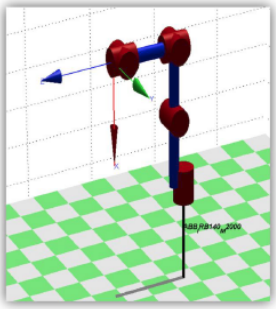
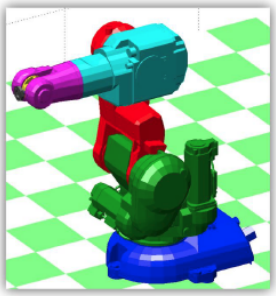
Trabajo Práctico Nº 1
Búsqueda y Optimización

EJERCICIO 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.





Para solucionar el ejercicio de un robot en espacio reticular se discretiza el espacio en 2°, quedando el tamaño total en 180°. En primera instancia se realizó el problema para 3GDL y luego se extendió para 6GDL para no sobrecargar la memoria.

```
import random
from node import node

grados = 180  # grados se refiere a la discretizacion, en este caso 180°, es decir de 2 en 2

xi=random.randint(0,grados)
yi=random.randint(0,grados)
zi=random.randint(0,grados)
ui=random.randint(0,grados)
vi=random.randint(0,grados)
wi=random.randint(0,grados)

xf=random.randint(0,grados)
yf=random.randint(0,grados)
zf=random.randint(0,grados)
uf=random.randint(0,grados)
vf=random.randint(0,grados)
wf=random.randint(0,grados)


ini = [xi,yi,zi,ui,vi,wi] # celda donde empezamos
fin = [xf,yf,zf,uf,vf,wf] #celda destino
```

Generamos los puntos de inicio y fin en forma aleatoria tal y como lo aclara el problema a través de la librería random

```
print("EL robot comienza en el espacio reticular (inicio):")
print(ini)
print("EL robot debe llegar al espacio reticular (meta):")
print(fin)
print("")
density=round(grados*grados*grados*50) #densidad de obstaculos, muy grande para evaluar complejidad
obstcl = [i for i in range(density)]
for i in range(density):
    a=random.randint(0,grados)
    b=random.randint(0,grados)
    c=random.randint(0,grados)
    d=random.randint(0,grados)
    e=random.randint(0,grados)
    f=random.randint(0,grados)
    for obs1 in obstcl:
        if([a,b,c,d,e,f]!=ini and [a,b,c,d,e,f]!=fin and [a,b,c,d,e,f]!=obs1): #un obstaculo no puede
            obstcl[i]= [a,b,c,d,e,f]                    #ser ni el inicio ni el final
            break
        else:                                   #ni tampoco otro obstaculo
            i=i-1
```

Una vez teniendo los puntos de inicio y partida se deben generar los obstáculos también de forma aleatoria. Se debe tener en cuenta que un obstáculo no puede ser el punto de inicio, de fin, ni otro obstáculo ya existente.La creación de obstáculos crece exponencialmente en la memoria por lo que este proceso demora.

```
nodeAct=node(ini,0,0)
#while no este en la casilla final
while (nodeAct.get_pos()!=fin):
    act=nodeAct.get_pos()
    actx=act[0]
    acty=act[1]
    actz=act[2]
    actu=act[3]
    actv=act[4]
    actw=act[5]
    gn=nodeAct.get_gn()
    # proximos nodos a evaluar
    nodes=[]
```

Una vez se tiene todo listo, empezará la búsqueda. Nuestra condición de salida será llegar a destino, para ello debemos saber nuestra posición actual (act) y se creará la lista nodes=[] para evaluar el mejor nodo a moverse, este será el que mejor valor fn()=hn()+gn() tenga.

```
for i in range(actx-1, actx+2):
    for j in range(acty-1, acty+2):
        for k in range(actz-1, actz+2):
            for l in range(actu-1, actu+2):
                for m in range(actv-1, actv+2):
                    for n in range(actw-1, actw+2):
                        if not [i,j,k,l,m,m] in obstcl: #si no es un obstaculo va a explorar
                            gn=nodeAct.get_gn()+1
                            hn=(((xf-i)**2)+((yf-j)**2)+((zf-k)**2)+((uf-l)**2)+((vf-m)**2)+((wf-n)**2))**(0.5)
                            fn=gn+hn # valor representativo
                            nod1=node([i,j,k,l,m,n],gn,hn)
                            nodes.append(nod1) # Lista de proximos nodos posibles
```

Se evalúan todos los nodos adyacentes al que estamos parados actualmente, calculamos su valor gn, en este caso el costo de trasladarse de una celda a otra será de 1, y calculamos su valor hn. La heurística utilizada en este caso es en línea recta (distancia euclidiana). Una vez calculado, se instancia un nodo con sus valores y lo agregamos a la lista siempre y cuando no sea un obstáculo, en caso de que lo sea ni siquiera será evaluado para moverse a esa celda. Finalmente se añade el nodo a la lista nodes.

```
75
76      #ordenar nodo para ver fn mas chico
77      min=grados*grados*grados*grados*grados #valor grande para comparar distancia
78      for nod1 in nodes:
79      # print(nod1.get_hn())
80          if (nod1.get_hn()+nodeAct.get_gn()<min):
81              min=nod1.get_hn()+nodeAct.get_gn()
82              nodeAct=nod1
83
84      print(nodeAct.get_pos())
85
86
87
```

Por último, se ordenan los nodos por su parámetro fn()=gn()+hn(), siendo el 1° el mejor, hay un traslado al nodo con mejor fn() que haya.

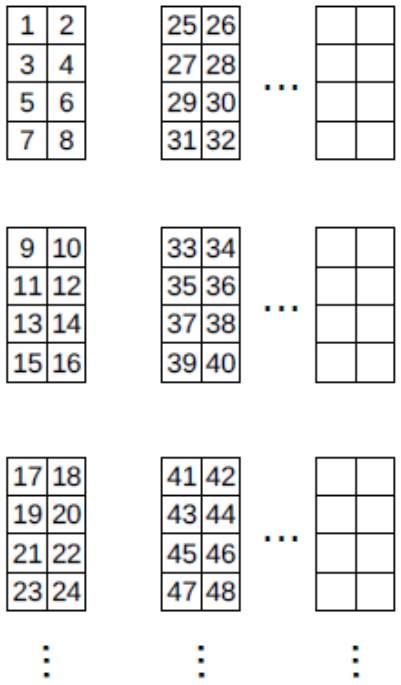
Se aclara que se creó la clase node con sus respectivos setters and getters para el uso del programa

```
1   class node:
2
3       def __init__(self,pos,gn,hn):
4           self.__pos = pos
5           self.__gn = gn
6           self.__hn = hn
7
8       def get_gn(self):
9           return self.__gn
10
11      def set_gn(self, gn):
12          self.__gn = gn
13
14
15      def get_hn(self):
16          return self.__hn
17
18      def set_hn(self, hn):
19          self.__hn = hn
20
21
22      def get_pos(self):
23          return self.__pos
24
```

En resumen, el algoritmo puede optimizarse de varias formas y tiene defectos. Uno de ellos es que no contiene un historial de tal forma que pueda realizar backTracking. En este ejercicio, específicamente, no representa un problema ya que al generarse los obstáculos aleatoriamente es muy difícil que estos obstáculos generen un camino cerrado. Otra optimización o cambio sería implementar la heurística de distancia de manhattan ya que la distancia en línea recta puede ser una mala heurística al momento de atravesar obstáculos más compactos como en el problema del almacén.

