scale: the scale at which the map is scaled
    ---
    Returns -> int representing the row at which the 'y' coordinate is found
    """
    return nr - round(y / scale)


def get_column(x: int, scale: float) -> int:
    """
    Returns the column number inside the mars map for a given 'x'
    coordinate.
    ---
    x: the x coordinate we want to find
    scale: the scale at which the map is scaled
```

```
    ---
    Returns -> int representing the row at which the 'x' coordinate is found
    """
    return round(x / scale)
```

Definimos nuestro problema similar al problema del laberinto.

[4]:
```
# Defining Search Problem
class Rover(SearchProblem):
    def __init__(self, mars_map, start_point, end_point):
        self.map = mars_map
        self.initial_state = (
            get_row(NR, start_point[1], SCALE),
            get_column(start_point[0], SCALE)
        )
        self.goal_position = (
            get_row(NR, end_point[1], SCALE),
            get_column(end_point[0], SCALE),
        )

    def actions(self, state):
        open_sides = []
        row = state[0]
        column = state[1]

        current_height = self.map[row][column]

        # UL
        if self.map[row - 1][column - 1] != -1 and abs(self.map[row - 1][column␣
↪- 1] - current_height) < .25:
            open_sides.append((row - 1, column - 1))
        # UU
        if self.map[row - 1][column] != -1 and abs(self.map[row - 1][column] -␣
↪current_height) < .25:
            open_sides.append((row - 1, column))
        # UR
        if self.map[row - 1][column + 1] != -1 and abs(self.map[row - 1][column␣
↪+ 1] - current_height) < .25:
            open_sides.append((row - 1, column + 1))
        # LL
        if self.map[row][column - 1] != -1 and abs(self.map[row][column - 1] -␣
↪current_height) < .25:
            open_sides.append((row, column - 1))
        # RR
        if self.map[row][column + 1] != -1 and abs(self.map[row][column + 1] -␣
↪current_height) < .25:
            open_sides.append((row, column + 1))
```

```
        # DL
        if self.map[row + 1][column - 1] != -1 and abs(self.map[row + 1][column
↪- 1] - current_height) < .25:
            open_sides.append((row + 1, column - 1))
        # DD
        if self.map[row + 1][column] != -1 and abs(self.map[row + 1][column] -
↪current_height) < .25:
            open_sides.append((row + 1, column))
        # DR
        if self.map[row + 1][column + 1] != -1 and abs(self.map[row + 1][column
↪+ 1] - current_height) < .25:
            open_sides.append((row + 1, column + 1))

        return open_sides

    def result(self, state, action):
        state = action
        return state

    def is_goal(self, state):
        return state == self.goal_position

    def heuristic(self, state):
        return math.sqrt(
            ((self.goal_position[0] - state[0]) ** 2) + ((self.goal_position[1]
↪- state[1]) ** 2)
        )
```

Nuestro punto de partida, y nuestro objectivo.

[5]:
```
# Starting and ending points
start_x = 2850
start_y = 6400
end_x = 3150
end_y = 6800
```

2 Resultados

Lo que queremos calcular con cada método es la distancia total recorrida por el robot, asi que definimos una función para hacer este cálculo.

[6]:
```
from typing import List
```

[7]:
```
def calculate_distance(points: List) -> float:
    """
    Calculates the distance traveled from a list of points.
```

```
    """
    distance = 0
    for i in range(len(points) - 1):
        x1 = points[i][1][0]
        x2 = points[i + 1][1][0]
        y1 = points[i][1][1]
        y2 = points[i + 1][1][1]

        distance += math.sqrt(((x2 - x1) ** 2) + ((y2 - y1) ** 2))
    return distance * SCALE  # Multiplied by scale to return to meters
```

[8]:
```
problem = Rover(mars_map=mars_map,
                start_point=(start_x, start_y),
                end_point=(end_x, end_y))
```

2.0.1 Breadth First

[9]:
```
breadth_result = breadth_first(problem, graph_search=True)
```

[10]:
```
print(breadth_distance := calculate_distance(breadth_result.path()))
```

```
3587.4296957056717
```

2.0.2 Depth First

[11]:
```
# depth_result = depth_first(problem, graph_search=True)
```

La búsqueda de 'depth_first' no da resultados a tiempo.

2.0.3 Uniform Cost

[12]:
```
uniform_result = uniform_cost(problem, graph_search=True)
```

[13]:
```
print(uniform_distance := calculate_distance(uniform_result.path()))
```

