Tarea 2 - Representación de soluciones

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.

   Algoritmo para codificar naturales:

   Require: n número a codificar y $nBits$ número de bits a utilizar
   Ensure: arreglo $resultado$ con número n en binario de tamaño $nBits$.
   1: $binN \leftarrow$ representación binaria de n
   2: Asegurarse de que la longitud de $binN$ sea menor o igual a $nBits$
   3: Inicializar $resultado$ como un arreglo de tamaño $nBits$
   4: **for** $i \leftarrow 0$ $nBits - 1$ **do**
   5: **if** $i <$ longitud($binN$) **then**
   6: $resultado[i] \leftarrow$ binN$[i]$
   7: **else**
   8: $resultado[i] \leftarrow 0$
   9: **end if**
   10: **end for**
   11: **return** $resultado$

   Implementación del algoritmo:

```java
public int[] codifica_aux(int n, int nBit) {
    int[] res = new int[nBit];
    if (Math.pow(a:2, nBit) > n) {
        String bin = Integer.toBinaryString(n);
        for (int i = 0; i < nBit; i++) {
            if (i < nBit - bin.length()) {
                res[nBit - 1 - i] = 0;
            } else {
                res[nBit - 1 - i] = Integer.parseInt(String.valueOf(bin.charAt(i - (nBit - bin.length()))));
            }
        }
    } else {
        throw new IllegalArgumentException(
                "El número " + n + " no se puede codificar con " + nBit + " bits.");
    }
    return res;
}
```

Figure 1: codificar natural

Algoritmo para decodificar naturales:

1: **function** DECODIFICA($nBits$)
2: $decimal \leftarrow 0$
3: **for** $i \leftarrow 0$ longitud($nBits$) $- 1$ **do**

4: $decimal \leftarrow decimal + nBits[i] \times 2^i$
5: **end for**
6: **return** *decimal*
7: **end function**

Implementación del algoritmo:

```
public double decdifica_aux(int x_cod[]) {
    double decimal = 0;
    for (int i = 0; i < x_cod.length; i++) {
        decimal += x_cod[i] * Math.pow(a:2, i);
    }
    return decimal;

}
```

Figure 2: decodificar natural

(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?**
Lo primero es asegurarse que el número a representar sea elemento del intervalo, para después a ese número n restarle a, estaermos trabajando con ese nuevo número que denotaremos $n' = n - a$.

Ahora necesitamos calcular la cantidad de números representables con el número de bits que nos proporcionaron.
Luego sacamos la diferencia entre b y a, para saber el tamaño del intervalo.
Luego dividimos el tamaño del intervalo entre la cantidad de números representables. Esa será nuestra precisión y lo siguiente que tenemos que hacer es dividir n' entre la precisión y tomar la parte entera de esa división. Ese será el número que debemos convertir a binario y tendremos nuestro número codificado.

¿Cuál es la máxima precisión?
Para mapear números reales entre mayor sea la cantidad de bits mayor será el número de números que podremos representar en un mismo intervalo, el único problema es que casi siempre lo números que metamos no seran representables, ya que en la división queda cierta parte fraccionaria que se perderá por falta de bits.
Por ejemplo en el inciso 1 se nos menciona que con n=100 tendremos una precisión de un dígito, el problema es que 100 no es potencia de 2, entonces usualmente en las particiones los números que quedan después de tomar la parte entera de

la división (la parte representable del número) tienden a tener muchos decimales y a su vez esos decimales tienden al número original entre más bits uses para su representación.

¿Qué relación hay entre n, el número de bits y la precisión de la representación?
La relación que tienen como ya lo había mencionado es que entre mayor sea el número de bits, más precisa será la representación de n.
Aunque otra relación que tienen es que hay un límite para el número de bits, ya que al calcularse la potencia de 2^n, esta función crece muy rápido y nos puede generar desbordamientos que afectan al cálculo de el número de bits representables.

