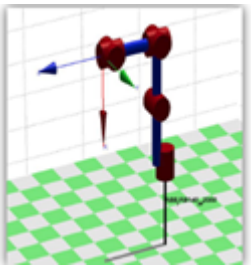
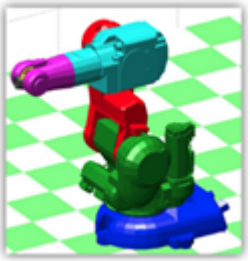
Informe de trabajo Práctico N°1: Búsqueda y optimización

Olguin Nahuel Molina Matías Maccapa Luis

10 de junio de 2021

1. **Dado un punto en el *espacio articular* de un robot serie de 6 grados de libertad, encontrar el camino más corto para llegar hasta otro punto utilizando el algoritmo A*. Genere aleatoriamente los puntos de inicio y fin, y genere también aleatoriamente obstáculos que el robot debe esquivar, siempre en el espacio articular**





Primero generamos el mapa, recordamos que la dimensión de la búsqueda depende del número de articulaciones del brazo robótico, en nuestro caso tenemos 6 articulaciones por lo que la matriz que representa al mapa debe ser 6D.

Con N articulaciones → Búsqueda N-Dimensional

```
def main():
    print("Ejercicio 2")
    NposD=14        #Elegimos el numero de posiciones q se pueden tomar para cada direccion
    map = np.zeros([NposD,NposD,NposD,NposD,NposD,NposD]) #Generamos la matriz de 6D
    NposT=NposD**6       #Numero de posiciones
    print ("Numero de posiciones", NposT)
    inicio=1                #inicio representado con '1'
    fin=3                   #fin representado con '3'
    map[0,0,0,0,0,0]=inicio          #pos inicial
    map[13,13,13,13,13,13]=fin       #pos final
```

Luego tomamos un porcentaje de las posiciones totales que se pueden ocupar en el mapa y las convertimos en obstáculos (los obstáculos se generan aleatoriamente en el mapa).

```
obstaculo=4   #generamos los obstaculos aleatorios
PorcObst=80   #40% del mapa son obstaculos
Nobstaculo=int(NposT*PorcObst/100)        #numero de obstaculos en la matriz
print ("Numero de obstaculos", Nobstaculo)
for i in range (Nobstaculo):
    x=random.randint(0, map.shape[0]-1) #generamos las 6 coordenadas aleatorias
    y=random.randint(0, map.shape[1]-1)
    z=random.randint(0, map.shape[2]-1)
    xx=random.randint(0, map.shape[3]-1)
    yy=random.randint(0, map.shape[4]-1)
    zz=random.randint(0, map.shape[5]-1)
    if map[x,y,z,xx,yy,zz]==0:
        map[x,y,z,xx,yy,zz]=obstaculo  #creamos un obstaculo en la posicion aleatoria
```

Una vez generado el mapa con sus respectivos obstáculos, podemos pasar a explicar la implementación del algoritmo A*

Para el desarrollo del algoritmo A* fue necesario aplicar conceptos de Programación Orientada a Objetos donde se ha creado una clase llamada “Nodo” cuyos atributos son el costo del camino “g”, la heurística “h” , ”f” ,su padre ”padre” y su posición en la matriz “pos”. Sus métodos permiten calcular “g”, “h” y “f”.

$$f(n) = g(n) + h(n)$$

g(n): costo de ruta desde el nodo raíz hasta el nodo n

h(n): costo estimado desde el nodo n al nodo objetivo

```
class Nodo:
    def __init__(self, padre=None, pos=[0,0,0,0,0,0]):
        self.padre = padre
        self.pos = pos
        self.g = 0
        self.h = 0
        self.f = 0
    def calculate_h(self,objetivo): #Heuristica entre nodo actual y final
        self.h = math.sqrt((self.pos[0]-objetivo.pos[0])**2+(self.pos[1]-objetivo.po
    def calculate_g(self,estado_actual,vecino): #camino recorrido
        self.g = estado_actual.g + math.sqrt((self.pos[0]-vecino.pos[0])**2+(self.po
    def calculate_f(self): #Calculo de f
        self.f = self.g + self.h
```