```
3612.3257533439705
```

2.0.4 A*

[14]:
```
astar_result = astar(problem, graph_search=True)
```

[15]:
```
print(astar_distance := calculate_distance(astar_result.path()))
```

```
3512.7415227907786
```

3 Visualización

Para ver la ruta que tomó cada algoritmo, necesitamos limpiar los datos primero.

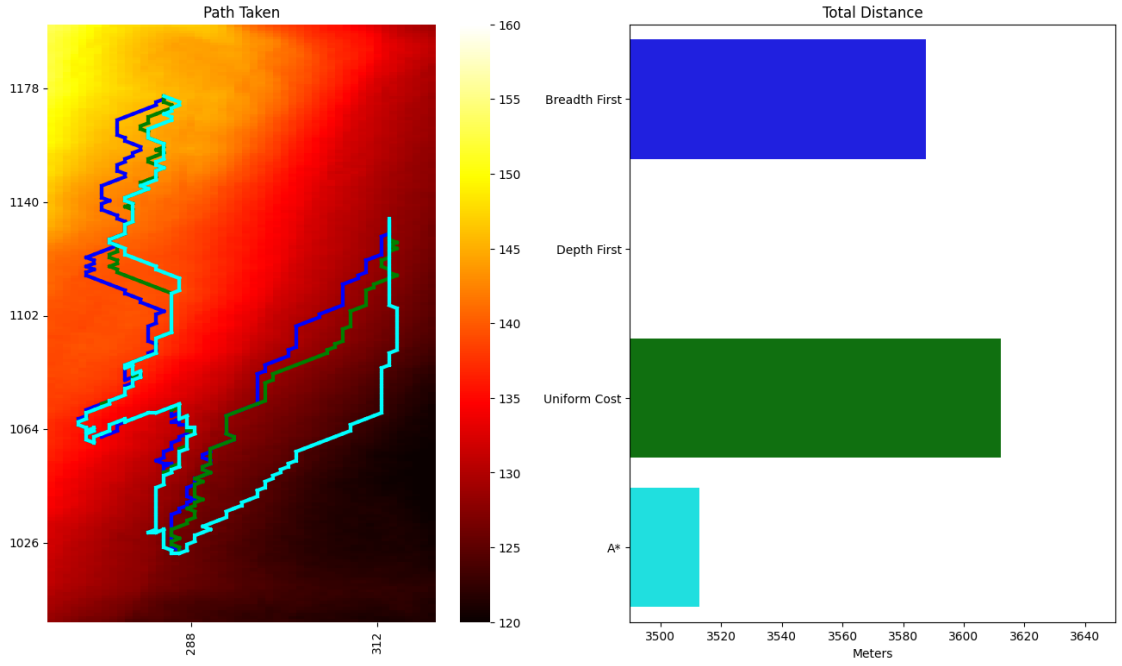
[16]:
```
# Function to create list of points
def get_points(points: List) -> List:
    """
    From the result of the search, we want a nicer, cleaner list
    """
    clean_points = []
    for i in range(len(points) - 1):
        x = points[i][1][1]
        y = points[i][1][0]

        clean_points.append((x, y))

    return clean_points
```

[17]:
```
breadth_points = get_points(breadth_result.path())
uniform_points = get_points(uniform_result.path())
astar_points = get_points(astar_result.path())
```

[18]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
```

[19]:
```
fig = plt.figure(figsize=(16, 9))

# Map and paths taken
ax1 = fig.add_subplot(121)
ax1.grid(visible=True)
ax1.set_title("Path Taken")

# Surface map
sns.heatmap(mars_map, cmap='hot', vmin=120, vmax=160, ax=ax1)
ax1.set_xlim(270, 320)
ax1.set_ylim(1000, 1200)

# Breadth Path
for i in range(len(breadth_points) - 1):
    plt.plot([breadth_points[i][0], breadth_points[i + 1][0]],
             [breadth_points[i][1], breadth_points[i + 1][1]],
             color='blue', lw=3)

# Uniform Path
for i in range(len(uniform_points) - 1):
    plt.plot([uniform_points[i][0], uniform_points[i + 1][0]],
             [uniform_points[i][1], uniform_points[i + 1][1]],
```

```
            color='green', lw=3)

# A* Path
for i in range(len(astar_points) - 1):
    plt.plot([astar_points[i][0], astar_points[i + 1][0]],
             [astar_points[i][1], astar_points[i + 1][1]],
             color='cyan', lw=3)


# Results
colors = ['blue', 'blue', 'green', 'cyan']
ax2 = fig.add_subplot(122)
ax2.set_xlim(3490, 3650)
ax2.set_title("Total Distance")
ax2.set_xlabel("Meters")
sns.barplot(
    x=[breadth_distance, 0, uniform_distance, astar_distance],
    y=["Breadth First", "Depth First", "Uniform Cost", "A*"],
    ax=ax2, palette=colors
);
```