EJERCICIO 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*



(El informe siguiente corresponde a la versión 2020)

Este ejercicio presentaba la interesante propuesta de un almacén de tamaño variable y con una numeración de productos difícil de comprender en un principio. La solución utilizada está compuesta por un programa principal, además de diversas clases y funciones; es por ello que se describirá el algoritmo en un orden más bien acorde al usuario que a su orden de escritura.

```python
#CREACION DEL ALMACEN   #1
cant_est_x= int(input("Cuantos estantes de 2x8 tiene el deposito a lo ancho (Oeste a Este) del deposito?\n"))
ancho= 3*cant_est_x+1 #suponiendo que los corredores tienen el ancho de 1 cuadricula

pasillo_horizontal=[]
for x in range(1,ancho+1):
    pasillo_horizontal.append(int(0)) #creamos corredores horizontales vacios para colocar entre estantes

cant_est_y=int(input("Cuantos estantes de 2x8 tiene el deposito en profundidad (Norte a Sur) del deposito?\n"))
prof= 5*cant_est_y-1 #suponiendo que en que los estantes estan pegados a la paredes Norte y Sur


almacen=[] #lista vacia para la matriz del almacen: cada lista agregada como elemento en el siguiente paso será una de las filas de la matriz
for y in range(1,prof+1):
    if (y%5)==0:
        almacen.append(pasillo_horizontal)
    else:
        ancho_en_areas=[]
        for x in range(1,ancho+1):
            if (x%3)==1:
                ancho_en_areas.append(int(0))
            else:
                ancho_en_areas.append(int(1))
        almacen.append(ancho_en_areas)
```

Lo primero que se hace es solicitar al usuario 2 valores, que determinarán en conjunto la cantidad total de estantes que habrá en el almacén. Éste se construye como un arreglo de arreglos, que se unen para conformar una matriz de dos dimensiones que es análoga al almacén, y posee un 0 en los lugares circulables y un 1 en los lugares donde hay estanterías. Cabe destacar que, por estar utilizando Python, nos referimos en realidad a Listas, no arreglos.

```python
print("Almacen:\n",almacen)
"""Con las entradas "3" y "2", se obtiene la lista de listas:
[[0, 1, 1, 0, 1, 1, 0, 1, 1, 0],
[0, 1, 1, 0, 1, 1, 0, 1, 1, 0],
[0, 1, 1, 0, 1, 1, 0, 1, 1, 0],
[0, 1, 1, 0, 1, 1, 0, 1, 1, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 1, 1, 0, 1, 1, 0, 1, 1, 0],
[0, 1, 1, 0, 1, 1, 0, 1, 1, 0],
[0, 1, 1, 0, 1, 1, 0, 1, 1, 0],
[0, 1, 1, 0, 1, 1, 0, 1, 1, 0]]"""
```

Se muestra en el código un ejemplo ilustrativo de cómo quedaría el almacén en un caso ejemplo. Nótese que hacia el Norte y hacia el Sur, los estantes llegan hasta la pared y no permiten el paso, por lo cual para desplazarse a lo ancho del almacén hay que utilizar los corredores centrales, cuyo ancho, por simplicidad, es similar a la de cualquiera de las celdas “espacio” que conforman los estantes.

```
#ESPACIOS DEL ALMACEN   #2
contad_vertical=1
fila_vertical_de_estantes=0

for x in range(0,ancho):
    if ((x+1)%3) == 0:
        fila_vertical_de_estantes+= 1
        for y in range(0,prof):
            if (almacen[y][x]) >= 1:
                almacen[y][x-1]= contad_vertical
                almacen[y][x]= contad_vertical+1
                contad_vertical+= 2

        contad_vertical= (8*int(cant_est_y))*(fila_vertical_de_estantes) + 1

print ("\n""\nAlmacen con espacios numerados:",almacen)
```

En seguida, se procesa un segmento que numera todos los espacios de los estantes de acuerdo a la función mostrada en el ejemplo. Es por ello que se seguirá esa regla de numeración, sin importar el tamaño del almacén. Así, todos los valores, distintos a 0, de la matriz pasan a ser su número de espacio de producto.

```
#CONDICIONES Y OBJETIVOS    #3
x_inic= int(input("\n\nIngrese las coord. de la pos. inicial, teniendo en cuenta que el area frente al Espacio 1 esta ubicada en x:0,y:0\n\nIngrese
y_inic= int(input("Ingrese la coord y: "))
print("Comienzo:", x_inic,",",y_inic)

if  x_inic>ancho-1 or x_inic<0 or y_inic>prof or y_inic<0:
    print("\nLa posicion inicial no puede estar fuera del almacen.\n")
else:
    if almacen[y_inic][x_inic] > 0:
        print("\nLa posicion inicial no puede estar dentro de un estante.\n")
    else:

        producto= int(input("\nIngrese el numero del espacio del producto que desea buscar: "))
        for y in range(0,prof):
            for x in range(0,ancho):
                if almacen[y][x]==producto:
                    if (almacen[y][x]%2) == 0:
                        x_objetivo= x+1
                        y_objetivo= y
                    else:
                        x_objetivo= x-1
                        y_objetivo= y
        print("\nEn la matriz del Almacen, el espacio ubicado frente al producto buscado esta en la columna", x_objetivo,"y la fila", y_objetivo)
        print("Ese espacio es el destino final y, para llegar a el,");
        print("Sí se permite el movimiento oblicuo.");
```

Se solicita al usuario que ingrese las posiciones iniciales x e y. Cabe aclarar que la matriz del almacén se numera desde arriba hacia abajo, partiendo de y=0, y de izquierda a derecha, comenzando con x=0. La posición inicial (0,0) se corresponde entonces a la esquina superior izquierda. Antes de seguir, se realiza un control de la supuesta posición inicial. Esto se debe a que no se puede comenzar ni sobre un estante ni fuera de los espacios que comprenden el almacén.

Luego, se realiza un cambio de variable, debido a que, si bien lo que se busca es un espacio numerado del almacén, no se puede llegar hasta su interior. Entonces, se cambian las coordenadas del producto buscado a las del espacio que se encuentra frente a sí mismo. Éstas están al Oeste en los pasillos impares y al Este en los pares.

Se imprime por pantalla este espacio objetivo, y además se aclara que el algoritmo permite el movimiento oblicuo.

```
#RESOLUCION

        busqueda1= Busqueda_Aestrella()

        busqueda1.definir_almacen(almacen, ancho, prof, x_inic, y_inic, x_objetivo, y_objetivo)

        busqueda1.Resolver()
```

Lo primero que se hace para empezar a resolver el problema es crear un objeto “busqueda1”, de la Clase “Busqueda_Aestrella”. Ésta es muy importante por contiene todos los métodos que se realizarán a continuación.

```
class Busqueda_Aestrella(object):
    def __init__(self):
        self.lista_abiertos = []
        self.lista_cerrados = []     #lista de nodos visitados
        self.nodos= []
        self.almacen_x = None
        self.almacen_y = None


    # "funcion main" - Es el paso inicial basico de mi programa
    def definir_almacen(self, almacen, ancho, prof, x_inic, y_inic, x_objetivo, y_objetivo):
        self.almacen_y = prof
        self.almacen_x = ancho
        for x in range(self.almacen_x):
            for y in range(self.almacen_y):
                if almacen[y][x] == 0:
                    alcanzable = True
                else:
                    alcanzable = False
                self.nodos.append(Nodo(x, y, alcanzable, False))
        self.comienzo = self.obtener_nodo(x_inic, y_inic)

        self.final = self.obtener_nodo(x_objetivo, y_objetivo)
```

La clase se instancia y Busqueda1 posee unas listas para nodos y una posición (x,y), utilizada para el tamaño de la búsqueda.

A continuación, lo que se hace en la Resolución es llamar al método “definir_almacen” de la instancia. Éste llena la lista “nodos” con los distintos Nodo-Espacio que conformarán todos los estados posibles del problema. A éstos les asigna la condición de ser sólidos o atravesables.