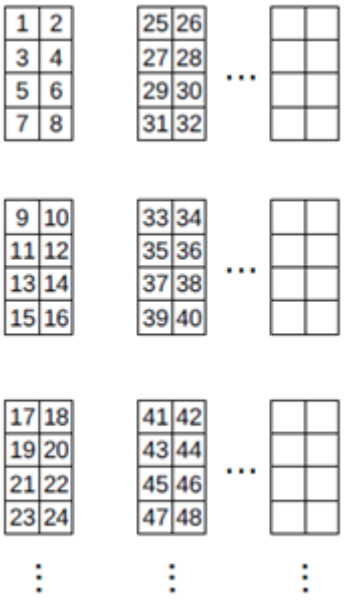
El algoritmo A* hace uso de dos listas: la lista abierta contiene los nodos aún no explorados y la lista cerrada contiene los nodos explorados. A continuación describimos la secuencia de pasos a realizar:

1. Establecer la “posición actual”
2. Añadir los vecinos a la lista abierta y establecer sus atributos (h,g,f, padre y pos)
3. El vecino que tenga menor “f” se toma como “posición actual” y se añade a la lista cerrada

La secuencia anterior se repite hasta que el “estado actual” corresponda al “estado objetivo”, en ese momento se hace una lista de los sucesivos padres de cada nodo partiendo desde el nodo objetivo para obtener el camino a seguir. En la salida se obtiene lo siguiente:

```
Ejercicio 1
Numero de posiciones 7529536
Numero de obstaculos 6023628
Camino optimo:
[(1, 1, 1, 1, 1, 1), (2, 2, 2, 2, 2, 2), (3, 3, 3, 3, 3, 3), (4, 4, 4, 4, 4, 3), (5, 5, 5, 5, 5, 4), (6, 6, 6, 5, 6, 5), (7, 7, 7, 6, 6, 6),
 (8, 8, 8, 7, 7, 7), (9, 9, 9, 8, 8, 8), (10, 10, 9, 9, 9, 9), (11, 11, 10, 10, 10, 10), (12, 12, 11, 11, 11, 11), (13, 13, 12, 12, 12, 12)]
```

2. **Dado un almacén con un layout similar al siguiente, calcular el camino más corto (y la distancia) entre 2 posiciones del almacén, dadas las coordenadas de estas posiciones, utilizando el algoritmo A***



Este problema es análogo al anterior pero en 2D. Primero se generó la matriz del mapa asignándole a cada producto un número diferente y a las casillas libres se les asigna el valor “0”

```
print("Ejercicio 2")
map = np.array([[ 0,  0,  0,  0,  0,  0,  0],
                [ 0,  1,  2,  0, 25, 26,  0],
                [ 0,  3,  4,  0, 27, 28,  0],
                [ 0,  5,  6,  0, 29, 30,  0],
                [ 0,  7,  8,  0, 31, 32,  0],
                [ 0,  0,  0,  0,  0,  0,  0],
                [ 0,  9, 10,  0, 33, 34,  0],
                [ 0, 11, 12,  0, 35, 36,  0],
                [ 0, 13, 14,  0, 37, 38,  0],
                [ 0, 15, 16,  0, 39, 40,  0],
                [ 0,  0,  0,  0,  0,  0,  0],
                [ 0, 17, 18,  0, 41, 42,  0],
                [ 0, 19, 20,  0, 43, 44,  0],
                [ 0, 21, 22,  0, 45, 46,  0],
                [ 0, 23, 24,  0, 47, 48,  0],
                [ 0,  0,  0,  0,  0,  0,  0]])
a=1 #inicio
b=48 #fin
solution = a_star(map,a,b)  #llamamos a la funcion A* indicando el mapa, el inicio y el fin
```

La implementación del algoritmo A* es igual al ejercicio anterior, se crea una clase nodo con los mismos métodos y atributos; se usan las listas abierta y cerrada, y finalmente se obtiene el camino más corto entre productos siguiendo la secuencia de pasos descrita anteriormente