(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).
La implementación fue la siguiente, donde usamos la función del inciso a) en la penúltima línea.
Implementación del algoritmo:

```java
public int[] codifica(double x, int nBits, double a, double b) {
    int[] res = new int[nBits];
    double valoresInte = b - a;
    x = x - a;
    double pres = valoresInte / (Math.pow(a:2, nBits) - 1);
    double valor = x / pres;
    res = codifica_aux((int) valor, nBits);
    return res;
}
```

Figure 3: codificar real

(d) Implementa un algoritmo para decodificar los vectores de bits como un número real.
La implementación de este algoritmo también usa su propia función auxiliar para decodificar un natural.
Implementación del algoritmo:

```java
public double decodifica(int x_cod[], double a, double b) {
    double decimal = decdifica_aux(x_cod);
    double pres = (b - a) / (Math.pow(a:2, x_cod.length) - 1);
    decimal = decimal * pres + a;
    return decimal;
}
```

Figure 4: decodificar real

(e) Implementa las funciones necesarias para codificar y decodificar vectores de números reales.
Este inciso fue sencillo teniendo los métodos para codificar y decodificar reales, ya que solo fue llamar dichos métodos cierta cantidad de veces para cada uno de los reales o binarios que representan un real respectivamente.
Quizá lo que fue más tedioso fue ir sacando los números en binario del arreglo entero antes de decodificarlos.

```java
public int[] codifica(double x[], int nBits, double a, double b) {
    int res[] = new int[x.length * nBits];
    for (int i = 0; i < x.length; i++) {
        int[] aux = codifica(x[i], nBits, a, b);
        for (int j = 0; j < nBits; j++) {
            res[i * nBits + j] = aux[j];
        }
    }
    return res;
}
```

Figure 5: codificar vector de reales

```java
public double[] decodifica(int x_cod[], int nBits, double a, double b) {
    double[] res = new double[x_cod.length / nBits];
    for (int i = 0; i < res.length; i++) {
        int[] aux = new int[nBits];
        for (int j = 0; j < nBits; j++) {
            aux[j] = x_cod[i * nBits + j];
        }
        res[i] = decodifica(aux, a, b);
    }
    return res;
}
```

Figure 6: decodificar vector de reales

Ejemplo de ejecución:
Usamos los siguientes parámetros para probar la codificación.

```
int nBits = 25;
int a = 0;
int b = 1;
double[] x = { 0.5, 0.75, 0.25, 0.10 };
```

La forma en la que lo probamos fue codificando el vector de números reales x y luego decodificándolo.

```
int[] xCodificado = f.codifica(x, nBits, a, b);
double[] xDecod = f.decodifica(xCodificado, nBits, a, b);
```

Ya lo último que hicimos fue imprimir los valores del vector de reales decodificado.

```
PS E:\Documentos\4t sem\Cómputo Evolutivo\CE_Tarea02> java -jar .\target\ejecuta.jar
Seleccione una opción:
1. Problema de coloración
2. Problema de codificación
2
0.49999998509883836
0.7499999925494192
0.24999997764825754
0.09999999701976767
PS E:\Documentos\4t sem\Cómputo Evolutivo\CE_Tarea02>
```

Figure 7: Ejecución de la codificación/decodificación

Como podemos darnos cuenta los números son aproximados, esto se debe a que nuestros números originales no eran representables con ese número de bits y ese intervalo, entonces se trunca hacia el valor más cerca representable hacia abajo.
Si usamos una menor cantidad de bits, entonces será menos precisa la decodificación.
Otra cosa a notar es que como 100 y 1000 no son potencia de 2, entonces poder decodificar los números que usé en dicho intervalo no es posible haciendo la partición con 2^n números (con n natural).

Ejecuciones extra:
Aunque hicimos distintas ejecuciones, no se nos pidió que se pudiera ejecutar con parámetros desde terminal, entonces si quieren hacer una ejecución extra, sería entrar el archivo ubicado en src/main/java/CE.java y cambiar el valor del número de bits, los límites del intervalo o el vector de números reales; Todos estos parámetros se encuentran entre las líneas 20-23.

