UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Facultad de Ciencias. Cómputo Evolutivo

Tarea 2

Reporte de tarea

Autores:

Rosas Marín Jesús Martín - **318291015**

González Arceo Carlos Eduardo **318286488**

Profesores:

Oscar Hernández Constantino

Malinali Gónzalez Lara

28 de Febrero 2024

1. Representación de Soluciones

1.1. Representación binaria para números reales

a. Describe e implementa el algoritmo de representación binaria para mapear (codificar y decodificar) números naturales. Menciona cuántos números son representables con m bits.

El algoritmo de representación binaria para mapear números naturales consiste en convertir un número natural a su representación binaria y viceversa.

Notemos que para convertir un número natural a su representación binaria, se divide el número entre 2 y se guarda el residuo de la división. El residuo se guarda en un arreglo y el cociente se divide entre 2. Este proceso se repite hasta que el cociente sea 0. Por otro lado, para convertir un número binario a su representación decimal, se recorre el arreglo de derecha a izquierda y se suma 2^i si el bit es 1.

Observemos la implementación en python:

```python
# Función que recibe un número natural y lo convierte a su representación binaria
def binario_a_decimal(arreglo):
    """
    Convierte un número binario a su representación decimal.
    Parameters:
    arreglo -- lista de 0s y 1s que representa el número binario
    Returns:
    decimal -- número decimal
    """
    # Inicializamos la variable que guardará el número decimal
    decimal = 0
    # Recorremos el arreglo de derecha a izquierda
    for i in range(len(arreglo)):
        # Si el bit es 1, sumamos 2^i al número decimal
        if arreglo[len(arreglo) - i - 1] == 1:
            decimal += 2**i
    return decimal
```

```python
# Función que recibe un número natural y lo convierte a su representación binaria
def decimal_a_binario(n, longitud):
    """
    Convierte un número decimal a su representación binaria.
    Parameters:
    n -- número decimal
    longitud -- número de bits usados para representar el número binario
    Returns:
    binario -- lista de 0s y 1s que representa el número binario
    """
    # Inicializamos el arreglo que guardará el número binario
    binario = np.zeros(longitud, dtype=int)
    # Recorremos el arreglo de derecha a izquierda
    i = longitud - 1
    while n > 0:
        # Si el residuo de dividir n entre 2 es 1, el bit es 1
        if n % 2 == 1:
            binario[i] = 1
        # Dividimos n entre 2
        n = n // 2
        # Decrementamos i
        i -= 1
    return binario
```

Por ultimo podemos notar que los numeros representables con m bits son 2^m por que cada bit puede tomar dos valores, 0 o 1.

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?

La representación binaria se generaliza para mapear números reales por medio de la codificación de un número real en su representación binaria. Para ello, requerimos un número de bits que represente la precisión con la que se desea codificar el número real. Además, se requieren los límites inferior y superior del intervalo en el que se desea codificar el número real.

Notemos que la máxima precisión está determinada por el número de bits usados para representar el número real, es decir a mayor número de bits, mayor precisión en la representación del número real.

¿Qué relación hay entre n, el número de bits y la precisión de la representación?

La relación entre n, el número de bits y la precisión de la representación es directa ya que como mencionamos anteriormente a mayor número de bits, mayor precisión en la representación del número dado. Por otro lado, a menor número de bits, menor precisión en la representación del número.

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).

```python
def codificar(x, num_bits, a, b):
    """
    Codifica un número real en su representación binaria.
    Parameters:
    x -- número real a codificar
    num_bits -- número de bits usados para representar el número real
    a -- límite inferior del intervalo
    b -- límite superior del intervalo
    Returns:
    binario -- lista de 0s y 1s que representa el número real

    """

    precision = (b - a) / (2**num_bits)

    # asegura que x esté en el intervalo [a, b]
    x = max(a, min(b, x))

    # calcula el indice del intervalo al que pertenece x
    indice = int((x - a) / precision)

    # codifica el indice en binario
    # Usamos la funcion decimal_a_binario implementada anteriormente
    binario = decimal_a_binario(indice, num_bits)

    return binario
```