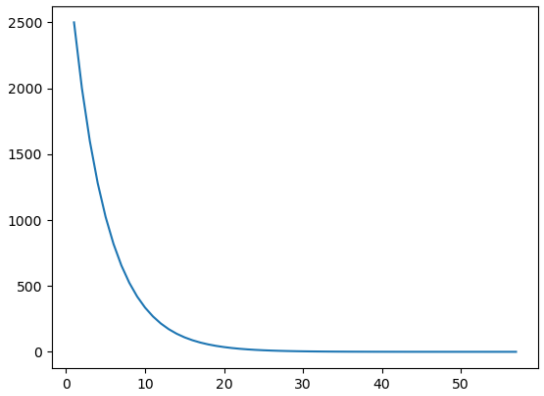
A continuación se muestra el camino a seguir en el mapa representado por los valores “-1”

```
Ejercicio 2
Camino optimo:
[(1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0), (8, 0), (9, 0), (10, 0), (10, 1), (10, 2), (10, 3), (11, 3), (12, 3), (13, 3), (14,
 3), (15, 3), (15, 4), (15, 5)]
[[ 0  0  0  0  0  0  0]
 [-1  1  2  0 25 26  0]
 [-1  3  4  0 27 28  0]
 [-1  5  6  0 29 30  0]
 [-1  7  8  0 31 32  0]
 [-1  0  0  0  0  0  0]
 [-1  9 10  0 33 34  0]
 [-1 11 12  0 35 36  0]
 [-1 13 14  0 37 38  0]
 [-1 15 16  0 39 40  0]
 [-1 -1 -1 -1  0  0  0]
 [ 0 17 18 -1 41 42  0]
 [ 0 19 20 -1 43 44  0]
 [ 0 21 22 -1 45 46  0]
 [ 0 23 24 -1 47 48  0]
 [ 0  0  0 -1 -1 -1  0]]
```

3. Dada una orden de pedido, que incluye una lista de productos del almacén anterior que deben ser despachados en su totalidad, determinar el orden óptimo para la operación de picking mediante Temple Simulado. ¿Qué otros algoritmos pueden utilizarse para esta tarea?

El ejercicio al principio se resolvió llamando al programa de A* y se realizaba el temple simulado con una función de enfriamiento lineal , lo que lo hacía muy lento ; tardaba alrededor de 30 minutos calcular la distancia mínima para una orden de 5 productos.

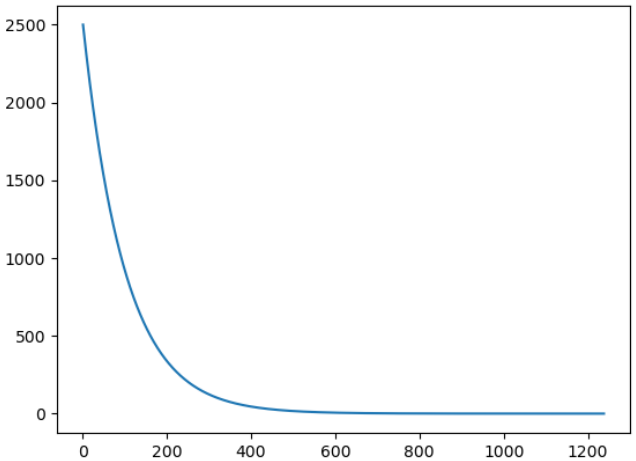
Para disminuir el tiempo de ejecución se procedió a realizar una caché , la cual pre calculaba todas las distancias entre dos productos y se las introdujo en una matriz llamada Matriz Costo y las distancias de los productos a la bahía de carga se cargaron en un vector llamado Costo BaseCarga. Esto disminuye los tiempos de ejecución pero , para disminuirlos mas se utilizo una función de enfriamiento logarítmica. La cual nos entrega el resultado casi instantáneamente.