Para las condiciones dadas inicialmente, podemos ver que casi todas llegan al objetivo, menos el de 'depth first'. Para recorridos cortos, no es necesario utilizar una búsqueda informada. En situaciones en donde sí se necesiten, la mejor heuristica será la distancia euclidiana entre el punto de inicio y la meta.

4 Condiciones Iniciales Distintas

Ahora, probaremos otras puntos iniciales y finales para ver el rendimiento del algoritmo de 'astar'.

4.1 500 Metros

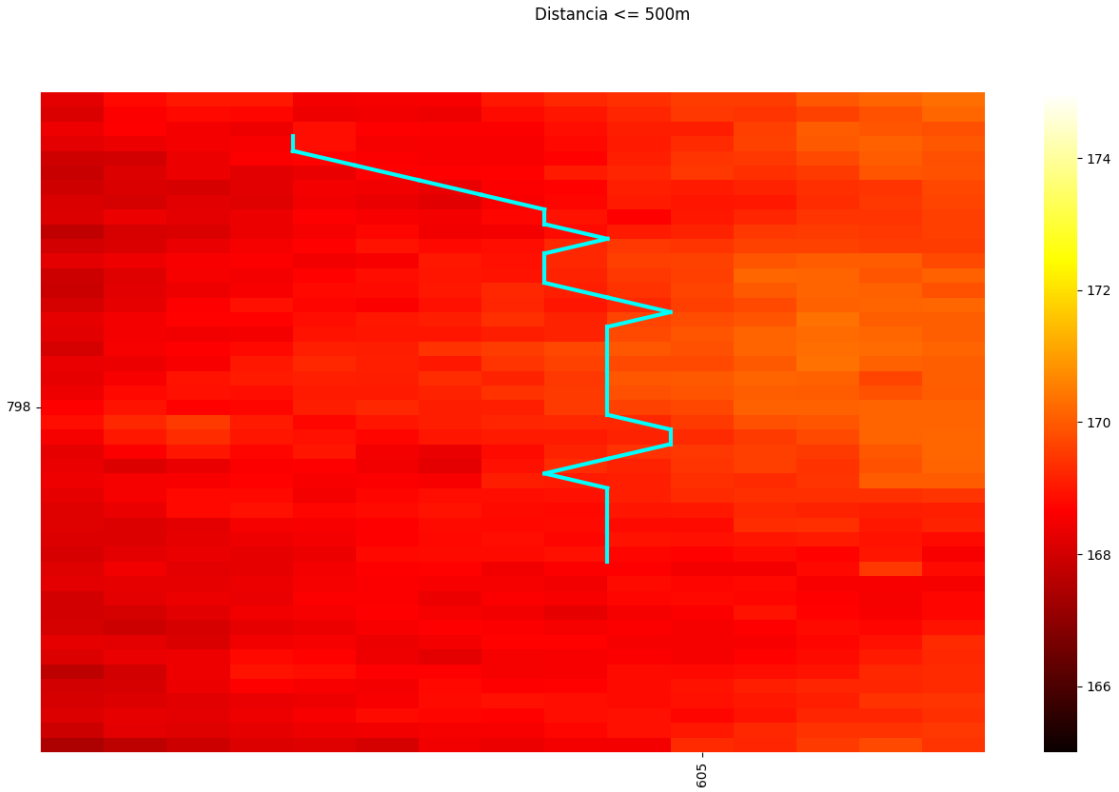
[20]:
```
# Starting and ending points under 500m apart
start_x_500 = 6000
start_y_500 = 10000
end_x_500 = 6050
end_y_500 = 10300
problem = Rover(mars_map=mars_map,
                start_point=(start_x_500, start_y_500),
                end_point=(end_x_500, end_y_500))
```

[21]:
```
astar_500_result = astar(problem, graph_search=True)
```

[22]:
```
astar_500_distance = calculate_distance(astar_500_result.path())
print(f"Total distance: {astar_500_distance}")
astar_500_points = get_points(astar_500_result.path())
```

```
Total distance: 354.4634582163112
```

[23]:
```
fig = plt.figure(figsize=(16, 9))
ax = fig.add_subplot(111)
sns.heatmap(mars_map, cmap='hot', vmin=165, vmax=175, ax=ax)
ax.set_xlim(595, 610)
ax.set_ylim(775, 820)
fig.suptitle('Distancia <= 500m')

# A* Path
for i in range(len(astar_500_points) - 1):
    plt.plot([astar_500_points[i][0], astar_500_points[i + 1][0]],
             [astar_500_points[i][1], astar_500_points[i + 1][1]],
             color='cyan', lw=3)
```



4.2 Distancia entre 1000m y 5000m

[24]:
```
# Starting and ending points over 1000m apart
start_x_1000 = 7000
start_y_1000 = 2500
end_x_1000 = 6000
end_y_1000 = 900
```

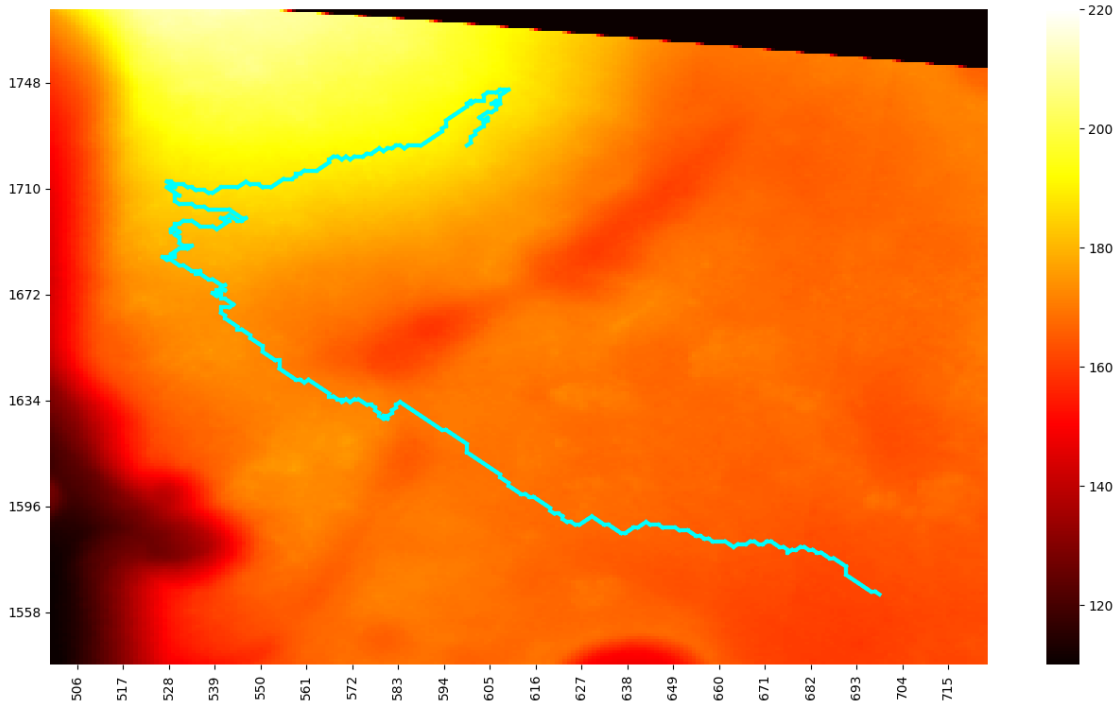
[25]:
```
problem = Rover(mars_map=mars_map,
                start_point=(start_x_1000, start_y_1000),
                end_point=(end_x_1000, end_y_1000))
```

[26]:
```
astar_1000_result = astar(problem, graph_search=True)
```

[27]:
```
astar_1000_distance = calculate_distance(astar_1000_result.path())
print(f"Total distance: {astar_1000_distance}")
astar_1000_points = get_points(astar_1000_result.path());
```

```
Total distance: 5033.149390255321
```

[28]:
```
fig = plt.figure(figsize=(16, 9))
ax = fig.add_subplot(111)
sns.heatmap(mars_map, cmap='hot', vmin=110, vmax=220, ax=ax)
ax.set_xlim(500, 725)
ax.set_ylim(1540, 1775)

# A* Path
for i in range(len(astar_1000_points) - 1):
    plt.plot([astar_1000_points[i][0], astar_1000_points[i + 1][0]],
             [astar_1000_points[i][1], astar_1000_points[i + 1][1]],
             color='cyan', lw=3)
```



4.3 Distancia mayor a 10000

[29]:
```
# Starting and ending points over 10000m apart
start_x_10000 = 7500
start_y_10000 = 700
end_x_10000 = 7600
end_y_10000 = 10700
```

[30]:
```
problem = Rover(mars_map=mars_map,
                start_point=(start_x_10000, start_y_10000),
                end_point=(end_x_10000, end_y_10000))
```

[31]:
```
astar_10000_result = astar(problem, graph_search=True)
```

[32]:
```
try:
    astar_10000_distance = calculate_distance(astar_10000_result.path())
    astar_10000_points = get_points(astar_10000_result.path())
except AttributeError:
    print("No se encontró una ruta")
```

```
No se encontró una ruta
```

5 Concluciones

Este problema fue planteado de forma muy similar al de un laberinto, ya que en el mapa topográfico que representa la superficie de Marte se deben evitar obstáculos, que en este caso eran los píxeles del mapa con altura mayor o igual a 0.25 metros. Se utilizó la librería "simpleai", para ser más específicos, de ahí se obtuvo la instancia SearchProblem y los algoritmos de búsqueda implementados. Aquí las acciones se definen como los ocho movimientos posibles, ya que el Rover tiene la capacidad de moverse en direcciones diagonales. El resultado se define simplemente como el cambio del estado por la acción recibida. Mientras tanto, la meta es la comparación del punto actual del Rover con el punto que se considera meta. La función heurística utilizada es la distancia euclidiana, ya que el objetivo es medir la distancia del punto actual a la meta establecida.

Para esta prueba, se implementaron los algoritmos Breadth First Search, Depth First Search, Uniform Cost Search y A*. Breadth First Search es un algoritmo de búsqueda no informada que cumple es completo, óptimo, e implementa una cola tipo FIFO para expandir la frontera. Con las condiciones iniciales, fue el segundo algoritmo que menos distancia recorrió para llegar al objetivo, recorriendo una distancia total de 3587.42 metros. El segundo algoritmo implementado, Depth First Search, que a diferencia del anterior utiliza una cola tipo LIFO, no llegó a una solución a tiempo. Esto se debe a que este algoritmo no tiene en cuenta el costo al abrirse el paso por los nodos, por lo que puede acabar en un ciclo y nunca terminar. El tercer algoritmo implementado fue el Uniform Cost Search, el cual expande primero los nodos de menor costo. Este quedó bastante parejo con el Breadth First Search con una distancia recorrida de 3612.32, lo cual nos indica que muy probablemente existe un camino más corto a la meta que tiene un costo mayor a otros caminos más largos.

Los algoritmos anteriores son de búsqueda no informada, pero el último algoritmo implementado no comparte esta característica. El algoritmo A* es un algoritmo de búsqueda informada, por lo que tiene información adicional sobre que favorece a la facilidad de búsqueda de la meta. Esta información adicional es la función heurística mencionada con anterioridad, la cual facilita la elección de orden de expansión de la frontera. Con las condiciones iniciales, la distancia recorrida fue de 350.66 metros. Posteriormente, se probaron nuevas condiciones iniciales en las que el punto inicial y la meta estaban más alejados de si. El algoritmo A* resolvió el problema en una distancia recorrida total de 3512.74 metros. Esto indica que una posición inicial más cercana no asegura más facilidad a la hora de encontrar una ruta óptima. Además, esto también es un indicador de que el algoritmo A* es el mejor de entre los algoritmos implementados para resolver este tipo de problemas de búsqueda.

Finalmente, cuando buscamos encontrar rutas desde diferentes puntos, pudimos ver que nuestro algoritmo sirve para las distancias más cortas, de 500 metros a 5000 metros. En la prueba de mayor

a 10000, no encontramos puntos que nos dieran una ruta. Esto se debe a que la superficie de marte es muy variada, y es dificil entontrar áreas suficientemente planas para el robot. Podemos decir que el algoritmo es capaz de resolver el problema cuando el espacio de estados es más limitado. Disminuyendo la cantidad de espacios al que tiene que visitar el algoritmo de búsqueda ayudará al agente resolver el problema.