Finalmente, y de acuerdo a la siguiente función,

```
def obtener_nodo(self, x, y):
    return self.nodos[x * self.almacen_y + y]
# obtengo un nodo de la lista TOTAL de elementos del almacen


def obtener_Hn(self, nodo):
        return abs(nodo.x - self.final.x) + abs(nodo.y - self.final.y)
```

Por otro lado, cada nodo-espacio es un objeto de la Clase Nodo, y tendrá valores que lo identifican:

```python
class Nodo(object):
    def __init__(self, x, y, alcanzable, diagonal):
        self.alcanzable = alcanzable
        self.diagonal = diagonal
        #
        self.x = x
        self.y = y
        #
        self.padre = None
        #
        self.Gn = 0  #Gn es el costo de moverse al siguiente nodo
        self.Hn = 0  #Hn es la Heuristica... En este caso, la Distancia de Manhattan es apropiada
        self.Fn = 0
```

El siguiente método de la instancia “busqueda1” que se llama es “Resolver”. Éste maneja las listas abierta y cerrada de nodos, siendo la primera aquella que contiene los espacios que son vecinos al “actual” y deben explorarse, y la segunda aquella que almacena los espacios ya recorridos.

Primero ordena la lista abierta de nodos de tal manera que aquel que tiene el atributo Fn más bajo, sea retirado primero. Fn es una función distintiva de A*, y está conformada por la suma entre el “costo del camino hasta ese punto”, Gn y la Heurísitca, Hn. Ésta última varía de problema a problema, y es una función que permite determinar la calidad de una solución. En esta situación particular, la Hn elegida es la Distancia de Manhattan, la cual funciona muy bien a causa de las características del almacén, incluso a pesar de que los movimientos pueden ser oblicuos.

Luego, se toma la lista abierta y se extrae el nodo cuyo Fn es el más bajo, se analiza si cumple con ciertas condiciones y luego se obtienen todos sus nodos adyacentes con la función mostrada “obtener_adyacentes”. Se toman luego todos esos nodos generados y, si no han sido recorridos ya y si no corresponden a una sólida estantería, se los designa como el “nodo_actual”.

```python
def Resolver(self):
    self.lista_abiertos.append(self.comienzo)

    while len(self.lista_abiertos):

        self.ordenar_segun_Fn(self.lista_abiertos)

        nodo= self.lista_abiertos.pop(0)

        if nodo not in (self.lista_cerrados):
            self.lista_cerrados.append(nodo)
        # Agregamos el nodo a la Lista Cerrada para no recorrerlo de nuevo

        if nodo is self.final:
            self.presentar_camino()
            break

        nodos_adyac= self.obtener_adyacentes(nodo)
        for adyac in nodos_adyac:
            if adyac.alcanzable and adyac not in self.lista_cerrados:
                if (adyac.Fn, adyac) in self.lista_abiertos:
                    if adyac.Gn > nodo.Gn + 1:
                        self.nodo_actual(adyac, nodo)
                else:
                    self.nodo_actual(adyac, nodo)
                    self.lista_abiertos.append(adyac)
```

Dicha función se muestra a continuación, y aquella que cumple la importante función de hacer que cada nodo apunte a su predecesor, asignándole la condición de "padre", y permitiendo así unir todo el camino que se irá formando a medida que se van abriendo, examinando y descartando nodos-espacio.

```
def nodo_actual(self, adyac, nodo):
        adyac.Gn= nodo.Gn + 1
        adyac.Hn= self.obtener_Hn(adyac)
        adyac.padre= nodo
        adyac.Fn= adyac.Hn + adyac.Gn
```

Para el ordenamiento de la lista abierta, se utilizó un algoritmo tipo Bubblesort, mostrado a continuación:

```
def ordenar_segun_Fn(self,lista):     #   Usando BubbleSort
    l= len(lista)

    # Recorremos todos los elementos
    for i in range(l):
        #print("i es: ",i)

        # Pero los ultimos "i" elementos ya estan en el el lugar correcto, asi que...
        for j in range(0,l-i-1):
            if lista[j].Fn > lista[j+1].Fn:
                lista[j] , lista[j+1] = lista[j+1] , lista[j]
```

Por otro lado, para generar los nodos-espacio adyacentes se utilizó la función mostrada a continuación

```
def obtener_adyacentes(self, nodo):                                          #12
    nodos = []
    #
    for i in range (nodo.x -1 , nodo.x +1 +1):
        for j in range (nodo.y -1 , nodo.y +1 +1):
            if (i != nodo.x) or (j != nodo.y):
                if not ( (i < 0) or (j < 0) or (i > self.almacen_x-1) or (j > self.almacen_y-1) ):

                    nodos.append(self.obtener_nodo(i , j))
    return nodos
```

Originalmente, se había escrito de la manera comentada abajo, pero luego se decidió que era poco correcta desde el punto de vista de la programación, y por ello se re ideó como se muestra arriba.

```
    if nodo.x < self.almacen_x-1:
        nodos.append(self.obtener_nodo(nodo.x +1 , nodo.y))
    if nodo.y > 0:
        nodos.append(self.obtener_nodo(nodo.x , nodo.y -1))
    if nodo.x > 0:
        nodos.append(self.obtener_nodo(nodo.x -1 , nodo.y))
    if nodo.y < self.almacen_y-1:
        nodos.append(self.obtener_nodo(nodo.x , nodo.y +1))
    #
    if (nodo.x > 0) and (nodo.y > 0):
        nodos.append(self.obtener_nodo(nodo.x -1 , nodo.y -1))
    if (nodo.x > 0) and (nodo.y < self.almacen_y-1 ):
        nodos.append(self.obtener_nodo(nodo.x -1 , nodo.y +1))
    if (nodo.x < self.almacen_x-1) and (nodo.y > 0 ):
        nodos.append(self.obtener_nodo(nodo.x+1 , nodo.y-1))
    if (nodo.x < self.almacen_x-1 ) and (nodo.y < self.almacen_y-1):
        nodos.append(self.obtener_nodo(nodo.x +1 , nodo.y +1))
    #
    return nodos
```

Finalmente, y como condición de salida, existe la siguiente función, a la cual se ingresa cuando el nodo sobre el que se está trabajando resulta ser el espacio objetivo del problema. Se genera entonces una lista “camino”, que contiene los atributos de posición de los nodos recorridos. Ésta se imprime para que el usuario pueda ver el camino recorrido.

```
def presentar_camino(self):                                                    #11
        nodo= self.final
        camino= [(nodo.x, nodo.y)]
        print("\n\nCamino(posiciones x,y):")
        if nodo != self.comienzo:   #Se agrega esta excepcion para el prevenir errores en aquellos casos
                                    #en los cuales se inicia frente al lugar del producto.
            while nodo.padre is not self.comienzo:
                nodo= nodo.padre
                camino.append((nodo.x, nodo.y))
            camino.append((x_inic , y_inic))

        camino.reverse()
        print (camino)
```

Con dicha función se da el programa por terminado, ya que se ha encontrado y mostrado el camino más corto entre el punto de inicio y el espacio frente al producto objetivo.

Este programa has sido completado de forma satisfactoria y ha permitido mejorar enormemente la utilización de Clases y Grafos en Python. Planteó además pequeños desafíos de lógica.

EJERCICIO 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 informe siguiente corresponde a la versión 2020)

La propuesta de este ejercicio es un código que combine las Búsquedas A* (global) y Temple Simulado (local). Este programa híbrido se fundamenta en las fortalezas de cada uno de los métodos. Se usa entonces en un nivel más superficial un Temple Simulado, con el fin de elegir el orden de retiro de productos que minimice el recorrido. Esto es importante, porque la cantidad de combinaciones posibles es, siendo k la cantidad de productos, k!. Este valor crece tan rápido que la utilización de un algoritmo de búsqueda local es adecuado, debido a que el tamaño del espacio de búsqueda que almacena en todo momento es mucho más pequeño que el que se estaría guardando en un método global.