2. **Búsqueda por escalada**

(a) SModelaremos el problema de la coloración haciendo uso de una matriz de adyacencia. Utilizaremos la siguiente gráfica como ejemplo:

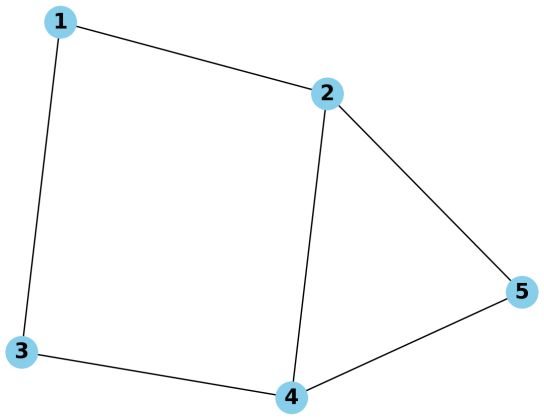


Figure 8: Ejemplo

El archivo que codifica la gráfica anterior, tiene la siguiente forma:

```
c Este es un archivo de ejemplo para una gráfica con 5 vértices y 6 aristas
p edge 5 6
e 1 2
e 1 3
e 2 4
e 2 5
```

```
e 3 4
e 4 5
```

Al leer el archivo anterior, nuestro programa crea una matriz de adyacencia utilizando el número de vértices proporcionado, cuya representación es la siguiente:

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	1
3	1	0	0	1	0
4	0	1	1	0	1
5	0	1	0	1	0

Las adyacencias de los vértices se reflejan con un 1 en la celda que representa la adyacencia de los mismos, aunque se registren dos veces cada una, la representación sigue siendo válida.

Describe e implementa un esquema de representación de soluciones.
Utilizaremos una representación clásica de solución que nos ayude a visualizar el resultado siguiendo la estructura de nuestro programa. Utilizaremos un *Vector de Valores discretos*; utilizamos esta representación de solución puesto que éste será representado por un arreglo donde cada celda representa el número de vértice y el valor que ésta almacene representa el color asignado.

i. **Tamaño del espacio de búsqueda:**
Al considerar que nuestra representación de soluciones es un vector de valores discretos, al tener n vértices, podemos representar $n!$ soluciones, sean óptimas o no.
Dentro de éste número se encuentran todas las posibles combinaciones de números dentro del arreglo, es decir, todas las posibles combinaciones de colores para cada vértice.

ii. **Directa o indirecta:**
Es una representación directa, partiendo de la idea de que las celdas son los vértices y el valor que almacena el color, se puede interpretar correctamente el resultado sin la necesidad de realizar algún cambio o tener que decodificarlo.

iii. **Lineal o no lineal**
Nuestra solución es lineal, puesto que la representamos haciendo uso de un arreglo que funciona como un vector de valores discretos, es decir, una secuencia unidimensional de números enteros que funcionan como color.

iv. **Tipo de mapeo**
Podemos interpretar el mapeo de uno a muchos, puesto que existen dos o más gráficas que pueden colorearse con la misma configuración de colores, aunque en alguna de ellas la solución no sea válida o no sea la óptima.
Un ejemplo son las gráficas con el mismo número de vértices, tendrán el mismo número de celdas y es posible almacenar en ellas el mismo valor, algunas de estas soluciones serán óptimas, algunas válidas y la mayoría no lo serán.

v. **Factibilidad de soluciones**
Todas las soluciones generadas son factibles, puesto que nuestro método para crear soluciones aleatorias tiene ciertas restricciones, una de ellas es que cada vértice adyacente debe tener un color distinto, esto provoca que siempre se genere una solución válida sacrificando el número de colores utilizados.
Sin embargo, podemos representar soluciones no factibles en el arreglo, que representa el vector de valores discretos, es por ello que aunque solo generemos soluciones factibles, es posible representar todas las soluciones, sean válidas o no.

vi. **Representación completa**
Es posible representar cada solución de una gráfica, teóricamente partiendo de una solución aleatoria y generando soluciones vecinas, es posible llegar a cualquier respuesta válida al problema, de igual manera el arreglo no supone ningún impedimento, es decir, de igual forma el vector de valores discretos puede representar en su totalidad todas las soluciones.

