UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Tarea 02 - Representación de Soluciones

ASIGNATURA

Cómputo Evolutivo

PROFESOR

Oscar Hernández Constantino

AYUDANTE

Malinali Gónzalez Lara

ALUMNOS

Carlos Emilio Castañon Maldonado & Dana Berenice Hérnandez Norberto

Ejercicio 1. Representación binaria para números reales

Existen diferentes métodos para codificar números reales, por ejemplo: números de punto flotante, representación binaria, representación de punto fijo y código de gray.

a) Describe e implementa el algoritmo de representación binaria para mapear (codificar y decodificar) números naturales. Menciona cuántos números son representables con m bits.
Por ejemplo, la función (método) puede tener la siguiente firma:

```
int [ ] codifica_aux( int n, int nBit )
```

- n es el número natural a codificar en binario
- $nBit$ es la cantidad de bits a utilizar (m)

La cantidad de números que pueden representarse con m bits son 2^m debido a que solo tenemos 2 números, el cero y el uno, por lo tanto el número de combinaciones que podemos hacer con estos números depende igualmente de la cantidad de lugares que tenemos, es decir, de los bits que tengamos o que ingresemos. Programamos las siguiente funciones en python para convertir a un número decimal a binario y a un número binario a decimal.

```
def codifica_aux(decimal,nBit):
    #Utilizamos como parametro un número natural en decimal y la
    #cantidad de bits que queremos utilizar en el arreglo para #decodificar
    binary = np.zeros(nBit,dtype=int)
    i = nBit - 1
    while decimal > 0:
        binary[i] = decimal % 2
        #Establecemos ahora al número natural como la división
        #entera del numero anterior
        decimal //=2
        i-=1
    return binary
```

```
def decodifica_aux(binary):
    #Utilizamos como parametros un arreglo de bits e inicializamos
    #un número decimal
    decimal = 0
    for i in range(len(binary)):
        #Utilizando el algoritmo para convertir de binario a decimal
        decimal += binary[i]*2**(len(binary)-i-1)
    return decimal
```

b) Si se requiere una representación uniforme de números reales en el intervalo $[a, b]$
¿Cómo se generaliza la representación binaria para mapear números reales?, ¿Cuál es la máxima precisión?
Sugerencia. Considera lo siguiente:
Se tiene una partición uniforme del intervalo $[a, b]$, por ejemplo:

$$10 = x_0 < x_1 < x_2 < ... < x_n = 20$$

Si hacemos $n = 10$, tenemos una precisión de 0 dígitos decimales (ya que $x\_i = 10 + i$, es decir, la partición uniforme sólo consideraría números enteros).
En cambio, si $n = 100$, tenemos una precisión de 1 dígito (p. e. $x\_25 = 10 + 25 * (0.1) = 12.5$), pero con $n = 100$ no es posible representar 12.52.
¿Qué relación hay entre n, el número de bits y la precisión de la representación?

Considerando que la cantidad de números representables con mBits es de 2^m eso quiere decir que con una mayor cantidad de bits, podemos representar una mayor cantidad de números reales en un intervalo de una partición uniforme. Obteniendo una precisión de

$$\frac{b-a}{2^m}$$

Es decir, la longitud del intervalo entre el número de reales que podemos representar en el mismo.
De esta forma generalizamos la representación binaria en un intervalo cualquiera, haciendo la partición uniforme tan fina como queramos, en particular podemos tomarla desde cero hasta algún número muy grande con el fin de podernos acercar más al valor en binario del número real.

c) Implementa un algoritmo que codifique números reales en una representación binaria, considerando una partición uniforme sobre un intervalo $[a, b]$, utilizando m bits.