Notemos que nuestra función recibe el numero a codificar, el numero de bits y nuestros limites a y b. Ahora, usamos el numero de bits para sacar la precisión. Tambien usamos el numero en cuestión (x) para sacar nuestro indice, estos dos valores reultantes son importantes para poder pasarselos como argumentos a nuestra función decimal_a_binario que implementamos en el inciso a para poder hacer la codificación.

d. Implementa un algoritmo para decodificar los vectores de bits como un número real.

```python
def decodificar(x_cod, num_bits, a, b):
    """
    Decodifica un número real a partir de su representación binaria.
    Parameters:
    x_cod -- lista de 0s y 1s que representa el número real
    num_bits -- número de bits usados para representar el número real
    a -- límite inferior del intervalo
    b -- límite superior del intervalo
    Returns:
    x -- número real decodificado

    """
    # calcula la precision
    precision = (b - a) / (2**num_bits)

    # decodifica el indice del intervalo al que pertenece x
    indice = binario_a_decimal(x_cod)

    # calcula x a partir del indice
    x = a + indice * precision
    return x
```

Notemos que nuestra función recibe el numero codificado, el numero de bits y nuestros limites a y b correspondientes. De la misma manera que hicimos en el punto anterior usamos el numero de bits para sacar la precisión . Despues usamos nuestro numero x codificado en cuestion (x_cod) para sacar el indice, esto lo hacemos usando nuestra función binario_a_decimal implementada en el inciso a. Por ultimo usando estos dos valores calculados (precision e indice) podemos calcular el valor de x decifrado sumandole nuestro limite a a la multiplicación de los dos valores calculados.

e. Implementa las funciones necesarias para codificar y decodificar vectores de números reales

```python
def codificar_vector(x, num_bits, a, b):
    """
    Codifica un vector de números reales en su representación binaria.
    Parameters:
    x -- vector de números reales a codificar
    num_bits -- número de bits usados para representar el número real
    a -- límite inferior del intervalo
    b -- límite superior del intervalo
    Returns:
    binario -- lista de listas de 0s y 1s que representa el vector de números reales

    """
    # Usamos la funcion codificar implementada anteriormente
    binario = [codificar(i, num_bits, a, b) for i in x]
    return binario

def decodificar_vector(x_cod, num_bits, a, b):
    """
    Decodifica un vector de números reales a partir de su representación binaria.
    Parameters:
    x_cod -- lista de listas de 0s y 1s que representa el vector de números reales
    num_bits -- número de bits usados para representar el número real
    a -- límite inferior del intervalo
    b -- límite superior del intervalo
    Returns:
    x -- vector de números reales decodificado

    """
    # Usamos la funcion decodificar implementada anteriormente
    x = [decodificar(i, num_bits, a, b) for i in x_cod]
    return x
```

De la misma manera que hicimos en los puntos anteriores nuestras funciones piden x (en este caso es un vector de varios numeros a codificar o un vector de vectores a decodificar), ademas del numero de bits y nuestros limites a y b. Lo que hicimos en estas funciones es en una lista por comprensión recorrer los elementod de nuestro vector o vector de vectores e irlo pasando como parametro a su función correspondiente que puede ser codificar o decodificar. En el caso de codificar_vector nos devuelve un vector de vectores y en el caso de decodificar_vectores nos devuelve un vector con los numeros decifrados.

Para mejor visualización, revisión e interacción del script puede verse adjunto con este PDF.