(b) **Describe e implementa una función de evaluación para las soluciones.**
Para realizar la función que evalúe las soluciones, podemos enfocar la evaluación de la siguiente manera:
Teniendo en cuenta que cada solución aleatoria generada es válida, un criterio para conocer la calidad de la solución es la cantidad de colores utilizados para colorear la gráfica, mientras menos colores sean utilizados, nuestra solución es mejor.

Algorithm 1 Evaluar Solución

Require: *solucion*: Arreglo de enteros que representa los colores asignados a cada vértice.
Ensure: Número total de colores únicos utilizados en la solución.

```
coloresUsados ← arreglo de booleanos de tamaño numVertices+1, inicializado en false
totalColores ← 0
for cada color en solucion do
    if coloresUsados[color] es false then
        coloresUsados[color] ← true
        totalColores ← totalColores + 1
    end if
end for
return totalColores
```

(c) **Describe e implementa un generador de soluciones aleatorias.**
Para generar las soluciones aleatorias, tomamos en cuenta de que cada solución generada sea válida, es decir, que ningún vértice adyacente tenga el mismo color. Si por el contrario, a cada vértice se le asigna un color aleatorio sin importar sus adyacencias, entonces muchas de las soluciones aleatorias no serían válidas y tendríamos que generar más soluciones para que encontremos una válida, cosa que nuestro generador aleatorio realiza en la primera ejecución.

Algorithm 2 Generar Solución Aleatoria

Require: $numVertices$: Número total de vértices en la gráfica.
Require: $matrizAdyacencia$: Matriz de adyacencia que representa la gráfica.
Ensure: Arreglo de enteros que representan una solución de colores aleatorios.

```
solucion ← arreglo de enteros de tamaño numVertices, inicializado en 0
Inicializar r como un generador de números aleatorios
for i ← 0 to numVertices − 1 do
    coloresUsados ← arreglo de booleanos de tamaño numVertices + 1, inicializado en false
    for j ← 0 to numVertices − 1 do
        if matrizAdyacencia[i][j] = 1 and solucion[j] ≠ 0 then
            coloresUsados[solucion[j]] ← true
        end if
    end for
    repeat
        color ← 1 + r.nextInt(numVertices)
    until coloresUsados[color] es false
    solucion[i] ← color
end for
return solucion
```

(d) **Describe e implementa una función u operador de vecindad, acorde al tipo de representación implementado en los incisos anteriores.**
Para implementar una función de vecindad, es decir, realizar un cambio en la solución actual (aleatoria), intentamos maximizar el uso de un color mientras éste respete las soluciones válidas. Primero contamos la frecuencia de cada color, paso siguiente tomamos un vértice aleatorio y verificamos si es posible asignarle el color que más se utiliza en la gráfica, si es posible se lo asignamos y devolvemos nuestra solución vecina; si no es posible entonces le asignamos un color aleatorio.

Algorithm 3 Generar Solución Vecina

Require: $solucionActual$: Arreglo de enteros representando la solución actual.
Require: $numVertices$: Número total de vértices de la gráfica.
Require: $matrizAdyacencia$: Matriz de adyacencia de la gráfica.
Ensure: Nueva solución vecina.