* La implementación debe usar la función del inciso a).
Por ejemplo, la función (método) puede tener la siguiente firma:

```
int [ ] codifica( double x, int nBit , double a, double b )
```

- x representa es el número real que se desea codificar en binario
- $nBit$ es el número de bits a utilizar (el tamaño que debe tener el arreglo que se devuelve)
- a, b son los valores del intervalo

Usando un razonamiento similar al inciso anterior, calculamos la precisión y de esa forma podemos asignarle una posición en nuestro intervalo con respecto a la longitud tiene nuestro número real desde su cota inferior y justamente esa proporción de la distancia entre nuestro número a evaluar con respecto a la distancia total de nuestro intervalo, por lo tanto, expresamos la posición de nuestro intervalo como

$$\frac{x-a}{b-a}2^m$$

Con base en lo anterior, asignamos la posición del número real en el intervalo y aplicamos la función que realizamos en el primer inciso, como si fuera un número natural.

```python
def codifica(x,nBit,a,b):
    # x es el número real que queremos codificar
    # nBit es la cantidad de Bits con la que queremos representar el
    #número real
    # a es el límite inferior del intervalo donde se encuentra el
    #número a evaluar
    # b es el límite superior, es importante que a sea menor que a
    # Calculamos la precisión dividiendo la longitud del intervalo
    #entre el número de reales que podemos representar con nBits
    precision = (b-a)/(2**nBit)
    #Checamos que el número que queremos codificar se encuentra
    #dentro del intervalo
    x= max(a,min(b,x))
    #Le asignamos una posición en el intervalo a x
    index = int((x-a)/precision)
    #Convertimos a binario usando la función del inciso a
    binary = codifica_aux(index,nBit)
    return binary
```

d) Implementa un algoritmo para decodificar los vectores de bits como un número real.
Por ejemplo, la función (método) puede tener la siguiente firma:

```
double decodifica( int x_cod[ ], <int nBits>, double a, double b )
```

- ➢ *intx_cod*[]: es un arreglo que representa el vector de bits que representa la codificación binaria de la solución.
- ➢ *int nBits*: es el número de bits que se utilizará para representar un número real (corresponde con el tamaño del vector *x_cod*; en algunos lenguajes este parámetro podría ser opcional).
- ➢ a, b: Se debe cumplir que el número devuelto es un valor en el intervalo $[a, b]$

En este punto realizamos un proceso inverso a la función anterior.

```python
def decodifica(x_cod,nBit,a,b):
    #x_cod ahora es el arreglo de bits
    precision = (b-a)/(2**nBit)
    #Hacemos un proceso inverso a la función de codificación
    index = decodifica_aux(x_cod)
    #Establecemos al número decimal como x y lo definimos conforme a
    #la función anterior
    x = a+ index *precision
    return x
```

e) Implementa las funciones necesarias para codificar y decodificar vectores de números reales
Por ejemplo, la función (método) puede tener la siguiente firma:

```
int [ ] codifica( double x[ ], <int dim_x>, <int nBits>, double a, double b)

double [ ] decodifica( int x_cod[ ], int dim x, <int nBits>, double a, double b)
```

Ya que tenemos las funciones anteriores, podemos utilizarlas para crear vectores con un ciclo que en cada iteración codifique o decodifique cada una de las entradas del vector a evaluar.

```
def codifica_v(x,nBit,a,b):
    #En este caso x es un vector de números reales que queremos
    #codificar
    binary = list()
    for i in x:
        binary.append(codifica(i,nBit,a,b))
    return binary
```

```
def decodifica_v(x_cod,nBit,a,b):
    decimal = list()
    #x_cod es un vector de arreglos de bits para convertir a decimal
    for i in x_cod:
        decimal.append(decodifica(i,nBit,a,b))
    return decimal
```

Ejercicio 2. Búsqueda por escalada

El problema de COLORACIÓN en grafos consiste en encontrar el mínimo número de colores que se requieren para asignar a cada vértice un color, de manera que dos vértices adyacentes no tengan el mismo color.
Considera el siguiente esquema de codificación de ejemplares.

a) Implementa la lectura de archivos para leer información de ejemplares (instancias) del problema. Los archivos seguirán el siguiente esquema de codificación:

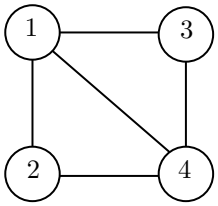
- El archivo puede empezar con comentarios, que son líneas que empiezan con el carácter c (y deben ser ignoradas).
- Justo después de los comentarios, estará la línea "p edge nVertices nAristas" que indica que es un ejemplar que tiene un total de nVertices número de vértices, y nAristas número de aristas.
- A continuación, siguen nAristas líneas, de la forma "e x y" que indican la lista de aristas. x, y son los índices de los vértices correspondientes. (los índices de los vértices siempre empiezan en 1).