1.2. Búsqueda por escalada

1. 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.

```python
def leer_instancia(nombre_archivo):
    """
    Lee un archivo con información de ejemplares (instancias) del problema de coloración.
    Parameters:
    nombre_archivo -- nombre del archivo a leer
    Returns:
    nVertices -- número de vértices del grafo
    aristas -- lista de aristas del grafo
    """

    # Inicializamos la lista de aristas
    aristas = []
    # Abrimos el archivo
    with open(nombre_archivo, 'r') as archivo:
        # Leemos el archivo línea por línea
        for linea in archivo:
            # Ignoramos las líneas que empiezan con c
            if linea[0] == 'c':
                continue
            # Si encontramos la línea que empieza con p, leemos el número de vértices
            # y aristas
            if linea[0] == 'p':
                _, _, nVertices, nAristas = linea.split() #
                nVertices = int(nVertices)
                nAristas = int(nAristas)
            # Si encontramos la línea que empieza con e, leemos las aristas
            if linea[0] == 'e':
                _, x, y = linea.split()
                x = int(x)
                y = int(y)
                aristas.append((x, y))
    return nVertices, aristas
```

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)

El esquema de representación de soluciones propuesto es un Vector de Valores Discretos donde cada posición corresponde a un vértice, y el valor en cada posición indica el color asignado al vértice. Por ejemplo, si tenemos un grafo con 5 vértices y asignamos los colores 1, 2, 3 a los vértices, una solución podría ser representada por el vector [1, 2, 1, 3, 2].

Justificación:

- Tamaño del espacio de búsqueda: El espacio de búsqueda es el conjunto de todas las posibles asignaciones de colores a los vértices del grafo. El número de colores posibles es finito, por lo que el espacio de búsqueda es finito. El tamaño del espacio de búsqueda es el número de colores posibles elevado a la cantidad de vértices del grafo. es decir n^m, donde n es el número de colores posibles y m es el número de vértices del grafo.

- Directa o indirecta: La representación es directa, ya que cada solución se representa directamente como un vector de números enteros.
- Lineal o no lineal: La representación es lineal, ya que cada solución se representa como un vector de números enteros.
- Tipo de mapeo: El mapeo es uno a uno, ya que cada solución se representa de forma única como un vector de números enteros.
- Factibilidad de soluciones: Una solución es factible si y solo si dos vértices adyacentes no tienen el mismo color.
- Representación completa: La representación es completa, ya que cada solución se representa completamente como un vector de números enteros.
- Características: La representación propuesta es sencilla y fácil de implementar. Además, el espacio de búsqueda es finito, por lo que es posible explorar todas las soluciones posibles.

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

La solución implementada evalúa las soluciones dependiendo de si son factibles o no. Esto lo hace iterando en el arreglo de la solución dada y determina que si hay dos vértices contiguos con el mismo color, entonces no es factible.

```python
def evaluar_solucion(solucion, nVertices, aristas):
    """
    Evalúa una solución al problema de coloración.
    Parameters:
    solucion -- vector de números enteros que representa la solución
    nVertices -- número de vértices del grafo
    aristas -- lista de aristas del grafo
    Returns:
    valido -- True si la solución es factible, False en otro caso
    """
    # Inicializamos la lista de colores
    colores = [0] * nVertices
    # Recorremos las aristas
    for x, y in aristas:
        # Si dos vértices adyacentes tienen el mismo color, la solución no es factible
        if solucion[x - 1] == solucion[y - 1]:
            return False
    return True
```

d) Describe e implementa un generador de soluciones aleatorias.

El generador de soluciones aleatorias lo hace creando un arreglo de longitud igual al número de vértices y lo rellena con números aleatorios entre 1 y el número máximo de colores.

```python
def generar_solucion_aleatoria(nVertices, num_colores):
    """
    Genera una solución aleatoria al problema de coloración.
    Parameters:
    nVertices -- número de vértices del grafo
    num_colores -- número de colores posibles
    Returns:
    solucion -- vector de números enteros que representa la solución
    """
    # Generamos un vector de números enteros aleatorios entre 1 y num_colores
    solucion = np.random.randint(1, num_colores + 1, nVertices)
    return solucion

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

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

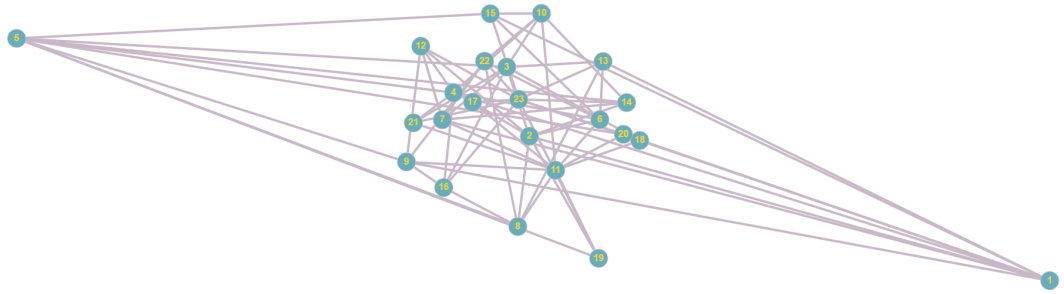
La función vecindad genera la vecindad iterando sobre la solución y sobre el numero de colores, generando uno copia de la solución y cambiando los colores.