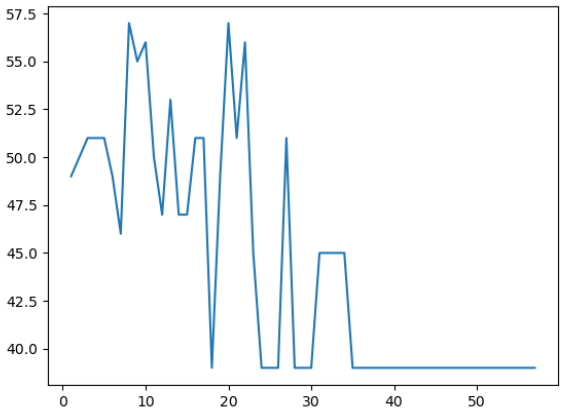


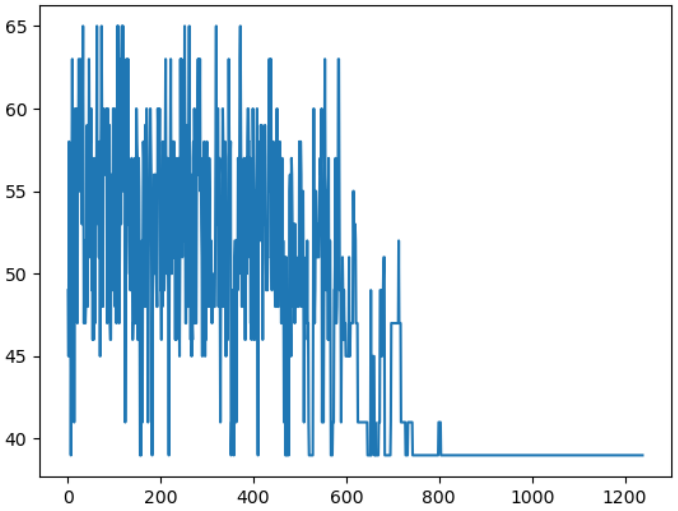
Como se puede observar con una temperatura de 2500 y una secuencia de enfriamiento de 0.8 el algoritmo converge en menos de 40 iteraciones.

```
distancia inicial: 49.0 con la lista: [37 47 11 48 24]
distancia minima: 39.0 con la lista: [11 37 47 48 24]
```

Y se puede observar que efectivamente disminuyó la distancia. Si se aumenta la secuencia de enfriamiento hasta un máximo de 0.99 obtiene un resultado similar en una mayor cantidad de iteraciones, en este caso alrededor de 800 iteraciones.





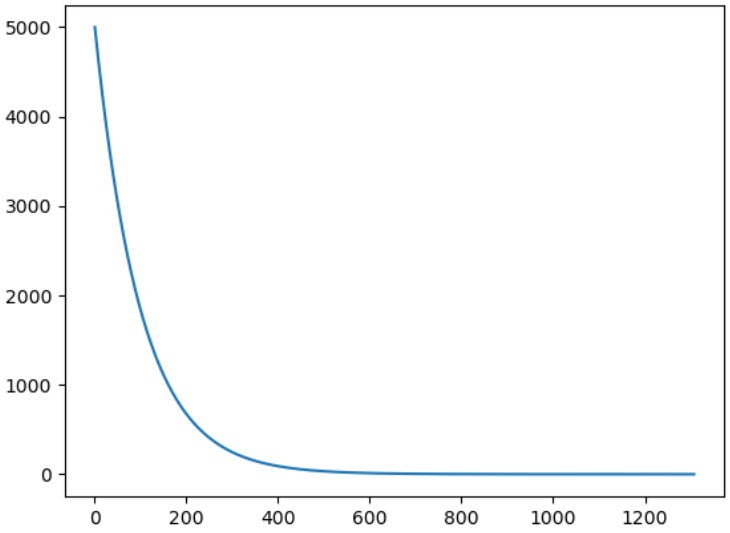


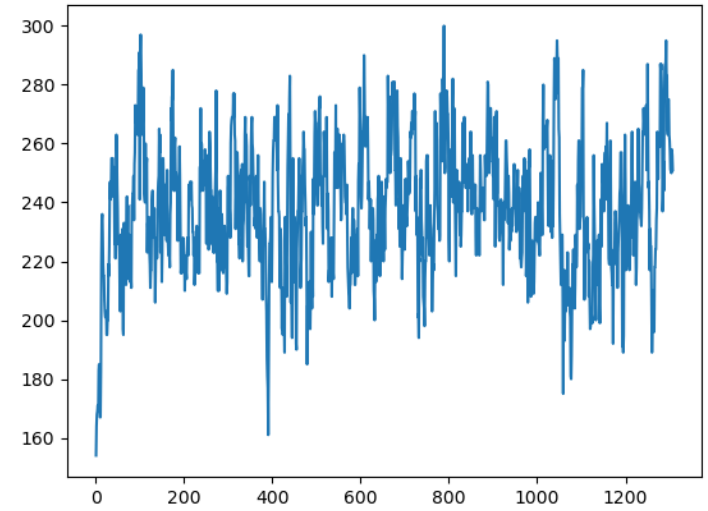
4. **Implementar un algoritmo genético para resolver el problema de optimizar la ubicación de los productos en el almacén, de manera de optimizar el picking de los mismos. Considere que**

- **El layout del almacén está fijo (tamaño y ubicación de pasillos y estanterías), solo debe determinarse la ubicación de los productos**
- **Cada orden incluye un conjunto de productos que deben ser despachados en su totalidad**
- **El picking comienza y termina en una bahía de carga, la cual tiene ciertas coordenadas en el almacén (por generalidad, puede considerarse la bahía de carga en cualquier borde del almacén)**
- **El "costo" del picking es proporcional a la distancia recorrida**

En la resolución de este ejercicio se procedió de manera similar a la mencionada en el ejercicio 3 , pero en este caso se le suma una matriz de 100 pedidos con listas de productos de longitud variable , para solucionar este problema se formateo la matriz y se le agregó 0 en donde la lista no tiene productos para obtener así una matriz de 100 órdenes con 28 productos como máximo.

La función fitness se realizó con temple simulado y la caché anteriormente citada. Pero en esta ocasión el algoritmo de temple simulado no converge y entrega peores resultados





A los individuos se le dio el formato de lista del 1 al 100 según su respectiva posición en el almacén. Esto permite que cuando el individuo pase a la siguiente generación se pueda ir modificando los valores de las matrices de la caché y así obtener los resultados derivados de las permutaciones de los productos.

```
map = np.array([[ 0,  0,  0,  0,  0,  0, 0, 0,   0],
                [ 0,  1,  2,  0, 25, 26,  0, 49, 50, 0],
                [ 0,  3,  4,  0, 27, 28,  0, 51, 52, 0],
                [ 0,  5,  6,  0, 29, 30,  0, 53, 54, 0],
                [ 0,  7,  8,  0, 31, 32,  0, 55, 56, 0],
                [ 0,  0,  0,  0,  0,  0,  0, 0 , 0 , 0],
                [ 0,  9, 10,  0, 33, 34,  0, 57, 58, 0],
                [ 0, 11, 12,  0, 35, 36,  0, 59, 60, 0],
                [ 0, 13, 14,  0, 37, 38,  0, 61, 62, 0],
                [ 0, 15, 16,  0, 39, 40,  0, 63, 64, 0],
                [ 0,  0,  0,  0,  0,  0,  0, 0 , 0 , 0],
                [ 0, 17, 18,  0, 41, 42,  0, 65, 66, 0],
                [ 0, 19, 20,  0, 43, 44,  0, 67, 68, 0],
                [ 0, 21, 22,  0, 45, 46,  0, 69, 70, 0],
                [ 0, 23, 24,  0, 47, 48,  0, 71, 72, 0],
                [ 0,  0,  0,  0,  0,  0,  0,  0, 0 , 0],
                [ 0, 73, 74,  0, 81, 82,  0, 89, 90, 0],
                [ 0, 75, 76,  0, 83, 84,  0, 91, 92, 0],
                [ 0, 77, 78,  0, 85, 86,  0, 93, 94, 0],
                [ 0, 79, 80,  0, 87, 88,  0, 95, 96, 0],
                [ 0,  0,  0,  0,  0,  0,  0,  0,  0, 0],
                [ 0, 97, 98,  0,  0,  0,  0,  0,  0, 0],
                [ 0, 99, 100, 0,  0,  0,  0,  0,  0, 0],
                [ 0,  0,  0,  0,  0,  0,  0,  0,  0, 0]])
```

Orden de los productos en el almacén

```
inicio de programa
Resultados por generacion
[210.75 204.56 211.47 209.37
fin de programa
```

En la imagen se observan los resultados obtenidos en cada nueva generación.

5. Implemente un algoritmo de satisfacción de restricciones para resolver un problema de scheduling. El problema de scheduling consiste en asignar recursos a tareas. Modele las variables, dominio de las mismas, restricciones, etc. Asuma que

- Existe una determinada cantidad de tareas que deben realizarse
- Cada tarea requiere una máquina determinada (no todas las máquinas son del mismo tipo)
- Cada tarea tiene una duración específica
- Se dispone de una determinada cantidad (limitada) de máquinas de cada tipo (puede haber más de una máquina de cada tipo)

5. Algoritmo de satisfacción de Restricciones para un problema de scheduling

El problema consiste en determinadas tareas que deben realizarse dándose un deadline determinado, se hace el uso de diccionarios para especificar las tareas:

```
variables = {"T1": 5, "T2": 15, "T3": 10, "T4": 30, "T5": 25}
```

se utiliza un deadline de 100 horas y cada tarea se especifica en horas siendo el periodo 1 hora.

Se utilizan dos tipos de restricciones, de precedencia donde determinada tarea debe realizarse específicamente antes de otra y de recursos donde cada tarea solo puede ser realizada por una máquina específica.

Se utilizó el algoritmo de backtracking mediante una función recursiva.

Restricciones binarias de precedencia:

```
restricciones_precedencia = {0: ("T2", "T3"), 1: ("T4", "T1"), 2:
("T4", "T3")}
```

Restricciones binarias de recursos:

```
restricciones_recursos = {"T1": (M1), "T2": (M1, M2, M3), "T3": (M1,
M4), "T4": (M1, M3)}
```

Solucion:

```
Orden optimo: ['T4', 'T2', 'T1', 'T3', 'T5']
```