Ejemplo:

```
c Archivo: prueba1.col
```

```
p edge 4 5
e 1 2
e 1 3
e 1 4
e 2 4
e 3 4
```

El ejemplar correspondiente es:



Para lograr lo anterior, implementamos la siguiente funcion la cual lee los datos codificados del archivo `.col` y de ella nos brinda información tal como el numero de aristas, numero de vértices, el nombre de todos los vértices y las aristas de la gráfica:

```python
def leer_ArchivoCol(archivo):
    vertices = set()
    aristas = []

    with open(archivo, 'r') as f:
        for linea in f:
            # Ignorar comentarios
            if linea.startswith('c'):
                continue

            # Leer información de la instancia
            if linea.startswith('p'):
                _, _, n_vertices, n_aristas = linea.split()
                n_vertices = int(n_vertices)
```

```
                n_aristas = int(n_aristas)
            elif linea.startswith('e'):
                _, v1, v2 = linea.split()
                v1 = int(v1)
                v2 = int(v2)
                aristas.append((v1, v2))
                vertices.add(v1)
                vertices.add(v2)

    return n_vertices, n_aristas, list(vertices), aristas
# Prueba con el archivo de ejemplo y su salida
archivo = 'prueba1.col'
n_vertices, n_aristas, vertices, aristas = leer_ArchivoCol(archivo)

print("Número de vértices:", n_vertices)
print("Número de aristas:", n_aristas)
print("Vértices:", vertices)
print("Aristas:", aristas)
```

```
Número de vértices: 4
Número de aristas: 5
Vértices: [1, 2, 3, 4]
Aristas: [(1, 2), (1, 3), (1, 4), (2, 4), (3, 4)]
```

b) Describe e implementa un esquema de representación de soluciones.
Justifica la elección del tipo de representación.
Describe las características del esquema de representación de soluciones propuesto (tamaño del espacio de búsqueda, directa o indirecta, lineal o no lineal, tipo de mapeo, factibilidad de soluciones, ¿representación completa?, etc)

Sea la siguiente:

```
class SColoracion:
    def __init__(self, n_vertices, colores_asignados=None):
        self.n_vertices = n_vertices
        if colores_asignados is None:
            # Inicialmente ningún vértice tiene color asignado
            self.colores_asignados = [-1] * n_vertices
        else:
            self.colores_asignados = colores_asignados

    def asignar_color(self, vertice, color):
        # Los índices de los vértices empiezan en 1
        self.colores_asignados[vertice - 1] = color

    def obtener_color(self, vertice):
        return self.colores_asignados[vertice - 1]
```

Notemos como es que este esquema de representación contiene una lista que almacena los colores asignados a cada vértice de la gráfica, a su vez que, cada vértice se representa mediante su índice en la lista, y el color asignado a ese vértice se almacena en la posición correspondiente en la lista.

El tamaño del espacio de búsqueda en este esquema de representación depende del número de vértices en la gráfica y del número de colores disponibles para la coloración, además de que la representación es completa

ya que garantiza que todas las posibles soluciones al problema se pueden representar, en otras palabras, cada vértice recibe un color único de la lista de colores.

La representación es directa porque cada solución se corresponde directamente con una asignación de colores a los vértices de la gráfica y a su vez es lineal porque el tamaño de la solución es proporcional al número de vértices en la gráfica

c) Describe e implementa una función de evaluación para las soluciones.

La funcion de evaluación que implementaremos es la de zakharov para obtener las soluciones:

Función de Zakharov

```
def zakharov(x):
    x = pd.Series(x)
    d = len(x)
    t1 = sum(x**2)
    t2 = 0.5*sum(range(d)*(x**2))
    y = t1 + (t2**2) + (t2**4)
    return y
```

d) Describe e implementa un generador de soluciones aleatorias

En el siguiente podemos ver como es que leemos el archivo con `leer_ArchivoCol`, para de esa forma crear la gráfica correspondiente y usamos la heurística de ordenar primero los vértices según su grado (numero de vecinos) para que de esa forma podamos iniciar con los vértices de mayor grado y de esta forma asegurarnos que al ponerles un color estos no rompan el algoritmo (lo cual seria probablemente el caso si seleccionamos otra heurística), una vez hecho lo anterior iteramos sobre los vértices para asignarles un color (y de paso revisar si ese color que estamos seleccionando no lo esta ocupando alguno de nuestros vecinos).

```
def colorearGraficaConNColores(archivo):
    n_vertices, n_aristas, vertices, aristas = leer_ArchivoCol(archivo)

    # Como en el peor de los casos se necesitaran n_vertices colores,
    # vamos a asignarle ese numero de colores
    n_colores = n_vertices
    solucion = SColoracion(n_vertices)
    vertices = list(range(1, n_vertices+1))

    #Ahora ordenamos la lista de vertices con los vertices
    # de mayor a menor, iniciando por aquellos que tienen mas vecinos
    vertices.sort(key=lambda v:
        len([v1 for v1, v2 in aristas if v1 == v or v2 == v]), reverse=True)

    for vertice in vertices:
        # Asignar un color aleatorio al vértice y verificar
        que ese color no esté asignado a ninguno de sus vecinos
        color = (random.randint(1, n_colores))
        while color in coloresVecinos(solucion, vertice):
            color = (random.randint(1, n_colores))
        solucion.asignar_color(vertice, color)

    return solucion
```

e) Describe e implementa una función u operador de vecindad, acorde al tipo de representación implementado en los incisos anteriores.

Para lograr este punto primero definimos a un vecino de solucion con:

```
def generar_vecino(solucion_actual):

    vertice_a_cambiar = random.randint(1, solucion_actual.n_vertices)
    #colores_vecinos = coloresVecinos(solucion_actual, vertice_a_cambiar)


    color_asignado_actual = solucion_actual.obtener_color(vertice_a_cambiar)
    nuevo_color = color_asignado_actual
    while nuevo_color in coloresVecinos(solucion, vertice_a_cambiar):
        nuevo_color = random.randint(1, solucion_actual.n_vertices)
        solucion_actual.asignar_color(vertice_a_cambiar, nuevo_color)

    # Crear una copia de la solución actual y modificar el color
    del vértice seleccionado
    nueva_solucion = SColoracion(solucion_actual.n_vertices,
        solucion_actual.colores_asignados[:])
    nueva_solucion.asignar_color(vertice_a_cambiar, nuevo_color)
    return nueva_solucion
```

Una vez teniendo esto, realizamos una vecindad de soluciones:

```
def generarVecindad(solucion):
    vecindad = []
    for i in range(100):
        vecino = generar_vecino(solucion)
        vecindad.append(vecino)
    return vecindad
```

Una vez propuesto todo lo anterior podemos hacer uso de una busqueda en escalada para obtener una solucion en nuestra vecindad que satisfaga que el mínimo número de colores que se requieren para asignar a cada vértice un color, de manera que dos vértices adyacentes no tengan el mismo color.

```
def busquedaEscalada(archivo):
    solucion = colorearGraficaConNColores(archivo)
    costo = zakharov(solucion.colores_asignados)
    while True:
        vecindad = generarVecindad(solucion)
        mejor_vecino = min(vecindad, key=lambda x: zakharov(x.colores_asignados))
        if zakharov(mejor_vecino.colores_asignados) < costo:
            solucion = mejor_vecino
            costo = zakharov(solucion.colores_asignados)
        else:
            break
    return solucion
```

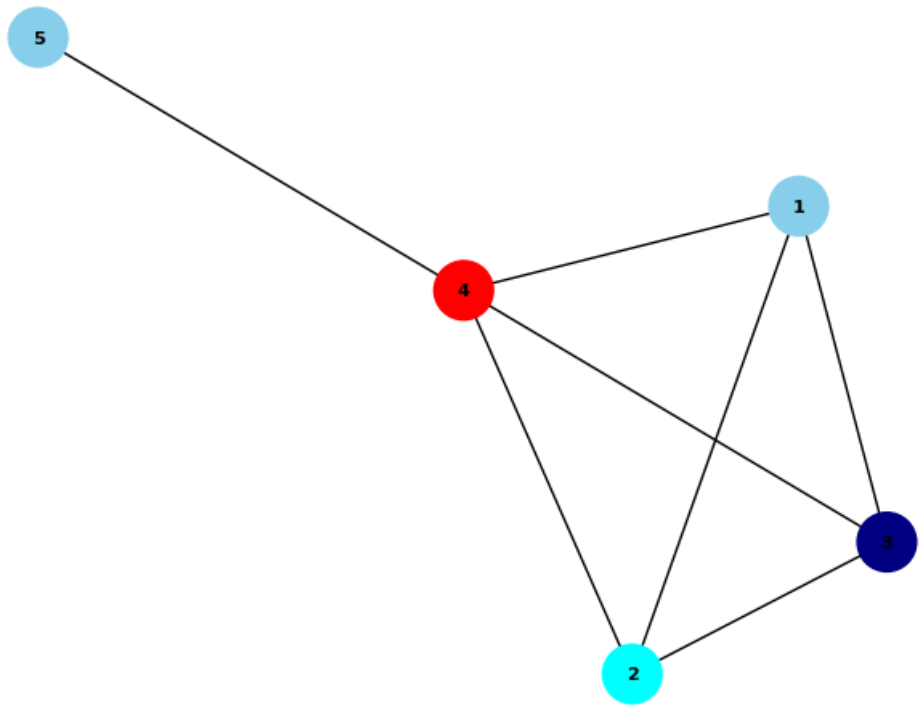
f) Propón y codifica algunos ejemplares de prueba y prueba tu implementación.
En el reporte deberás agregar:

- Nombre del ejemplar de prueba (archivo)
- Representación gráfica del ejemplar
- Ejemplo de solución aleatoria generada
- Evaluación de la solución aleatoria
- Ejemplo de la aplicación del operador (o función) de vecindad

✧ Agrega al menos un ejemplar que sea suficientemente corto para poder representarlo en una gráfica pequeña, con al menos 5 vértices y 6 aristas.

```
c Archivo: graficaPapalote.col
p edge 5 7
e 4 5
e 1 3
e 2 3
e 2 1
e 2 4
e 3 4
e 4 1
```

Al ejecutar lo anterior tenemos la siguiente gráfica:

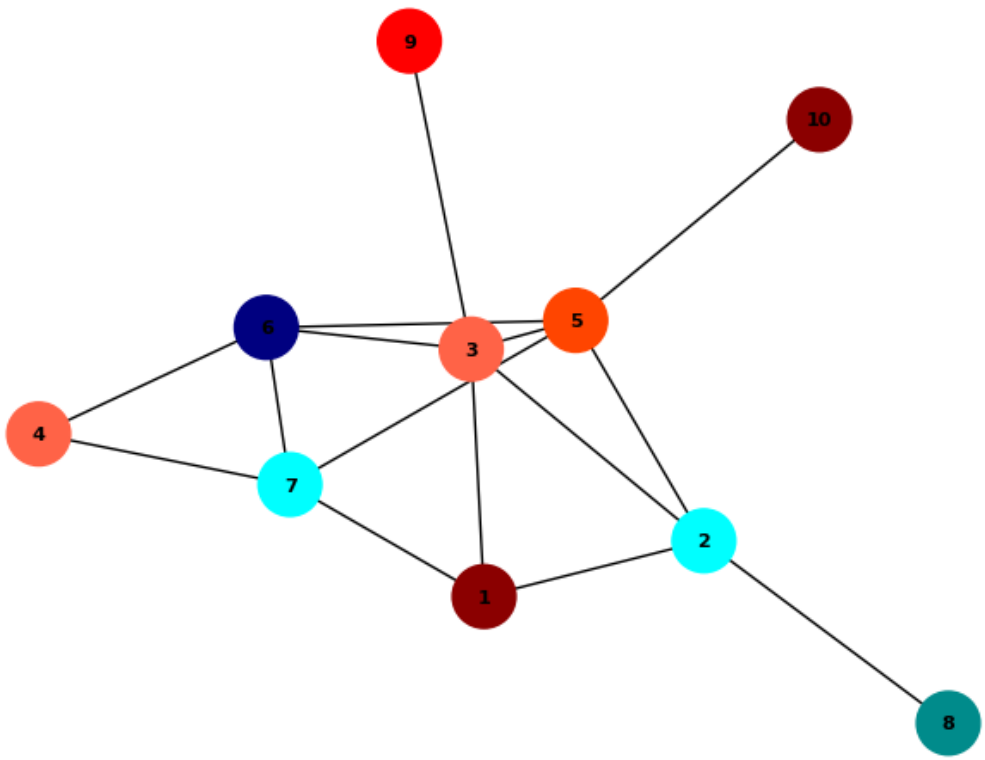


Nota: Los vértices 3 y 5 son de diferentes colores, en el notebook se aprecia mejor eso.

- Agrega al menos un ejemplar, suficientemente grande (al menos 10 vértices y 15 aristas), que solo esté representado en archivos. Deberás adjuntar el archivo que codifica el ejemplar, y un archivo que codifique la solución.

```
c Archivo: graficaTres.col
p edge 10 15
e 1 2
e 1 3
e 2 3
e 2 5
e 3 5
e 3 6
e 4 6
e 4 7
e 5 6
e 5 7
e 6 7
e 7 1
e 8 2
e 9 3
e 10 5
```

Al ejecutar lo anterior tenemos la siguiente gráfica:



Para poder ejecutar todo lo relacionado en código, usar la ultima celda del notebook y brindar la siguiente estructura:

```
archivo = 'nombreArchivo.col'
n_vertices, n_aristas, vertices, aristas = leer_ArchivoCol(archivo)

print("Número de vértices:", n_vertices)
print("Número de aristas:", n_aristas)
print("Vértices:", vertices)
print("Aristas:", aristas)

print("Solucion Aleatoria:")
solucion = colorearGraficaConNColores(archivo)
dibujar_Grafica_coloreado(solucion, vertices, aristas)

print("Solucion por escalada:")
solucionEscalada = busquedaEscalada(archivo)
dibujarsolucionEscalada(solucionEscalada, vertices, aristas)
```

En donde solo se modifica el parámetro:

```
archivo = 'nombreArchivo.col'
```

Con el nombre del archivo del que queramos ver su coloración.

Lo mismo aplica para las funciones:

```
codifica_aux(decimal,nBit)

decodifica_aux(binary)

codifica(x,nBit,a,b)

decodifica(x_cod,nBit,a,b)

codifica_v(x,nBit,a,b)

decodifica_v(x_cod,nBit,a,b)
```

Ejercicio 3. Preguntas de repaso

1 Menciona algún ejemplo de representación de soluciones con codificación indirecta.

2 Para cada una de los siguientes tipos de representaciones, proponer un problema de optimización combinatoria e ilustrar la representación de soluciones con un ejemplar concreto; deben ser problemas diferentes a los vistos en clase.

a) Codificación Binaria
b) Vector de valores discretos
c) Permutaciones

En cada caso se debe indicar claramente:

- El espacio de búsqueda
- La función objetivo
- Tamaño del espacio de búsqueda
- Ejemplar concreto del problema
- Ejemplo de una solución, codificada con la representación correspondiente.
- ¿Qué tipo de mapeo induce la representación?

a) **Codificación Binaria:** Un problema que aplica perfectamente para codificación binaria es el problema de la mochila, nuevamente planteamos el problema al igual que en la tarea pasada. El planteamiento se basa en que queremos llevar en una mochila la mayor cantidad de objetos más importantes que podamos, basándonos en la capacidad máxima de la mochila la cual denotaremos como W.
Debemos tomar en cuenta ciertos factores para decidir si llevar el objeto o no, como el peso o importancia del objeto. A los pesos los denotaremos como w_i y v_i respectivamente, para $i = 1, \ldots, n$, con n como la cantidad de objetos que consideramos llevar en la mochila. El problema lo podemos ver como un problema de condificación binaria donde x_i representa si llevamos o no el objeto i, con el valor de 1, si llevamos el objeto y 0, si no lo llevamos. Por lo tanto, se convierte en un problema con la siguiente función objetivo:

$$max \sum_{i=1}^{n} x_i v_i$$

Con la restricción:

$$W(x) = \sum_{i=1}^{n} x_i w_i \leq W$$

Por lo tanto, nuestro espacio de búsqueda se convierte en todas las soluciones que x que cumplen la restricción dada, con un tamaño de búsqueda de 2^n, como mencionamos en los ejercicios anteriores.Para visualizar mejor el problema es conveniente dar un ejemplo con los siguientes pesos.

Objeto	1	2	3
Peso	2	3	2
Importancia	1	2	3

Por lo tanto, que queremos maximizar

$$max\ x_1 + 2x_2 + 3x_3$$

s.a

$$2x_1 + 3x_2 + 2x_3 \leq 4$$

En esta ocasión utilizaremos otra metodología, utilizaremos el algoritmo heurístico del coeficiente de rendimiento, donde tenemos primero que ordenar la lista de objetos de forma descendente de acuerdo a la siguiente valor:

$$r_i = \frac{v_i}{w_i}$$

Obtenemos como resultado la siguiente tabla:

x	v_i	w_i	r_i
1	1	2	0.5
2	2	3	0.75
3	3	2	1.75

Ordenamos por v_i:

x	v_i	w_i	r_i
3	3	2	1.75
2	2	3	0.75
1	1	2	0.5

Vamos eligiendo en orden los objetos para llenar la mochila, en este caso serían los objetos 2 y 3, con un peso de 5, descartamos el objeto 2 porque excede el peso máximo que establece la restricción. Pasamos a tomar el objeto 1 y esta vez sí respetamos el peso máximo, por lo tanto la solución óptima resulta ser $x = (1, 0, 1)$. Sin embargo, con este método tampoco se garantiza llegar a una solución óptima. Induce un mapeo binario, del objeto numerado al 0 y 1.

b) **Vector de valores discretos:** Un tipo de un problema que tiene como solución un vector de valores discretos es un problema de programación lineal. La programación es un método matemático que es utilizado para determinar cómo maximizar o minimizar un proceso específico cuando existen una o más restricciones en el resultado del proceso. Dado $x = (x_1, x_2, \ldots, x_n)$ podemos tener como función objetivo a maximizar o minimizar una combinación lineal de x, sujeta a alguna otra combinación de x.

$$\begin{aligned} max, min\ & a_1x_1 + \cdots + a_nx_n \\ s.a\ & b_1x_1 + \cdots + b_nx_n \leq k \\ con\ & a_n, b_n, k \in \mathbb{R} \end{aligned}$$

El espacio de búsqueda está indicado por las restricciones de nuestro problema, pero en general busca valores reales. Es decir, el tamaño del espacio de soluciones es tan grande como los números reales, sin embargo, aún se encuentra restringido por las ecuaciones que se nos son dadas.
Consideremos el siguiente problema:

$$\begin{aligned} max\ z &= 3x_1 + 4x_2 \\ s.a\ 3x_1 &+ 2x_2 \leq 12 \\ -2x_1 &+ x_2 \leq 4 \\ x_1, x_2 &\geq 0 \end{aligned}$$

Para resolver este tipo de problemas un método muy utilizado es el método simplex. Realizamos la primera iteración

z	-3	-4	0	0	0
x_3	3	2	1	0	12
x_4	-2	1	0	1	4

Agarramos el valor más pequeño o más negativo de los coeficientes de costos reducidos, debido a que estamos maximizando. En este caso es x_1 y lo metemos a la base.

z	-11	0	0	4	16
0 x_3	7	0	1	-2	4
4 x_2	-2	1	0	1	4

z	0	0	1.57	0.85	22.28
3 x_1	1	0	0.14	-0.28	0.57
4 x_2	0	1	0.28	0.42	5.14

Ahora podemos ver como los coeficientes de costos reducidos asociados a las variables no básicas son mayores a cero. Por lo tanto, por el criterio de optimalidad tenemos que la solución óptima es

$$x_1 = \frac{4}{7} \approx 0.57$$
$$x_2 = \frac{36}{7} \approx 5.14$$
$$z = \frac{156}{7} \approx 22.28$$

Por lo tanto, nuestra solución puede ser representada por el vector $x = (\frac{4}{7}, \frac{36}{7})$. Induciendo un mapeo de cada una de las variables buscadas, a los reales, dependiendo de la restricción dada.

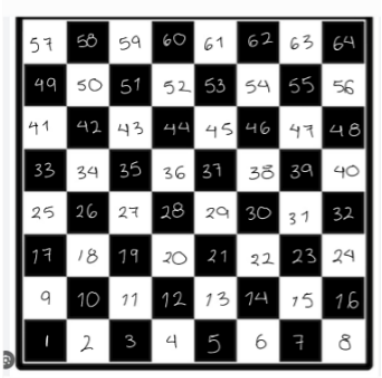
c) **Permutaciones:** Un problema que aplica para una representación de soluciones por medio de permutaciones, es el problema del caballo en el tablero de ajedrez, el cual nos pide que en un caballo en un tablero de ajedrez de n x n casillas pase por todas las casillas una sola vez, desde cualquier casilla en la que se encuentre.
Existen múltiples soluciones para este problema ya que depende del lugar en el que comience su recorrido el caballo para dar el orden en el que va a avanzar, además de que su espacio de búsqueda son todas las combinaciones de recorridos que puede dar, con un tamaño de $n!$ con n como el número de casillas del tablero. Un ejemplo concreto del problema y tomando en cuenta un tablero con 64 casillas numeradas.

8	a8	b8	c8	d8	e8	f8	g8	h8
7	a7	b7	c7	d7	e7	f7	g7	h7
6	a6	b6	c6	d6	e6	f6	g6	h6
5	a5	b5	c5	d5	e5	f5	g5	h5
4	a4	b4	c4	d4	e4	f4	g4	h4
3	a3	b3	c3	d3	e3	f3	g3	h3
2	a2	b2	c2	d2	e2	f2	g2	h2
1	a1	b1	c1	d1	e1	f1	g1	h1
	a	b	c	d	e	f	g	h

El matemático francés Edouard Lucas propuso una solución al problema, iniciando en la casilla a1, dando el siguiente recorrido como la solución.

$$(a1, c2, e1, g2, h4, g6, h8, f7, d8, b7, c5, a6, b4, a2, c1, e2, g1, h3, g5$$
$$h7, f8, e6, f4, d3, e5, f3, d4, f5, e3, d5, c3, a4, b2, d1, f2, h1, g3, h5, g7$$
$$e8, d6, c8, e7, g8, h6, g4, h2, f1, d2, b3, a5, c4, d6, e4, f6)$$

Ahora, si le asignamos un numero natural a cada casilla, tenemos un mapeo del conjunto de los naturales desde uno hasta 56 hacia el mismo conjunto pero en un orden en particular.Proponemos un orden como el siguiente.



Por lo tanto, tendremos la siguiente solución dada

$$x = (1, 11, 5, 15, 32, 47, 64, 54, 60, 50, 35, 41, 26, 9, 3, 13, 7, 24, 39$$
$$56, 62, 45, 30, 20, 37, 22, 28, 38, 21, 36, 19, 25, 10, 4, 14, 8, 23, 40, 55$$
$$61, 44, 59, 53, 63, 48, 31, 16, 6, 12, 18, 33, 27, 44, 29, 46)$$

Referencias

- https://repositorio.unan.edu.ni
- https:runestone.academy/ns/books/published/pytho ned/Graphs/Construccion-DelGrafoDeLaGiraDelCaballo.html
- Notas de clase de Investigación de operaciones con la Dra. Claudia Orquidea.
- https://networkx.org/documentation/stable/reference/index.html
- https://webs.ucm.es/BUCM/tesis/mat/ucm-t24770.pdf