```python
def vecindad(solucion):
    """
    Genera la vecindad de una solución al problema de coloración.
    Parameters:
    solucion -- vector de números enteros que representa la solución
    Returns:
    vecindad -- lista de soluciones vecinas
    """
    # Inicializamos la lista de soluciones vecinas
    vecindad = []
    # Recorremos los vértices de la solución
    for i in range(len(solucion)):
        # Generamos una copia de la solución
        vecina = solucion.copy()
        # Recorremos los colores posibles
        for color in range(1, max(solucion) + 1):
            # Si el color es diferente al color actual, cambiamos el color del vértice
            if color != solucion[i]:
                vecina[i] = color
                vecindad.append(vecina)
    return vecindad
```

f) Propón y codifica algunos ejemplares de prueba y prueba tu implementación.

1) **Ejemplar 1**

- Nombre del ejemplar de prueba: prueba1.co
- Representación gráfica del ejemplar



- Ejemplo de solución aleatoria generada:

  [1 1 1 1 1 2 2 1 2 1 1 2 1 1 1 3 2 2 3 1 2 3 1]

- Evaluación de la solución aleatoria: False

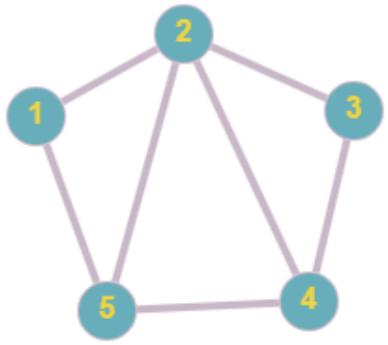
- Ejemplo de la aplicación del operador (o función) de vecindad:

  [array([3, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([3, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 3, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 3, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 3, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 3, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 3, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 3, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 3, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 3, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 3, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 3, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 3, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 3, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 3, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 3, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 3, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 3, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 3, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 3, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 3, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 3, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 3, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 3, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 3, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 3, 1, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 3, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 3, 1, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 3, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 3, 3, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 2, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1,

2, 2, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 3, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 3, 2, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 3, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 3, 3, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 2, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 2, 1, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 3, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 3, 2, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 3, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 3, 3, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 2, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 2, 1]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 3]), array([1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1, 3, 2, 2, 3, 1, 2, 3, 3])]

2) **Ejemplar 2**

- Nombre del ejemplar de prueba: prueba2.co
- Representación gráfica del ejemplar:



- Ejemplo de solución aleatoria generada:

  [2 1 1 1 3]

- Evaluación de la solución aleatoria: False
- Ejemplo de la aplicación del operador (o función) de vecindad:

  [array([3, 1, 1, 1, 3]), array([3, 1, 1, 1, 3]), array([2, 3, 1, 1, 3]), array([2, 3, 1, 1, 3]), array([2, 1, 3, 1, 3]), array([2, 1, 3, 1, 3]), array([2, 1, 1, 3, 3]), array([2, 1, 1, 3, 3]), array([2, 1, 1, 1, 2]), array([2, 1, 1, 1, 2])]

1.3. Preguntas de repaso

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

   Supongamos que estamos tratando de optimizar un conjunto de parámetros para un problema dado. En lugar de codificar directamente los valores de los parámetros, podríamos codificar un conjunto de reglas que dicten cómo se deben ajustar esos parámetros. Estas reglas podrían ser representadas como cadenas de bits o estructuras más complejas.

   Por ejemplo, en un problema de optimización de un vehículo autónomo, en lugar de codificar directamente los valores de velocidad, dirección y otros parámetros, podríamos representar una solución como un conjunto de reglas que indican cómo el vehículo debe ajustar sus parámetros en función de ciertas condiciones del entorno. Estas reglas podrían evolucionar y adaptarse a lo largo del tiempo a través de algoritmos genéticos, produciendo soluciones más efectivas para el problema en cuestión.

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

   Problema: En la gestión de proyectos, se enfrenta el desafío de asignar recursos a diferentes tareas a lo largo del tiempo de manera eficiente. Cada tarea puede o no ser realizada en un determinado periodo de tiempo, y el objetivo es maximizar la eficiencia del proyecto.

   Ejemplo de Solución:

   - Tareas: A, B, C, D
   - Periodos de tiempo: 1, 2, 3, 4
   - Posible solución (codificación binaria): [1010, 0111, 1001, 0100]

   En esta solución, cada tarea está representada por una cadena binaria de longitud igual al número de periodos de tiempo. Un "1.$^{\text{en}}$ una posición específica indica que la tarea se realiza en ese periodo, mientras que un "0ïndica que no se realiza.

   b) Vector de valores discretos

   Problema: Imagina que tienes un conjunto de trabajadores y un conjunto de tareas. Cada trabajador tiene una habilidad específica y cada tarea requiere ciertas habilidades. El objetivo es asignar a cada tarea un trabajador de manera que se maximice la eficiencia total. Cada trabajador puede ser representado por un valor discreto que indica su habilidad.

Ejemplo de Solución:

- Trabajadores: A (Habilidad: 2), B (Habilidad: 3), C (Habilidad: 1)
- Tareas: X (Requiere habilidad: 1), Y (Requiere habilidad: 3), Z (Requiere habilidad: 2)

Una solución con vector de valores discretos podría ser [A, B, C], donde:

- Tarea X es asignada a A (Habilidad: 2)
- Tarea Y es asignada a B (Habilidad: 3)
- Tarea Z es asignada a C (Habilidad: 1)

c) Permutaciones

Problema: En el campo de la bioinformática, se enfrenta el problema de secuenciar fragmentos de ADN. Se tienen varios fragmentos de ADN, y el objetivo es determinar el orden correcto de estos fragmentos para reconstruir la secuencia original del ADN.

Ejemplo de Solución: Fragmentos de ADN: .ATG", "GCA", "TGC", ÇAA"Posible solución (permutación): [.ATG", ÇAA", "GCA", "TGC"]

En esta solución, se propone que la secuencia original del ADN es .ATGCAATGC". La permutación define el orden en el que se ensamblan los fragmentos para formar la secuencia completa.

1.4. Conclusiones:

La forma en la que se representa la solución de un problema debe de ser seleccionada de forma cuidadosa para cada problema en especifico.

La codificación de la solución debe ser eficiente y compacta. En algunos casos, la codificación binaria puede ser útil para representar la presencia o ausencia de elementos, mientras que en otros problemas, una representación basada en números enteros o secuencias (como en el ejercicio 2) puede ser más apropiada.

La representación debe abordar eficazmente el espacio de búsqueda del problema. Un buen diseño de representación puede ayudar a reducir la complejidad del espacio de búsqueda y permitir la aplicación de operadores de búsqueda de manera efectiva.

La elección de una representación adecuada es un paso fundamental en el diseño de algoritmos de optimización, ya que impacta directamente en la capacidad de encontrar soluciones óptimas de manera eficiente. Es importante considerar estas cuestiones al abordar problemas específicos y adaptar

la representación en consecuencia.

2. Referencias: