Descenso al fondo de un cráter en Marte

Francisco Javier Chávez Ochoa A01641644

Miguel Emiliano González Gauna A01633816

Laura Merarí Valdivia Frausto A01641790

```
import numpy as np
mars_map = np.load('mars_map (1) (1).npy')
mars_map.shape
```

```
import numpy as np
import plotly.graph_objs as go
import plotly.io as pio

import math
import time
import random
```

```
# Definir la clase MazeState
class crater(object):


    def __init__(self, r, c,mapa):

        self.r = r
        self.c = c
        self.mapa=mapa


    def cost(self):

        return self.mapa[self.r][self.c]

    def neighbor(self):

      vecino=[]
      if abs(mars_map[self.r][self.c+1]- mars_map[self.r][self.c])<= 2:
        vecino.append([self.r, self.c+1])

      if abs(mars_map[self.r][self.c-1]- mars_map[self.r][self.c]) <=2:
        vecino.append([self.r, self.c-1])

      if abs(mars_map[self.r+1][self.c+1]- mars_map[self.r][self.c]) <=2:
        vecino.append([self.r+1, self.c+1])

      if abs(mars_map[self.r+1][self.c]- mars_map[self.r][self.c]) <=2:
        vecino.append([self.r+1, self.c])

      if abs(mars_map[self.r+1][self.c-1]- mars_map[self.r][self.c]) <=2:
        vecino.append([self.r+1, self.c-1])

      if abs(mars_map[self.r-1][self.c+1]- mars_map[self.r][self.c]) <=2:
        vecino.append([self.r-1, self.c+1])

      if abs(mars_map[self.r-1][self.c]- mars_map[self.r][self.c]) <=2:
        vecino.append([self.r-1, self.c])

      if abs(mars_map[self.r-1][self.c-1]- mars_map[self.r][self.c]) <=2:
        vecino.append([self.r-1, self.c-1])

      vecino_random=random.choice(vecino)
      new_map=crater(vecino_random[0], vecino_random[1], self.mapa)

      return new_map


nr, nc = mars_map.shape
scale = 10.045
```

```
r = nr-round(5800/scale)
c =round(3350/scale)
```

Greedy Search

```
random.seed(time.time()*1000)

mapa_crater= crater(r,c,mars_map)      # Initialize board


cost = mapa_crater.cost()         # Initial cost
step = 0
path1_r=[]
path1_c=[]
while  step<10000 and cost > 0:

    step += 1
    neighbor = mapa_crater.neighbor()
    new_cost = neighbor.cost()

    if new_cost < cost:
      mapa_crater = neighbor
      cost = new_cost


    path1_r.append(mapa_crater.r)
    path1_c.append(mapa_crater.c)


    print("Iteration: ", step, "    Cost: ", cost)



print("--------Solution-----------")
```

```
Iteration:  87     Cost:  109.61716064453147
Iteration:  88     Cost:  109.61716064453147
Iteration:  89     Cost:  109.61716064453147
Iteration:  90     Cost:  109.61716064453147
Iteration:  91     Cost:  109.61716064453147
Iteration:  92     Cost:  109.61716064453147
Iteration:  93     Cost:  109.61716064453147
Iteration:  94     Cost:  109.61716064453147
Iteration:  95     Cost:  109.61716064453147
Iteration:  96     Cost:  109.61716064453147
Iteration:  97     Cost:  109.61716064453147
Iteration:  98     Cost:  109.61716064453147
Iteration:  99     Cost:  109.61716064453147
Iteration:  100     Cost:  109.61716064453147
--------Solution-----------
```

Recocido Simulado

```
random.seed(time.time()*1000)

mapa_crater = crater(r,c,mars_map)      # Initialize board
dic={}
cost = mapa_crater.cost()               # Initial cost
step = 0                                # Step count

alpha = 0.9995                          # Coefficient of the exponential temperature schedule
t0 = 20                                 # Initial temperature
t = t0

path2_r=[]
path2_c=[]
while t > 0.005 and cost > 0:

    # Calculate temperature
    t = t0 * math.pow(alpha, step)
    step += 1

    # Get random neighbor
    neighbor = mapa_crater.neighbor()
    new_cost = neighbor.cost()


    if new_cost < cost:
        mapa_crater = neighbor
        cost = new_cost
    else:
        # Calculate probability of accepting the neighbor
        p = math.exp(-(new_cost - cost)/t)
        if p >= random.random():
            mapa_crater = neighbor
            cost = new_cost


    path2_r.append(mapa_crater.r)
    path2_c.append(mapa_crater.c)
    #if step%100 ==1:
    print("Iteration: ", step, "    Cost: ", cost, "    Temperature: ", t)
```

```
Iteration:  16540     Cost:  0.26782226562521827     Temperature:  0.005113680954061567
Iteration:  16541     Cost:  0.26782226562521827     Temperature:  0.005111124113584537
Iteration:  16542     Cost:  0.26782226562521827     Temperature:  0.005108568551527745
Iteration:  16543     Cost:  0.26782226562521827     Temperature:  0.005106014267251981
Iteration:  16544     Cost:  0.26782226562521827     Temperature:  0.005103461260118356
Iteration:  16545     Cost:  0.26782226562521827     Temperature:  0.005100909529488297
Iteration:  16546     Cost:  0.26782226562521827     Temperature:  0.005098359074723552
Iteration:  16547     Cost:  0.26782226562521827     Temperature:  0.005095809895186906
Iteration:  16548     Cost:  0.26782226562521827     Temperature:  0.005093261990238596
Iteration:  16549     Cost:  0.26782226562521827     Temperature:  0.00509071535924348
Iteration:  16550     Cost:  0.26782226562521827     Temperature:  0.005088170001563858
Iteration:  16551     Cost:  0.26782226562521827     Temperature:  0.005085625916563077
Iteration:  16552     Cost:  0.26782226562521827     Temperature:  0.005083083103604794
Iteration:  16553     Cost:  0.26782226562521827     Temperature:  0.005080541562052994
Iteration:  16554     Cost:  0.26782226562521827     Temperature:  0.005078001291271967
Iteration:  16555     Cost:  0.26782226562521827     Temperature:  0.005075462290626331
Iteration:  16556     Cost:  0.26782226562521827     Temperature:  0.005072924559481017
Iteration:  16557     Cost:  0.26782226562521827     Temperature:  0.005070388097201278
Iteration:  16558     Cost:  0.26782226562521827     Temperature:  0.005067852903152677
Iteration:  16559     Cost:  0.26782226562521827     Temperature:  0.005065318976701102
Iteration:  16560     Cost:  0.26782226562521827     Temperature:  0.005062786317212750 5
Iteration:  16561     Cost:  0.26782226562521827     Temperature:  0.005060254924054145
Iteration:  16562     Cost:  0.26782226562521827     Temperature:  0.005057724796592118
Iteration:  16563     Cost:  0.26782226562521827     Temperature:  0.005055195934193823
Iteration:  16564     Cost:  0.26782226562521827     Temperature:  0.005052668336226726
Iteration:  16565     Cost:  0.26782226562521827     Temperature:  0.005050142002058612 5
Iteration:  16566     Cost:  0.26782226562521827     Temperature:  0.00504761693105758 4
Iteration:  16567     Cost:  0.26782226562521827     Temperature:  0.005045093122592056
Iteration:  16568     Cost:  0.26782226562521827     Temperature:  0.00504257057603076
Iteration:  16569     Cost:  0.26782226562521827     Temperature:  0.005040049290742744
Iteration:  16570     Cost:  0.26782226562521827     Temperature:  0.005037529266097373
Iteration:  16571     Cost:  0.26782226562521827     Temperature:  0.005035010501464325
Iteration:  16572     Cost:  0.26782226562521827     Temperature:  0.005032492996213593
Iteration:  16573     Cost:  0.26782226562521827     Temperature:  0.005029976749715486
Iteration:  16574     Cost:  0.26782226562521827     Temperature:  0.00502746176134063
Iteration:  16575     Cost:  0.26782226562521827     Temperature:  0.00502494803045996
Iteration:  16576     Cost:  0.26782226562521827     Temperature:  0.005022435556444729
Iteration:  16577     Cost:  0.26782226562521827     Temperature:  0.005019924338666507
Iteration:  16578     Cost:  0.26782226562521827     Temperature:  0.005017414376497175
Iteration:  16579     Cost:  0.26782226562521827     Temperature:  0.005014905669308926
Iteration:  16580     Cost:  0.26782226562521827     Temperature:  0.005012398216474272
Iteration:  16581     Cost:  0.26782226562521827     Temperature:  0.005009892017366035
Iteration:  16582     Cost:  0.26782226562521827     Temperature:  0.005007387071357352
Iteration:  16583     Cost:  0.26782226562521827     Temperature:  0.005004883377821673
Iteration:  16584     Cost:  0.26782226562521827     Temperature:  0.005002380936132762
Iteration:  16585     Cost:  0.26782226562521827     Temperature:  0.004999879745664697
```

```
path_x1 = np.array([p for p in path1_c])*scale
path_y1 = (nr-np.array([p for p in path1_r]))*scale
path_z1 = np.array([mars_map[path1_r[j]][path1_c[j]] for j in range(len(path1_r))])

path_x2 = np.array([p for p in path2_c])*scale
path_y2 = (nr-np.array([p for p in path2_r]))*scale
path_z2 = np.array([mars_map[path2_r[j]][path2_c[j]] for j in range(len(path2_c))])


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


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='Greedy', 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='Recocido', mode='markers',
                marker=dict(color=np.linspace(0, 1, len(path_x1)), colorscale="Viridis", 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)
```

Preguntas

¿Qué algoritmo logra llegar más profundo en el cráter?

El recocido simulado es el algoritmo que nos permite llegar hasta el fondo del cráter. Esto porque la superficie es muy irregular y muchas subidas y bajadas, por esta razón el algoritmo de búsqueda voraz no funciona para resolver este algoritmo, pues llega a un mínimo local y no puede salir de ahí; en cambio el recocido simulado, si puede salir de esos mínimos locales y puede buscar el objetivo.

¿Recomendarían a los ingenieros del robot utilizar alguno de estos algoritmos? Recomendamos que los ingenieros implementen el algoritmo de recocido simulado para la búsqueda al fondo del cráter, pues en la simulación, el rover perseverance pudo llegar al fondo del cráter después de 13,000 movimientos. Consideramos que se puede editar un poco el algoritmo de recocido simulado,utilizando un hashmap,con el fin de que el algoritmo no repita posiciones y se pueda mover con más fluidez, de esta manera, el rover podría llegar a su objetivo de una manera más rápida

Conclusión

Para el desarrollo de estos algoritmos para llegar al fondo del cráter,tuvimos una infinidad de dificultades, la mayoría de estos problemas eran con la función que utilizabamos para definir vecinos de nuestro estado actual. El primer acercamiento que tuvimos para realizar la función que te da el mejor vecino fue realizar una lista cuyos elementos eran los números -1,0 y 1, esta lista la recorríamos con un doble ciclo for, de manera que nos dieran todas las posibilidades de escoger 2 elementos. Esto nos regresaba 9 posibles vecinos(8 vecinos + actual) y de ahí seleccionamos el mejor de estos vecinos con la función costo. El error que nos daba esta lógica era en el algoritmo de recocido simulado cuando la posición actual era la mejor, pues al ser la mejor de las opciones, el algoritmo siempre tomaba esa posición y no tenía la posibilidad de cambiarse a un vecino peor, y por lo tanto se quedaba atorado.

La siguiente opción que intentamos para la función para encontrar los vecinos, era revisar todos los posibles vecinos y agregarlos a una lista, posteriormente esta lista la ordenamos de menor a mayor dependiendo de su costo, por último está lista la regresábamos al programa principal. Una vez en el programa principal, íbamos recorriendo la lista, y elegíamos el mejor vecino posible, siempre y cuando este vecino no hubiera sido elegido en el pasado, esto con el fin de implementar la idea mencionada en la pregunta 2 de este documento.

La última opción que implementamos, fue revisar con 8 ifs si el vecino cumplía con las condiciones especificadas por el problema, la cual nos decía que el robot no podía pasar a posiciones cuya diferencia de altura superara los 2 metros, si el posible vecino cumplia con esta condición, se agregaba a una lista de posibles vecinos. Una vez revisados los posibles vecinos, retornamos,de manera aleatoria, a un vecino que cumplía las condiciones. En el programa principal revisábamos este vecino y dependiendo el algoritmo, lo aceptamos o no.

Otro problema que tuvimos, fue en la generación del archivo numpy, por alguna extraña razón, nuestro archivo numpy estaba dañado, lo cual nos impidió, probar nuestro código de manera correcta. De hecho, una vez corregido el archivo numpy, pudimos implementar las tres opciones que describimos anteriormente, en realidad estas funciones si funcionaban de la manera correcta en la mayoría, solo teníamos que arreglar unos casos pequeños que hacían que no funcionara la función, pero con el archivo numpy arreglado, pudimos corregir todo lo que hacía falta.

Por último, también tuvimos algunos problemas a la hora de graficar en 3D la imagen del cráter y la ruta que utilizó el algoritmo de recocido simulado para llegar hasta el fondo del cráter. El error que teníamos era que estaban volteados el eje X, y el eje Y en la función para graficar la ruta, esto provocaba que la ruta apareciera en otro lugar en el mapa. que no correspondía a la ruta que el algoritmo había marcado.