En un segundo nivel, lo que se hace es utilizar múltiples veces la búsqueda A* para encontrar la distancia más corta entre los distintos productos listados por el usuario.

Para este programa, y por simplicidad, se partió desde el algoritmo creado para el Ejercicio 2, por lo que muchas de las funcionalidades se pasarán por alto, ya que están repetidas. Primero se evaluó posibilidad de encadenar múltiples caminos A* con varios productos y realizando recorridos cerrados, y luego se implementó el Temple Simulado, que funciona en este caso probando muchas combinaciones posibles de "orden de picking de productos".

Cabe destacar que hay algunas pequeñas diferencias, como el hecho de importar dos librerías que tienen propósitos bastante sencillos pero indispensables, así como renombrar los "nodos", para ahora hablar de "espacios". Esto pareció más acorde para distinguir bien los distintos objetos.

```
# Optimizacion Hibrida:  Temple Simulado + A*
#     "Orden de retiro de los productos" + "Camino para llegar a cada producto"

import random
import math

#CLASES Y FUNCIONES USADAS

class Espacio(object):
    def __init__(self, x, y, alcanzable):
        self.alcanzable = alcanzable
        #
        self.x = x
        self.y = y
        #
        self.padre = None
        #
        self.Gn = 0  #Gn es el costo de moverse al siguiente espacio
        self.Hn = 0  #Hn es la Heuristica... En este caso, la Distancia de Manhattan es apropiada
        self.Fn = 0
```

Se vuelve a utilizar la Clase "Busqueda_Aestrella" con sus respectivos métodos, por lo cual no se repetirá esa parte. Como diferencias, se puede mencionar además que se dejaron fijos (pueden cambiarse simplemente comentando una línea) el tamaño del almacén (en 2x3 estantes, como se ve en el esquema de ejemplo) y el espacio inicial (en x=0,y=0).

```
def obtener_adyacentes(self, espacio):
    espacios = []
#Movimiento Axial:
    for i in range (espacio.x -1 , espacio.x +1 +1):
        for j in range (espacio.y -1 , espacio.y +1 +1):
            if (i != espacio.x) or (j != espacio.y):                                    #si no es el espacio mismo...
                if ((i == espacio.x) or (j == espacio.y)):                              #si no es un espacio diagonal...
                    if not ( (i < 0) or (j < 0) or (i > self.almacen_x-1) or (j > self.almacen_y-1) ): #si no excede los limites del almacen...

                        espacios.append(self.obtener_espacio(i , j))
                        #obtengo un espacio de la lista TOTAL de elementos del almacen y lo agrego a la lista "espacios"
    return espacios

#Movimiento Diagonal:
"""
    for i in range (espacio.x -1 , espacio.x +1 +1):
        for j in range (espacio.y -1 , espacio.y +1 +1):
            if (i != espacio.x) or (j != espacio.y):
                if not ( (i < 0) or (j < 0) or (i > self.almacen_x-1) or (j > self.almacen_y-1) ):

                    espacios.append(self.obtener_espacio(i , j))
    return espacios"""
```

Por otro lado, los movimientos permitidos ahora son axiales; sólo se puede mover dentro del almacén siguiendo direcciones puramente horizontales o verticales.

Finalmente, se ha cambiado el nombre de “Resolver” y se ha agregado una función que se utiliza para un atributo nuevo de la Clase mencionada anteriormente, llamado “caminos_sumados”, que se utiliza para obtener la extensión combinada de las distancias entre múltiples productos.

```
  def sumando_caminos(self):
      return self.caminos_sumados


  def Resolver_Astar(self):
      self.lista_abiertos.append(self.comienzo)

      while len(self.lista_abiertos):

          self.ordenar_segun_Fn(self.lista_abiertos)

          espacio= self.lista_abiertos.pop(0) #saca el elemento de la lista "para verlo"      #Tomo 1 "nodo espacio" para trabajar con El

          if espacio not in (self.lista_cerrados):
              self.lista_cerrados.append(espacio) # Agregamos el espacio a la Lista Cerrada para no recorrerlo de nuevo

          if espacio is self.final:
              self.presentar_camino()
              break

          espacios_adyac= self.obtener_adyacentes(espacio)    #obtengo una lista de nodos adyacentes
          for adyac in espacios_adyac:
              if adyac.alcanzable and adyac not in self.lista_cerrados:
                  if (adyac.Fn, adyac) in self.lista_abiertos:
                      if adyac.Gn > espacio.Gn + 1:
                          self.espacio_actual(adyac, espacio)
                  else:
                      self.espacio_actual(adyac, espacio)
                      self.lista_abiertos.append(adyac)
```

Para el Temple Simulado, se ha añadido una nueva Clase, cuyas instancias se utilizan para representar los diferentes estados posibles.

```
class Picking(object):                  #Objetos que el Temple Simulado analizará. Cada uno es un posible camino para buscar los k productos
    def __init__(self, orden):
        self.orden_productos = orden    #debe recibir una lista
        #
        self.padre = None
        #
        self.E = 10000
```

En el Main, la “creación del almacén” es similar a la usada en el ejercicio anterior, “así como la generación de espacios” y las “condiciones iniciales”. Pero ahora,lo que se toma es una lista de productos y, antes, su cantidad. Luego, se presentan en pantalla con sus respectivos espacios objetivos delante de ellos.

```
if  x_inic>ancho-1 or x_inic<0 or y_inic>prof or y_inic<0:
    print("\nLa posicion inicial no puede estar fuera del almacen.\n")
else:
    if almacen[y_inic][x_inic] > 0:
        print("\nLa posicion inicial no puede estar dentro de un estante.\n")
    else:
        cant_prod= int(input("\nIngrese la cantidad de productos distintos que desea buscar: "))
        for k in range(0,cant_prod):
            print("\nIngrese el numero del espacio del producto", k, "que desea buscar: ");
            lista_prod.append(int(input()));
        print("\nLista de productos:",lista_prod);

        #Presentacion de los lugares objetivos:
        for k in range(0,cant_prod):
                for y in range(0,prof):
                    for x in range(0,ancho):
                        if almacen[y][x] == lista_prod[k]:
                            if (almacen[y][x]%2)==0:
                                xs_objetivos.append(x+1)
                                ys_objetivos.append(y)
                            else:
                                xs_objetivos.append(x-1)
                                ys_objetivos.append(y)

        for k in range(0,cant_prod):
            print("\nEn la matriz del Almacen, el espacio ubicado frente al producto", lista_prod[k], "esta en la fila", ys_objetivos[k],"y
        print("\nEstos espacios son los destinos finales y, para recorrer el almacen y llegar a ellos,");
        print("NO se permiten movimientos oblicuos");
        print("\nInicio del Temple:\n")
```

La Resolución del Temple comienza con la definición de una “Temperatura”, y una mezcla aleatoria de las lista de productos ingresada. Luego, la lista mezclada se utiliza para crear el primer estado. Éste es un objeto tipo Picking y uno de sus atributos es la lista de productos que lo distingue.

```
#RESOLUCION
        t= 0
        Tinic= 4000
        Temperatura= Tinic

        random.shuffle(lista_prod)

        estado= Picking(lista_prod)     #actual <-- HACER-NODO(ESTADO-INICIAL[problema])
```

Se inicia un bucle en el cual la Temperatura va descendiendo rápidamente. Luego de múltiples pruebas, se decidió que un descenso Lineal de esta variable era demasiado lento y poco fructífero en sus primeras iteraciones. Es por ello que se optó por el valor experimental mostrado abajo. Éste hace que, cada 2 iteraciones, la Temperatura disminuya a la mitad.

```
#Inicio del temple:
    while Temperatura > 0.1:
        t+= 1
        print("iteracion t=",t)
        deltaE= 0
        Emejor= 0
        Temperatura= Temperatura/(pow(2, 0.5))
```

Este bucle, la parte principal del Temple, se está conformado por varias secuencias bien definidas. La siguiente es el proceso de buscar el mejor camino del picking de la lista del "estado" que se analiza actualmente. Se aclara que por "picking", se hace referencia al proceso de buscar el primer elemento de la lista, luego seguir con los demás en orden y, finalmente, volver al punto de origen. Se almacena también la longitud del recorrido como una variable Energía, debido a que este algoritmo de búsqueda utiliza ese concepto para encontrar los estados mejores. Éstos son aquellos cuya E es menor. En el problema actual, la Energía representa la extensión de un camino de picking.

```
#inicio picking 1
    print("\n\n\nMejor camino hasta ahora:")
    print("Se recogeran en este orden:", estado.orden_productos)   #Muestra la lista de picking que se probara
    camino_total= 0          #Utilizado para sumar los caminos mas cortos entre cada producto (se usa para A*)
    x_inic= x_origen
    y_inic= y_origen
    xs_objetivos= [];       #Reinicio de los arrays que contienen las posiciones x y y de cada espacio
    ys_objetivos= [];       #correspondiente a cada producto ingresado
    for k in range(0,cant_prod):
        for y in range(0,prof):
            for x in range(0,ancho):
                if almacen[y][x] == estado.orden_productos[k]:
                    if (almacen[y][x]%2)==0:
                        xs_objetivos.append(x+1);
                        ys_objetivos.append(y);
                    else:
                        xs_objetivos.append(x-1);
                        ys_objetivos.append(y);
```

```
  for k in range(0,cant_prod):
      x_objetivo= xs_objetivos[k];
      y_objetivo= ys_objetivos[k];

      camino_total= Camino_Astar(almacen, ancho, prof, x_inic, y_inic, x_objetivo, y_objetivo, camino_total)


      x_inic= xs_objetivos[k];
      y_inic= ys_objetivos[k];

  x_objetivo= x_origen;   #VUELTA A CASA
  y_objetivo= y_origen;

  #Se realiza un ultimo recorrido, para volver a la posicion de origen:
  camino_total= Camino_Astar(almacen, ancho, prof, x_inic, y_inic, x_objetivo, y_objetivo, camino_total)


  estado.E= camino_total
  E1= camino_total
  print("\n El mejor camino total recorrido hasta ahora tiene una extension de:", camino_total, "\n")
  #fin del picking 1
```

Se creó la función "Camino_Astar" debido al reiterativo uso de la combinación de las siguientes funciones:

```
def Camino_Astar(almacen, ancho, prof, x_inic, y_inic, x_objetivo, y_objetivo, camino_total):
    camino_entre_productos= Busqueda_Aestrella()
    camino_entre_productos.definir_almacen(almacen, ancho, prof, x_inic, y_inic, x_objetivo, y_objetivo)
    camino_entre_productos.Resolver_Astar()
    camino_total+= camino_entre_productos.sumando_caminos()
    return camino_total
```

```
#Creacion del estado Vecino:
    listaaux= estado.orden_productos.copy()
    orden_vecino= picking_vecino(listaaux) #devuelve un "orden de picking" vecino aleatorio
    vecino= Picking(orden_vecino)
    estado_actual(vecino,estado)    #El estado actual se convierte en el padre del estado vecino
```

Luego, en el medio del bucle “while”, se busca un estado “vecino”, un nodo cuya lista de productos interna tenga una sólo una permutación entre dos de sus elementos. Se hace además que este vecino contenga un apuntador hacia el “estado” que se estaba manejando hasta ahora. Las funciones utilizadas son las mostradas a continuación:

```
def estado_actual(estado_vecino, estado):
    estado_vecino.padre = estado   #El estado (nodo) con el que trabajabamos se convierte en el "padre" --> Lista enlazada


def picking_vecino(lista):
    #Se elige el vecino aleatoriamente. Para ser considerado un estado vecino, solo puede diferir del actual en 1 permutacion
    aux= random.randint(0,len(lista)-1)
    aux2= random.randint(0,len(lista)-1)
    while aux == aux2:
        aux2= random.randint(0,len(lista)-1)
    num=lista[aux]
    num2=lista[aux2]
    lista[aux]=num2
    lista[aux2]=num
    return lista
```

Luego, y como siguiente paso dentro del bucle “while”, se procede a otro proceso de picking, similar al primero, pero que trabaja con la lista de productos que es atributo del estado “vecino”, en lugar del estado que ahora es “padre”.

A continuación, se muestra la parte final del proceso de temple. Luego del picking del vecino (2), se calcula “deltaE”, que es la diferencia de “Energías” entre los dos estados evaluados. Siguiendo la lógica del Temple Simulado, si el estado “vecino” es mejor que el original, se elige. Por otro lado, si es peor, aún puede ser elegido, pero de acuerdo a una función de probabilidad, la cual se hace más estricta y disminuye con el tiempo. El estado elegido, va entonces al inicio del bucle, para ser sometido proceso de comparación con un “vecino”. Este proceso se repite hasta la condición de salida, que estipula una Temperatura mínima.

```
        #fin del picking 2

        deltaE= E2-E1   #La "Energia" del Vecino menos la del Estado Actual
        #print("deltaE:",deltaE)
        #print("T:",Temperatura)
        Emejor=E1
        if deltaE <= 0:
            estado= vecino
            Emejor= E2
            #print("Se eligio el vecino")
        else:
            prob=(math.exp(-deltaE/Temperatura))
            #print("prob",prob)
            rand=(random.uniform(0, 1))
            if rand <= prob:
                estado= vecino
                Emejor= E2
                #print("Se eligio el vecino")
```

Al salir del bucle, se da el Temple por finalizado y se arroja la mejor solución encontrada.

```
#Saliendo del bucle, se imprime el mejor resultado:
print("\n\n\nEl mejor camino se obtiene con el picking", estado.orden_productos, ", y tiene una extension de:", Emejor)
print("\nPara verlo en detalle, suba hasta el proximo -Mejor camino hasta ahora-")
print("Se tardo", t, "iteraciones en llegar a el\n")
```

A continuación, se muestra parte de la interfaz del usuario y un ejemplo:

```
Comienzo: 0 , 0

Ingrese la cantidad de productos distintos que desea buscar: 8

Ingrese el numero del espacio del producto 0 que desea buscar:
13

Ingrese el numero del espacio del producto 1 que desea buscar:
45

Ingrese el numero del espacio del producto 2 que desea buscar:
2

Ingrese el numero del espacio del producto 3 que desea buscar:
34

Ingrese el numero del espacio del producto 4 que desea buscar:
5

Ingrese el numero del espacio del producto 5 que desea buscar:
17

Ingrese el numero del espacio del producto 6 que desea buscar:
27

Ingrese el numero del espacio del producto 7 que desea buscar:
39
```

Con su solución:

```
Mejor camino hasta ahora:
Se recogeran en este orden: [34, 5, 2, 27, 45, 13, 39, 17]

SubCamino(posiciones x,y):
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (4, 4), (5, 4), (6, 4), (6, 5)]
La extensión de este camino es:  11

SubCamino(posiciones x,y):
[(6, 5), (6, 4), (5, 4), (4, 4), (3, 4), (2, 4), (1, 4), (0, 4), (0, 3), (0, 2)]
La extensión de este camino es:  9

SubCamino(posiciones x,y):
[(0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (3, 3), (3, 2), (3, 1), (3, 0)]
La extensión de este camino es:  9

SubCamino(posiciones x,y):
[(3, 0), (3, 1)]
La extensión de este camino es:  1

SubCamino(posiciones x,y):
[(3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6), (3, 7), (3, 8), (3, 9), (3, 10), (3, 11), (3, 12)]
La extensión de este camino es:  11

SubCamino(posiciones x,y):
[(3, 12), (3, 11), (3, 10), (3, 9), (2, 9), (1, 9), (0, 9), (0, 8), (0, 7)]
La extensión de este camino es:  8

SubCamino(posiciones x,y):
[(0, 7), (0, 8), (0, 9), (1, 9), (2, 9), (3, 9), (3, 8)]
La extensión de este camino es:  6

SubCamino(posiciones x,y):
[(3, 8), (3, 9), (2, 9), (1, 9), (0, 9), (0, 10)]
La extensión de este camino es:  5

SubCamino(posiciones x,y):
[(0, 10), (0, 9), (0, 8), (0, 7), (0, 6), (0, 5), (0, 4), (0, 3), (0, 2), (0, 1), (0, 0)]
La extensión de este camino es:  10

 El mejor camino total recorrido hasta ahora tiene una extension de: 70
```

Este problema funciona aceptablemente, pero tiene gran cantidad de fallos en su implementación. Se pueden mencionar:

- La interfaz es mala, ya que, por cómo fue planteado, es muy difícil solicitar la impresión sólo del mejor camino logrado.
- Secciones de código extensas se repiten en el Main. Esto no ha sido solucionado debido a la complicación que a veces se genera con el pasaje de listas en python, y que a veces implica la copia de la lista para trabajar, y no la utilización de ella en sí.
- El problema principal es que la implementación involucra a dos procesos de "picking" distintos dentro del bucle "while". Esto se ha considerado grave, ya que utilizando Nodos como en un grafo o una lista enlazada, la solución debió ser más sencilla y por otro camino.

Debido a la conclusión formulada, es que se seguirá trabajando sobre este problema, el cual, a pesar de estar resuelta, utiliza incorrectamente los recursos.

EJERCICIO 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
- Se debe generar un conjunto de órdenes ficticias, simulando órdenes reales que el almacén tendría que satisfacer. Las órdenes deberían tener cantidades distintas de artículos e incluir distinto mix de artículos (SKU, Stock Keeping Unit) cada una

```
def templeAstar(Matrix, productos):
    ini = [0,0,0]
    buscar=[33, 1 ,3,25]
    genomas=[]

    tem=temple(100,buscar,0)

    buscar1=[37,33, 1 ,3,25,22,21,4,10,4,2,11,40]
    buscar2=[25,10,35,8]
    buscar3=[11,18,40,31,5]
    buscarList=[]
    buscarList.append(buscar1)
    buscarList.append(buscar2)
    buscarList.append(buscar3)
```

Una vez que tengamos nuestro código de Temple Simulado y A* andando, los separamos en un archivo de código distinto llamado templeAstar.py, la cual será una función que recibirá como parámetros una Matriz(las dimensiones del almacén) y una lista Productos(esta lista está asociada a productos, que tiene como atributos un nombre, ejemplo "1" como producto 1, y un nodo, el cual contiene el valor de posición,gn y hn). Además se generaron varias órdenes de pedido ficticias.

Otra cosa a tener en cuenta es que este algoritmo genético será con permutaciones, ya que la cantidad de productos es fija, y no podemos permitir ni la sobre especificación ni la sub especificación. No podemos permitir que 2 genes tengan un mismo valor ni que no estén asignados(productos), ya que si hubieran 2 genes con el mismo valor nuestra solución sería inválida.

```
        print("el mejor orden de los productos es ")
        for ordprod in tem.listAnt:
            print(str(ordprod))

        print("La energia para estos fue: "+ str(tem.Eant))
        #input()
        fitness+=tem.Eant

    fitness=fitness/len(buscarList)
    print("el promedio de energia, o fitness total de ordenes es "+str(fitness))
    genoma=gen(productos,fitness)
    return genoma
```

Para evaluar la calidad del almacén, su fitness, la medida evaluada fue el **promedio de los costos históricos** para esa configuración del almacén, por lo cual la función nos devuelve la lista de los productos (orden en el cual fueron ubicados) y su valor de Fitness con respecto a las órdenes. En el modelo (la clase) se llama genoma pero esta, por afuera de la función será instanciada como un individuo de la población para la resolución del problema.

```
for pob in range(0, poblacion+1):
    cont=0
    productos=[]
    ordenProducto=random.sample(range(42), 42)
    Matrix = [[0 for i in range(x)] for j in range(y)]
    for j in range(0, y): #de 0 a y-1
        for i in range(0, x):
            if (((j%5)==0) or (j==(y-1))):
                Matrix[j][i]=0
            else:
                if((i%3)==0):
                    Matrix[j][i]=0
                else:
                    Matrix[j][i]=1
                    nod1=node([j,i],0,100000000)
                    prod1=producto(nod1,ordenProducto[cont]) # se le asocia el nodo y el nombre
                    productos.append(prod1)
                    cont=cont+1

    genoma=templeAstar(Matrix,productos)
    genomas.append(genoma)

k=0.35
```

Lo primero a realizar en nuestro algoritmo fue crear la (población)i , que consta de varios individuos, en este caso 5, la creación de individuos se realizó al azar (se puede observar en ordenProducto que se utilizó random.sample, esta crea 42 números del 0 al 41 de tal forma que no se repitan) los asignamos a su respectiva posición, y hacemos que esta sea un parámetro de la función templeAstar, se añade a genoma,que luego será agregada a lista genomas (esta lista contiene la población entera, tanto su orden de Producto como su valor de fitness correspondiente)

Se debe aclarar el método de selección. El que se utilizó por parte de los alumnos fue “los k mejores” o selección por ranking. Este **no es la mejor selección,** ya que nos podría llevar a un **máximo local** de las configuraciones del almacén, por lo cual es una de las tantas mejoras pendientes del algoritmo, al final se aclara formas de optimizar a este.

El k debe ser menos a 0.5 y es un parámetro del modelo, nos permite una variedad, pero no una variedad extrema ya que no se compara con la selección probabilística.

```
for x in range(0, len(genomas)):
    for y in range(0, len(genomas)):
        if(genomas[y].get_fitness()>genomas[x].get_fitness()):
            aux=genomas[y]
            genomas[y]=genomas[x]
            genomas[x]=aux

for gen1 in genomas:
    print("productos: ")
    for prod1 in gen1.get_individuo():
        print(prod1.get_name())
    print("fitness es: "+str(gen1.get_fitness()))
```

Ordenamos los individuos por su valor de fitness, de menor a mayor ya que queremos el menor costo de fitness posible (menor cantidad de paso nos da la idea de que los productos están mejor ubicados por lo cual requiere menos movimientos para las órdenes históricas)

```
while(poblacion>1):
    puntosCorte=random.sample(range(42), 2) #cruce de orden
    sorted(puntosCorte)
    pos1=puntosCorte[0]
    pos2=puntosCorte[1]
    #individuo1=list(genomas[0].get_individuo()) #.copy()
    #individuo2=list(genomas[1].get_individuo()) #.copy()
    individuo1=[]
    individuo2=[]
    norep1=[]
    norep2=[]
    genomas2=[]
    individuos=[]
    for x in range(0,round(poblacion-1)):
        #try:
            prod1=copy.deepcopy(genomas[x].get_individuo()) #.copy()
            prod2=copy.deepcopy(genomas[x+1].get_individuo()) #.copy()
            individuo1=copy.deepcopy(prod1) #.copy()
            individuo2=copy.deepcopy(prod2) #.copy()

            print("prod1 antes de cruzamiento")
            for i in range(0, len(prod1)):
                print(prod1[i].get_name())

            for i in range(pos1,pos2):
                norep1.append(prod1[i].get_name()) # elementos q no se deben repetir
                norep2.append(prod2[i].get_name())
                individuo1[i].set_name(prod2[i].get_name())
                individuo2[i].set_name(prod1[i].get_name())

            #cruce de orden
```

Una vez obtenida la (población)i, procedemos a generar las futuras poblaciones (poblacion)i+1, para esto procedemos a realizar una copia local de las variables refiriéndose prod1 como padre 1, prod2 como padre2, individuo1 seria nuestro hijo1, o individuo de la (poblacion)i+1 e individuo2 como hijo2.

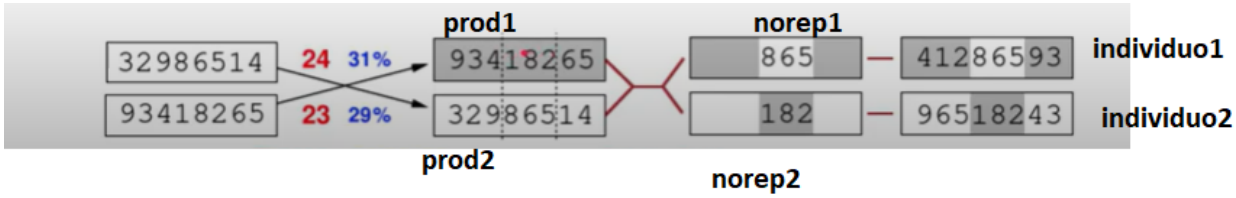
Otra cosa a destacar es que los hijos se generaron de la siguiente forma, padre1 y padre2 generan al individuo1 e individuo2, pero en la siguiente iteración será padre2 y padre3, estas generarán individuo3 e individuo4, es decir la cantidad de individuos en la población se mantiene.Solo se reduce solo porque se eligen los k mejores en cada iteración al finalizar el entrecruzamiento, mutación y re-evaluación del valor de fitness.

```
92              #cruce de orden
93              cont=0
94              for i in range(1, len(prod1)+1):
95                  if(not prod1[i-1].get_name() in norep2):
96                      if((pos2+i)<len(prod1)):
97                          individuo1[pos2+i].set_name(prod1[i-1].get_name())
98                      else:
99                          if (not (cont==pos1)):
100                             individuo1[cont].set_name(prod1[i-1].get_name())
101                             cont+=1
102             cont=0
103             for i in range(1, len(prod2)+1):
104                 if(not prod2[i-1].get_name() in norep1):
105                     if((pos2+i)<len(prod2)):
106                         individuo2[pos2+i].set_name(prod2[i-1].get_name())
107                     else:
108                         if (not (cont==pos1)):
109                             individuo2[cont].set_name(prod2[i-1].get_name())
110                             cont+=1
111
112
113
```

El crossover que se utilizó es **cruce de orden**, de tal forma que preservamos el orden relativo. En la imagen arriba de esta última se ve que se genera norep1 y norep2, y los 2 puntos de cruce, pos1 y pos2 a través de random.sample de tal forma que no se repitan y se generen aleatoriamente.

Los genes que se encuentran entre los 2 puntos de corte se realizaron también en la imagen que está arriba de esta última.

En la siguiente imagen se detalla el cruce y los nombres de las variables por simplicidad de comprensión:



Tenemos prod1 y prod2, el individuo1 tendrá los genes centrales de prod2 y el individuo2 los genes centrales de prod1. De allí lo que se realiza en las funciones for de la imagen de arriba es partiendo desde el extremo derecho pos2, el primer valor a la derecha del corte del extremo derecho será el primer valor del padre original, y así se progresa sucesivamente siempre y cuando no sea un valor del gen central, para esto se utilizó “if (not in rep2)” ya que se agregará el gen siempre y cuando no esté en los valores centrales, ya que si se agregara tendríamos valores repetidos, por ende productos repetidos y al tener un espacio limitado como resultado tendríamos productos inexistentes.

```
#Mutacion x intercambio
puntosMutar=random.sample(range(42), 2)
aux=individuo1[puntosMutar[0]].get_name()
individuo1[puntosMutar[0]].set_name(individuo1[puntosMutar[1]].get_name())
individuo1[puntosMutar[1]].set_name(aux)

aux=individuo2[puntosMutar[0]].get_name()
individuo2[puntosMutar[0]].set_name(individuo2[puntosMutar[1]].get_name())
individuo2[puntosMutar[1]].set_name(aux)

print("individuo post cruzamiento y mutacion")
for i in range(0, len(individuo1)):
    print(individuo1[i].get_name())
input()
print("individuo2 post cruzamiento y mutacion")
for i in range(0, len(individuo2)):
    print(individuo2[i].get_name())

individuos.append(individuo1)
individuos.append(individuo2)
```

Por último se realizó una **mutación por intercambio** que no es más que realizar un swapping entre dos valores de los individuos.Luego se agregan los individuos a una lista llamada individuos.

```
for ind in individuos:
    #genoma=templeAstar(Matrix,ind)
    genomas2.append(templeAstar(Matrix,ind))
genomas.clear()
genomas=copy.deepcopy(genomas2)
for x in range(0, len(genomas)):
    for y in range(0, len(genomas)):
        if(genomas[y].get_fitness()>genomas[x].get_fitness()):
            aux=genomas[y]
            genomas[y]=genomas[x]
            genomas[x]=aux

poblacion=round(poblacion*k)
```

Al ya tener la (poblacion)i+1 lo que debemos hacer es dos acciones, una re-evaluar la función de fitness y dos ordenarla nuevamente ya que no sabemos la que tiene menor valor. Una vez hecho esto se seleccionan los k mejores y vuelve a iterar, el algoritmo termina una vez que quede 1 solo individuo el cual se considerará como el mejor.

```

individuo1=genomas[0].get_individuo().copy()
for i in range(0, len(individuo1)):
    print(individuo1[i].get_name())
print("el mejor valor de fitness es: "+str(genomas[0].get_fitness()))
```

Una vez salga el algoritmo al haber 1 solo individuo dice su orden de producto y su valor de fitness

En resumen, el algoritmo y el problema se pudo solucionar pero el algoritmo realizado tiene muchos defectos y hay muchas maneras de optimizarlo. Una seria implementar que el valor de Fitness sea una probabilidad. Otra mejora sería mejorar el temple Simulado y encontrar el mejor valor de T que se adapte al modelo. También se puede optimizar el algoritmo de A* entre muchas otras.

EJERCICIO 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)
- No puede sobrepasarse la capacidad de cada máquina en un momento determinado: una máquina solo puede realizar una tarea en un momento determinado

Los alumnos se basaron en el algoritmo recursivo de backTracking del libro de Stuart “Un enfoque Moderno” 2° edición

```
función BÚSQUEDA-CON-VUELTA-ATRÁS(psr) devuelve una solución, o fallo
  devolver VUELTA-ATRÁS-RECURSIVA({}, psr)

función VUELTA-ATRÁS-RECURSIVA(asignación, psr) devuelve una solución, o fallo
  si asignación es completa entonces devolver asignación
  var ← SELECCIONA-VARIABLE-NOASIGNADA(VARIABLES[psr], asignación, psr)
  para cada valor en ORDEN-VALORES-DOMINIO(var, asignación, psr) hacer
  si valor es consistente con asignación de acuerdo a las RESTRICCIONES[psr] entonces
    añadir {var = valor} a asignación
    resultado ← VUELTA-ATRÁS-RECURSIVA(asignación, psr)
    si resultado ≠ fallo entonces devolver resultado
    borrar {var = valor} de asignación
  devolver fallo
```

Figura 5.3 Un algoritmo simple de vuelta atrás para problemas de satisfacción de restricciones. El algoritmo se modela sobre la búsqueda primero en profundidad recursivo del Capítulo 3. Las funciones SELECCIONA-VARIABLE-NOASIGNADA y ORDEN-VALORES-DOMINIO pueden utilizarse para implementar las heurísticas de propósito general discutidas en el texto.

Lo primero que se realizó al momento de plantear el problema fue modelizar los datos, para esto se utilizaron listas y se explicará en el siguiente caso que significan

```python
from machine import Machine
from cspStuart import backTrackingRecursive

X=[5,15,10,30] #T0=5m T1=15m T2=10m T3=30, en esta caso seria X, nuestro conjunto de variables fin.
limit=0
D=[]
for i in range(0,len(X)):
    limit=limit+X[i] #limit max tiempo de tarea, no podria durar mas q la ssuma por separado
    #incluso si tenemos en cuenta q cada tarea usan todas las misma maquina

for i in range(0,limit):
    D.append(i) #Dominio de nuestro CSP

C_P=[[1,2],[3,0],[3,2]] #C restricciones precedencias T1 antes q T2,
                        # T3 antes q T0 y T3 antes q T2

M1=Machine("M1")
M2=Machine("M2")
                #restricciones maquinas
C_M=[M1,M1,M2,M2] #T0 y T1 usa M1 T3 y T4 usa M2
historial=[]
historialTiempo=[]
backTrackingRecursive(X,D,C_P,C_M,historial,historialTiempo)

```

Para el problema se utilizaron los datos del powerPoint de CSP de la cátedra.

Tenemos las Variables X, con sus respectivas tareas y sus tiempos para completarlas. Las tareas están referidas dependiendo del índice de la lista. En X[0] se encuentra la tarea T0, X[1] T1, y así sucesivamente.

Luego creamos nuestro dominio D, el cual su periodo será en minutos (1m) por lo cual cada tarea en el algoritmo será asignada en cada minuto en paralelo, ya que no todas pueden usar la misma máquina.
Luego creamos nuestras restricciones, en este problema tenemos restricciones por parte de las máquinas y por parte del orden de las tareas(procedencia)

- C_M (restricciones máquina) están también indexadas respecto a las tareas, C_M[0]=M1, significa que la tarea T0 usa la máquina 1 (M1)

- C_P (restricciones procedencia), se lee de la siguiente forma, la tarea T1<T2, T1 debe realizarse antes que T2, T3<T0 la tarea T3 debe realizarse antes que la T0, T3<T2 la tarea T3 debe realizarse antes que la T2.

machine.py > Machine

```python
class Machine():
    def __init__(self,name):
        self.name=name
        self.state=False #True=Ocupada
        self.time=0
        self.task=[] #Historial maquina

    def get_name(self):
        return self.__name

    def set_name(self, name):
        self.__name = name

    def get_state(self):
        return self.__state

    def set_state(self, state):
        self.__state = state

    def get_time(self):
        return self.__time

    def set_time(self, time):
        self.__time = time

    def get_task(self):
        return self.__task

    def set_task(self, task):
        self.__task = task
```

Se creó la clase Machine principalmente para saber si la máquina está siendo ocupada en el momento.

Una vez teniendo todas las variables se define el código de búsqueda y asignación.

```python
def backTrackingRecursive(X,D,C_P,C_M,historial,historialTiempo):

    for i in range(0,len(D)):
        var=variableMasRestringida(historial,C_P) #var se refiere a tarea asignada
        consistent=consistentValue(var,X,i,C_P,C_M,historial) #i se refiere a dominioActual
        if ((consistent==True)&(var!=-1)):
            historial.append(var)
            historialTiempo.append(i)
            backTrackingRecursive(X,D,C_P,C_M,historial,historialTiempo)
        elif((consistent==False)and(var==-1)):
            print(historial)
            print(historialTiempo)

            exit()
```

Tenemos una función recursiva la cual usará las variables X, el dominio D, las restricciones C_P y C_M, un historial en el cual se añadirán las tareas ya completadas para facilitar su impresión y un historialTiempo para referirse en cuanto tiempo se completaron las tareas.

Var se referirá a la variable actual (tarea en concreto que se está realizando en el momento). La heurística utilizada es la más restringida que se explicará a continuación.

Se elige la variable, una vez elegida se analiza si es consistente, si es consistente se añade, si no lo es simplemente en ese bucle recursivo no se añadirá (no se toma en cuenta, nunca se añade). Si vemos que ya no hay consistencia y var=-1 (quiere decir que no hay más variables) significa que el algoritmo finaliza, puede ser porque no se puede añadir ninguna variable o porque el algoritmo ha finalizado

```python
def most_frequent(List):
    return max(set(List), key = List.count)

def variableMasRestringida(historial,C_P):
    listNew=[]
    try:
        for i in range(0,len(C_P)):
            if(C_P[i][0] not in historial): #asigna ejemplo T3 ya q es la mas restringida
                listNew.append(C_P[i][0])
            elif(len(listNew)==0): #si no hay ningun valor x asignar empieza con las q restringen
                if(C_P[i][1] not in historial):
                    listNew.append(C_P[i][1])

        return most_frequent(listNew)
    except:
        print("Todos los elementos ya fueron completados en historial")

        return -1
```

La variable más restringida simplemente cuenta cual es la variable que más se repite en C_P, las restringidas son las que están en C_P[i][0] y luego están las que restringen, que se encuentran en C_P[i][1]. Se toma la variable siempre y cuando no se haya completado, es decir, que no se encuentre en historial. Si ya se añadieron todas las variables restringidas, empieza con las que restringen.

Una vez todo completado devuelve -1 el cual se refiere a que ya han sido completada todas las tareas.

```
#Xact es el periodo actual
def consistentValue(var,X,tiempo,C_P,C_M,historial):
    estado=False
    for i in range(0,len(C_P)): #empezamos con las procedencias ya q son lo q mayor restricciones tiene
        if var == C_P[i][1]:
            if C_P[i][0] not in historial: #tiene q estar en historial, si no esta es inconsistente
                estado=False

    if ((var not in C_M[var].task) & (tiempo>=C_M[var].time)):
        C_M[var].state=True

    if ((var not in C_M[var].task) & (C_M[var].state==True)): #tarea no esta terminada pero
                                                              # maq sigue ocupada
        tiempoTask=0
        for tarea in C_M[var].task:
            tiempoTask+=X[tarea]
        if (tiempo-tiempoTask>=X[var]): #cuando se complete la tarea
            C_M[var].task.append(var) #variable local de tareas terminadas x maquina
            C_M[var].time=tiempo #ultimo tiempo de uso
            #C_M[var].time.append(tiempo)
            C_M[var].state=False #ya que se sumo el tiempo de dominio y se realizo la tarea
            #historial.append(var)
```

Ahora se analizará el caso cuando una variable es consistente. Una variable que restringe sólo puede ser consistente si la tarea que restringía fue completada, si esta no se encuentra en historial entonces no es consistente.

Luego se procede a la ocupación de la máquina. Si la máquina no está siendo ocupada, y el tiempo actual es mayor o igual al tiempo que ha sido ocupada entonces esta puede ocuparse.

Finalmente, si la tarea no ha sido completada por la máquina, y la máquina está siendo ocupada, y si el tiempo actual es mayor al tiempo de tarea, entonces puede completarse.

```
            #historial.append(var)
            estado=True
        elif(tiempo-tiempoTask<X[var]):
            C_M[var].state=False #esto se triggea a True false todo el tiempo mientras no este
                                 #completo el tiempo en Dominio

    else:
        estado=False

    return estado
```

Si el tiempo actual es menor de lo que demora la tarea, la máquina se desocupará. Este es el comienzo y fin de cada periodo (en minutos para este problema).

En conclusión, hay muchas maneras de optimizar este código, un caso sería implementar el código optimizados de búsqueda Recursiva o el algoritmo AC-3 ya que la complejidad espacial es exponencial. Ambos códigos se encuentran en el libro de Stuart.