Planeación de rutas para la exploración en Marte

- Miguuel Emiliano Gozalez Gauna A01633816
- Francisco Javier Chávez Ochoa A01641644
- Laura Merarí Valdivia Frausto A01641790


```
pip install simpleai
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: simpleai in /usr/local/lib/python3.9/dist-packages (0.8.3)
```

```
import numpy as np
import pandas as pd
import plotly.graph_objs as go
import plotly.io as pio
import seaborn as sns
import simpleai.search as search
import math
import time

# Definir la clase MazeState
class Marte(search.SearchProblem):
    def __init__(self,start, goal, board):
        self.board = board
        self.start = start
        self.goal = goal
        super().__init__(initial_state=self.start)

    def actions(self, state):
        actions = []
        row, col = state

        if row > 0 and self.board[row][col] < self.board[row-1][col]+0.25 and self.board[row][col] > self.board[row-1][col]-0.25:
            actions.append((row-1, col))
        if col > 0 and self.board[row][col] < self.board[row][col-1]+0.25 and self.board[row][col] > self.board[row][col-1]-0.25:
            actions.append((row, col-1))
        if row < len(self.board)-1 and self.board[row][col] < self.board[row+1][col]+0.25 and self.board[row][col] > self.board[row+1][col]
            actions.append((row+1, col))
        if col < len(self.board[row])-1 and self.board[row][col] < self.board[row][col+1]+0.25 and self.board[row][col] > self.board[row][c
            actions.append((row, col+1))
        if row > 0 and col > 0 and self.board[row][col] < self.board[row-1][col-1]+0.25 and self.board[row][col] > self.board[row-1][col-1]
            actions.append((row-1, col-1))
        if row > 0 and col < len(self.board[row])-1 and self.board[row][col] < self.board[row-1][col+1]+0.25 and self.board[row][col] > sel
            actions.append((row-1, col+1))
        if row < len(self.board)-1 and col > 0 and self.board[row][col] < self.board[row+1][col-1]+0.25 and self.board[row][col] > self.boa
            actions.append((row+1, col-1))
        if row < len(self.board)-1 and col < len(self.board[row])-1 and self.board[row][col] < self.board[row+1][col+1]+0.25 and self.board
            actions.append((row+1, col+1))
        return actions

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

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

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

    def path(self, node):
        actions = []
        total_distance = 0
        while node.parent_action is not None:
            actions.append(node.parent_action)
            total_distance += math.sqrt((node.state[0]-node.parent.state[0])**2 + (node.state[1]-node.parent.state[1])**2)
            node = node.parent
        actions.reverse()
        return actions, total_distance
    def cost(self, state, action, next_state):
        row, col = state
        next_row, next_col = next_state
        if abs(row - next_row) + abs(col - next_col) > 1:
            return math.sqrt(2)
        else:
            return 1
```

```
# importacion del mapa marciano, cuenta numero columnas y numero filas, escala

mars_map = np.load('mars_map.npy')
nr, nc = mars_map.shape
scale = 10.0174
```

RUTAS

```
# Primera ruta
r = nr-round(6400/scale)
c =round(2850/scale)
start= (r,c)

r = nr-round(6800/scale)
c =round(3150/scale)
```

```
goal= (r,c)

# Rendimiento de los algoritmos de búsqueda para rutas cortas y largas

# coordenadas a 1000 metros
#r = nr-round(6900/scale)
#c =round(3350/scale)
#goal= (r,c)

# coordenadas a 5000 metros
#r = nr-round(7000/scale)
#c =round(5350/scale)
#goal= (r,c)

# coordenadas a 10,000 metros (El algoritmo no llega a su objetivo)
#r = nr-round(3500/scale)
#c =round(11000/scale)
#goal= (r,c)
```

```
# Primera ruta
r = nr-round(6400/scale)
c =round(2850/scale)
start= (r,c)

#r = nr-round(6800/scale)
#c =round(3150/scale)
#goal= (r,c)

# Rendimiento de los algoritmos de búsqueda para rutas cortas y largas

# coordenadas a 1000 metros
r = nr-round(6900/scale)
c =round(3350/scale)
goal= (r,c)

# coordenadas a 5000 metros
#r = nr-round(7000/scale)
#c =round(5350/scale)
#goal= (r,c)

# coordenadas a 10,000 metros (El algoritmo no llega a su objetivo)
#r = nr-round(3500/scale)
#c =round(11000/scale)
#goal= (r,c)
```

```
# Primera ruta
r = nr-round(6400/scale)
c =round(2850/scale)
start= (r,c)

#r = nr-round(6800/scale)
#c =round(3150/scale)
#goal= (r,c)

# Rendimiento de los algoritmos de búsqueda para rutas cortas y largas

# coordenadas a 1000 metros
#r = nr-round(6900/scale)
#c =round(3350/scale)
#goal= (r,c)

# coordenadas a 5000 metros
r = nr-round(7000/scale)
c =round(5350/scale)
goal= (r,c)

# coordenadas a 10,000 metros (El algoritmo no llega a su objetivo)
#r = nr-round(3500/scale)
#c =round(11000/scale)
#goal= (r,c)
```

ALGORITMOS

```
times = []
total_distances = []
step_costs=[]
```

A*

```
# Define the problem and run the search algorithm A*
problem = Marte(start,goal,mars_map)
start_time = time.time()
result = search.astar(problem, graph_search=True)
end_time = time.time()
```

```
actions = result.path()
total_distance = len(actions)*scale
step_cost = result.cost

times.append(end_time - start_time)
total_distances.append(total_distance)
step_costs.append(step_cost)
```

```
print("Total Distance:"+str(round(total_distance))+'m')
print("Time:", end_time - start_time)
print("Cost:"+str(round(step_cost,2)))
```

```
Total Distance:3105m
Time: 0.668189287185669
Cost:337.99
```

```
path1 =np.array([p[0] for p in result.path()])
path1[0]=start

# Convert the result path to a format that can be used with Scatter3d
path_x1 = np.array([p[1] for p in path1])*scale
path_y1 = (nr-np.array([p[0] for p in path1]))*scale
path_z1 = np.array([mars_map[p[0]][p[1]] for p in path1])
```

```
<ipython-input-76-7770a714ab6f>:1: VisibleDeprecationWarning:

Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shap
```

Búsqueda de haz local (Hill Climbing)

```
# Define the problem and run the search algorithm busqueda voraz
problem = Marte(start,goal,mars_map)
start_time = time.time()
result = search.greedy(problem, graph_search=True)
end_time = time.time()
```

```
actions = result.path()
total_distance = len(actions)*scale
step_cost = result.cost

times.append(end_time - start_time)
total_distances.append(total_distance)
step_costs.append(step_cost)

print("Total Distance:"+str(round(total_distance))+'m')
print("Time:", end_time - start_time)
print("Cost:"+str(round(step_cost,2)))
```

```
Total Distance:3656m
Time: 1.2546443939208984
Cost:415.78
```

```
path2 =np.array([p[0] for p in result.path()])
path2[0]=start

# Convert the result path to a format that can be used with Scatter3d
path_x2 = np.array([p[1] for p in path2])*scale
path_y2 = (nr-np.array([p[0] for p in path2]))*scale
path_z2 = np.array([mars_map[p[0]][p[1]] for p in path2])
```

```
<ipython-input-79-553ab644280e>:1: VisibleDeprecationWarning:

Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shap
```

Búsqueda de costo uniforme (UCS)

```
# Define the problem and run the search algorithm busqueda voraz
problem = Marte(start,goal,mars_map)
start_time = time.time()
result = search.greedy(problem, graph_search=True)
end_time = time.time()
```

```
actions = result.path()
total_distance = len(actions)*scale
step_cost = result.cost

times.append(end_time - start_time)
total_distances.append(total_distance)
step_costs.append(step_cost)

print("Total Distance:"+str(round(total_distance))+'m')
print("Time:", end_time - start_time)
print("Cost:"+str(round(step_cost,2)))
```

```
Total Distance:3656m
Time: 0.5837433338165283
Cost:415.78
```

```
path3 =np.array([p[0] for p in result.path()])
path3[0]=start

# Convert the result path to a format that can be used with Scatter3d
path_x3 = np.array([p[1] for p in path3])*scale
path_y3 = (nr-np.array([p[0] for p in path3]))*scale
path_z3 = np.array([mars_map[p[0]][p[1]] for p in path3])
```

```
<ipython-input-82-d4fa9d8cf17b>:1: VisibleDeprecationWarning:
```

```
Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shap
```

Búsqueda en amplitud (BFS)

```
# Define the problem and run the search algorithm busqueda voraz
problem = Marte(start,goal,mars_map)
start_time = time.time()
result = search.breadth_first(problem, graph_search=True)
end_time = time.time()
```

```
actions = result.path()
total_distance = len(actions)*scale
step_cost = result.cost

times.append(end_time - start_time)
total_distances.append(total_distance)
step_costs.append(step_cost)

print("Total Distance:"+str(round(total_distance))+'m')
print("Time:", end_time - start_time)
print("Cost:"+str(round(step_cost,2)))
```

```
Total Distance:3095m
Time: 2.7921621799468994
Cost:342.38
```

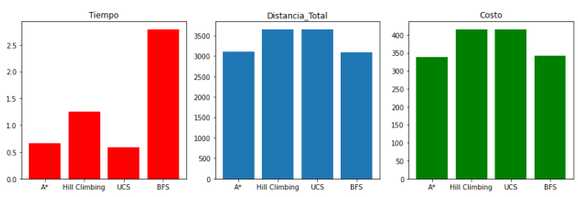
```
path4 =np.array([p[0] for p in result.path()])
path4[0]=start

# Convert the result path to a format that can be used with Scatter3d
path_x4 = np.array([p[1] for p in path4])*scale
path_y4 = (nr-np.array([p[0] for p in path4]))*scale
path_z4 = np.array([mars_map[p[0]][p[1]] for p in path4])
```

```
<ipython-input-85-609d4620eec7>:1: VisibleDeprecationWarning:

Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shap
```

ANALISIS de Tiempo, Distancia total y costo de mis algoritmos

```
resultados = pd.DataFrame({'Tiempo':times, 'Distancia_Total': total_distances, 'Costo': step_costs})
resultados.index=['A*','Hill Climbing','UCS', 'BFS']
```

```
# Plot of each column in DataFrame Resultadopos
fig, axs = plt.subplots(nrows=1, ncols=3, figsize=(12,4))

axs[0].bar(resultados.index, resultados['Tiempo'], color = 'red')
axs[0].set_title('Tiempo')

axs[1].bar(resultados.index, resultados['Distancia_Total'])
axs[1].set_title('Distancia_Total')

axs[2].bar(resultados.index, resultados['Costo'], color = 'green')
axs[2].set_title('Costo')

fig.tight_layout()

plt.show()
```

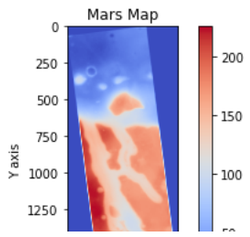


PLOT del Crater

```
x = scale*np.arange(mars_map.shape[1])
y = scale*np.arange(mars_map.shape[0])
X, Y = np.meshgrid(x, y)
```

```
import matplotlib.pyplot as plt

# Create the figure and axis objects
fig, ax = plt.subplots()
```

```
# Create the heatmap using the imshow function
heatmap = ax.imshow(mars_map, cmap='coolwarm')

# Add a colorbar to the heatmap
cbar = ax.figure.colorbar(heatmap, ax=ax)

# Set the axis labels and title
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_title('Mars Map')

# Show the plot
plt.show()
```




```
fig = go.Figure(data=[
            go.Surface(
                x=X, y=Y, z=np.flipud(mars_map), colorscale='hot', cmin=0,
                lighting=dict(ambient=0.0, diffuse=0.8, fresnel=0.02, roughness=0.4, specular=0.2),
                lightposition=dict(x=0, y=nr/2, z=2*mars_map.max())
            ),
            go.Scatter3d(
                x=path_x1, y=path_y1, z=path_z1, name='A*', mode='markers',
                marker=dict(color=np.linspace(0, 1, len(path_x1)), colorscale="Viridis", size=4)
            ),
            go.Scatter3d(
                x=path_x2, y=path_y2, z=path_z2, name='Hill Climbing', mode='markers',
                marker=dict(color=np.linspace(0, 1, len(path_x2)), colorscale="YlOrRd", size=4)
            ),
            go.Scatter3d(
                x=path_x3, y=path_y3, z=path_z3, name='UCS', mode='markers',
                marker=dict(color=np.linspace(0, 1, len(path_x3)), colorscale="Blues", size=4)
            ),
            go.Scatter3d(
                x=path_x4, y=path_y4, z=path_z4, name='BFS', mode='markers',
                marker=dict(color=np.linspace(0, 1, len(path_x4)), colorscale="Reds", size=4)
            )
        ], layout=go.Layout(
            scene_aspectmode='manual',
            scene_aspectratio=dict(x=1, y=nr/nc, z=max(mars_map.max()/x.max(), 0.2)),
            scene_zaxis_range=[0, mars_map.max()]
        ))

#fig.show()
pio.write_html(fig, file='mapa_marte.html', auto_open=True)
```

Respuesta a las preguntas planteadas

¿Qué algoritmos lograron encontrar una ruta válida?

1. Los Cuatro algoritmos utilizados fueron capaces de encontrar una ruta valida para el camino determinado excepto para el mayor a 10,000 m de distancia

¿Es necesario utilizar búsquedas informadas para este caso?

2. No creemos que el robot necesite tener más información sobre aquellos nodos no objetivos prometedores para llegar. El robot conforme vaya explorando su entorno puede ir aprendiendo del mismo. Aun así los es dificil llegar a resultados lejanos para el agente ya que el territioro varia mas conforma mas lejano sea.

¿Qué función heurística resultó adecuada para este problema?

3. La funcion heurística que utilizamos fue la distancia euclidanea desde el punto actual hasta el punto objetivo, creemos que esta fue la mejor heurística por que la distancia de bloque puede tener mucho error al calcular la distancia cuando son rutas muy largas

¿En qué casos el algoritmo es capaz de resolver el problema en un tiempo aceptable?

4. Realmente el tiempo en el que corren los algoritmos no es muy elevado, pero al ser una distancia tan corta consideramos que el algoritmo UCS y A* son los mejores tiempos. En el primer caso y en el caso cuando son 1000 el tiempo es aceptable, pero mientras mas lejano sea el objetivo mayor es el tiempo de compilacion

¿En los casos que el algoritmo no encuentra un resultado, ¿qué acciones se podrían realizar para ayudar al algoritmo a resolver el problema?

5. Una de las posibles causas para que el algoritmo no pueda llegar al resultado es la restricción de la altura. Podemos poner un poco más de flexibilidad de altura entre dos puntos adyacentes, de esta manera sería mucho más difícil que el robot se quede atorado o no pueda llegar al objetivo. Otra opción para que el algoritmo pueda llegar a la solución, sería dividir la ruta en rutas más pequeñas. Por ejemplo, si queremos llegar de A a B pero hay mucha distancia, podemos poner puntos intermedios C y D, y ponemos al algoritmo a encontrar una ruta de A a C, otra de C a D y una última de D a B.

Conclusión

La primera dificultad que tuvimos fue a la hora de intentar implementar las condiciones necesarias que se debían cumplir para poder retornar la lista de posibles acciones. Inicialmente nuestra condición era que si la diferencia entre el punto actual y el posible punto vecino era menor que 0.25, se agregara a la lista de posibles acciones, esto en principio funciona correctamente para algunos casos, pero no para todos. El problema es cuando en vez de ser positiva la diferencia de altura, es negativa, por ejemplo si estamos en un punto de altura 115 y queremos bajar a un punto de 114.5, no deberíamos poder, pues hay una diferencia de más de 0.25, pero como planteamos el problema si se cumple que 115 - 114.5 < .25 y entonces esta acción era agregada a la lista. La solución de este problema fue sencilla, simplemente tenemos que verificar que la magnitud de la altura de la posición actual menos la altura del posible vecino fuera menor que 0.25, esto se puede hacer utilizando el valor absoluto. A la hora de plantear las condiciones también tuvimos el error que estábamos comparando la altura de la posición vecina contra sí misma, y pues esto siempre da 0, que claramente cumple con las condiciones descritas anteriormente, entonces cualquier ruta que elegía el algoritmo era válida en teoría, pero al graficar las rutas, notamos que el robot bajaba por pendientes muy pronunciadas, lo cual no debería de pasar.Este error fue muy fácil de corregir, y una vez implementando los cambios, a la hora de graficar, ya salían rutas razonables.

Otra dificultad que tuvimos fue que se nos olvidó pasar el argumento de graph search a la función astar. Al no pasarle este parámetro, el algoritmo se ciclaba y no podía llegar a la solución deseada. Esto sucede porque la búsqueda de grafo nos permite recordar estados pasados del problema y de esta manera evitar ciclos, que era justamente el problema que teníamos nosotros.

Un error que tuvimos fue a la hora de construir el mapa con el arreglo numpy, se nos olvidó escalar los valores y nuestro mapa tenía otro sistema coordenado que no era el adecuado para resolver este problema. Al principio no notamos este error, pues el mapa se imprimía aparentemente de una manera correcta, pero a la hora de tratar de implementar la ruta, salía en una posición que no debería de salir. La solución de este problema estuvo un poco complicada, pues tuvimos que cambiar nuestro sistema coordenado y a la vez tuvimos que aplicar una escala a las rutas que ya teníamos, para que pudieran salir en el nuevo sistema coordenado.

Por último, tuvimos ciertos problemas para implementar la función costo, pues tenemos 2 diferentes costos dependiendo del movimiento que había hecho el robot, si fue un movimiento en diagonal, el costo es de raiz de 2, y si se mueve hacia adelante o atrás, izquierda o derecha, el costo es de 1.