$nuevaSolucion \leftarrow solucionActual$.clone()
$vertice \leftarrow$ número aleatorio entre 0 y $numVertices - 1$
$frecuenciaColores \leftarrow$ arreglo de enteros de tamaño $numVertices + 1$, inicializado en 0
for cada $color$ en $solucionActual$ **do**
    if $color \neq 0$ **then**
        $frecuenciaColores[color] \leftarrow frecuenciaColores[color] + 1$
    end if
end for
$coloresUsados \leftarrow$ arreglo de booleanos de tamaño $numVertices+1$, inicializado en $false$
for $j \leftarrow 0$ to $numVertices - 1$ **do**
    if $matrizAdyacencia[vertice][j] = 1$ **then**
        $coloresUsados[solucionActual[j]] \leftarrow true$
    end if
end for
$colorSeleccionado \leftarrow 0$
$maxFrecuencia \leftarrow 0$
for $color \leftarrow 1$ to $numVertices$ **do**
    if no $coloresUsados[color]$ y $frecuenciaColores[color] > maxFrecuencia$ **then**
        $colorSeleccionado \leftarrow color$
        $maxFrecuencia \leftarrow frecuenciaColores[color]$
    end if
end for
if $colorSeleccionado = 0$ **then**
    repeat
        $colorSeleccionado \leftarrow 1+$ número aleatorio entre 0 y $numVertices - 1$
    until no $coloresUsados[colorSeleccionado]$
end if
$nuevaSolucion[vertice] \leftarrow colorSeleccionado$
return $nuevaSolucion$

(e) Propón y codifica algunos ejemplares de prueba y prueba tu implementación.
Prueba ejemplo1.
Probaremos nuestro programa ingresando la gráfica llamada $ejemplo1$ que se encuentra codificada en la ruta $src/graficas/$, la cual corresponde al siguiente formato:

```
c Este es un archivo de ejemplo para una gráfica con 18 vértices y 25 aristas
p edge 18 25
e 1 2
e 1 3
e 1 18
e 2 4
e 2 5
e 2 17
e 3 6
e 3 16
e 4 7
e 4 8
e 5 9
e 5 10
e 6 11
e 6 12
e 7 13
e 8 14
e 9 15
e 10 16
e 11 17
e 12 18
e 13 14
e 14 15
e 15 16
e 16 17
e 17 18
```

Cuya solución es la siguiente:

```
Primer solución aleatoria: [8, 4, 1, 11, 7, 17, 13, 14, 6, 1, 2,
6, 11, 2, 18, 5, 3, 16]
Evaluación de la primer solución aleatoria: 14
Solución vecina generada: [8, 4, 1, 11, 7, 17, 13, 14, 6, 1, 2,
6, 11, 2, 18, 5, 3, 1]
Evaluación de la solución vecina: 13
Solución más optimizada encontrada después de 7 iteraciones y 30
iteraciones sin mejora.
Vértice 1: Color 9
Vértice 2: Color 5
```

```
Vértice 3: Color 5
Vértice 4: Color 10
Vértice 5: Color 9
Vértice 6: Color 6
Vértice 7: Color 5
Vértice 8: Color 9
Vértice 9: Color 10
Vértice 10: Color 5
Vértice 11: Color 5
Vértice 12: Color 1
Vértice 13: Color 9
Vértice 14: Color 5
Vértice 15: Color 6
Vértice 16: Color 9
Vértice 17: Color 1
Vértice 18: Color 5
Total de colores utilizados: 5
Arreglo de valores discretos: [9, 5, 5, 10, 9, 6, 5, 9, 10, 5,
5, 1, 9, 5, 6, 9, 1, 5]
```

Prueba ejemplo2.
Probaremos nuestro programa ingresando la siguiente gráfica llamada $ejemplo2$ que se encuentra codificada en la ruta $src/graficas/$:

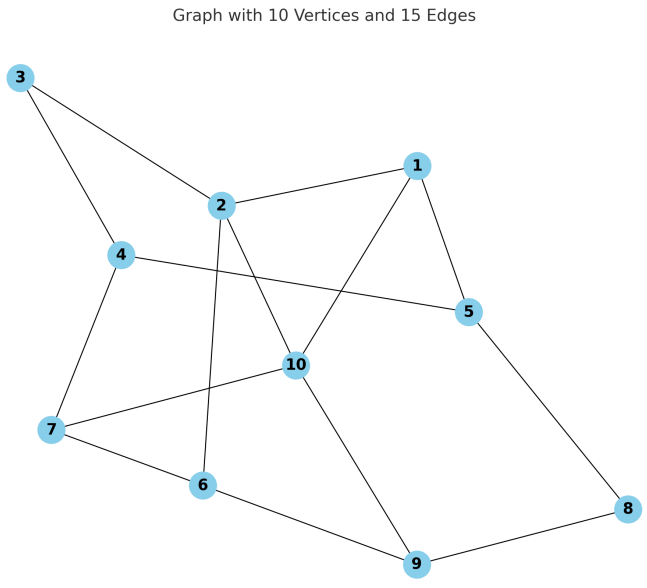


Figure 9: Representación de la gráfica

La cual corresponde al siguiente formato:

```
c Este es un archivo de ejemplo para una gráfica con 10 vértices y 15 aristas
p edge 10 15
e 1 2
e 1 5
e 1 10
e 2 3
e 2 6
e 2 10
e 3 4
e 4 5
e 4 7
e 5 8
e 6 7
e 6 9
e 7 10
e 8 9
e 9 10
```

Y genera la siguiente solución:

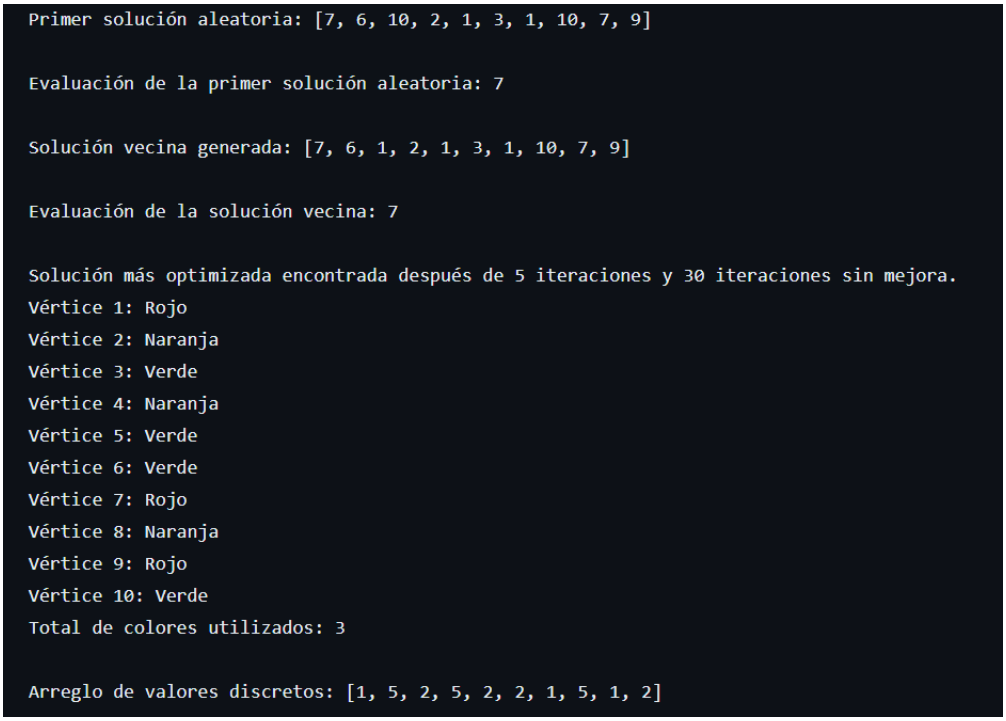


Figure 10: Resultado de la gráfica

Donde después de resolver el problema, nos devuelve lo siguiente:
Arrojamos una solución aleatoria donde a cada vértice se le asigna un color con la única restricción de que dos vértices adyacentes no tengan el mismo color.
Como vemos, la calidad de la solución aleatoria es 7, puesto que utiliza 7 colores para colorear la gráfica sin errores.
La solución vecina comienza contando la frecuencia de los colores y se asigna el color más utilizado a un vértice aleatorio, no necesariamente debe mejorar. Notamos que la calidad de la solución aleatoria es la misma que la de la solución vecina, solamente cambia el color a un vértice aunque no disminuya los demás colores.
Para solucionar el problema de coloración, repetimos el mismo proceso un total de treinta veces para ejemplares relativamente pequeños, nos referimos a las treinta iteraciones como tolerancia, puesto que son las veces que genera una solución vecina y su calidad es peor o igual a la solución actual; cuando la solución vecina es mejor que la actual, reemplazamos la actual por la mejor solución y continuamos el proceso, estas iteraciones no son contempladas en la tolerancia sino en otro contador que nos indica cuántas veces mejora. Utilizamos *primero en mejorar* para encontrar la solución más óptima o acercanos a ésta. Renunciamos a la idea de crear una función que relacione a un número con un color puesto que la cantidad de colores limitaba la cantidad de vértices de las gráficas a resolver. La cual corresponde a la siguiente coloración:

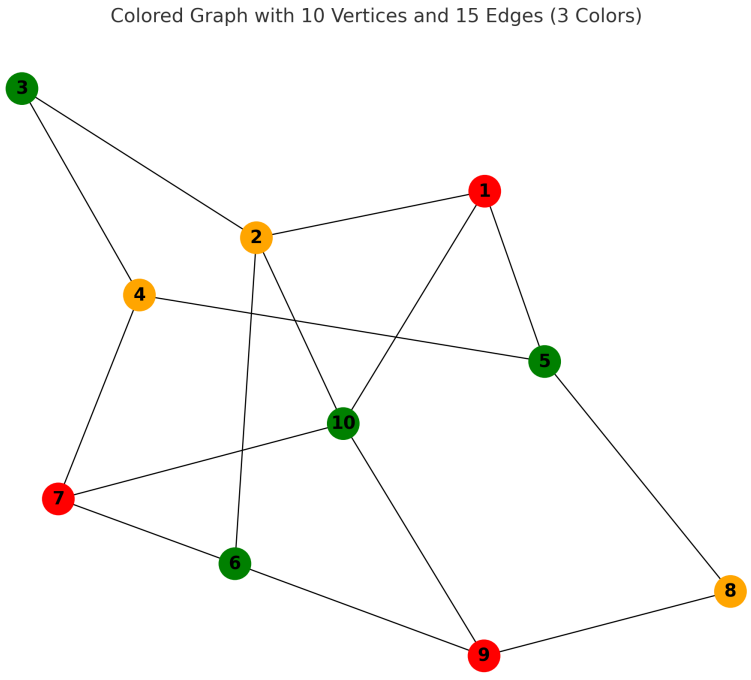


Figure 11: Representación del resultado

3. **Preguntas de repaso**

(a) **Menciona algún ejemplo de representación de soluciones con codificación indirecta.**
En el problema de la coloración, la solución que queremos es una gráfica coloreada con el menor número de colores posibles, sin embargo la representación de soluciones para facilitar la búsqueda de vecindades se hace de forma numérica.
Una de las opciones para representar las soluciones es utilizando un Vector de valores continuos, donde cada vértice tiene asignada una prioridad representada por números continuos, el vértice con más prioridad escoge un color inicial, el vértice con la prioridad posterior escoge el mismo color si es que no son adyacentes, para cada vértice se verifica el color de sus adyacencias, si ya son tomados (sus adyacentes tienen mayor prioridad) entonces toma otro color.
Esto implica que la solución obtenida a través de nuestro algoritmo no sea la solución que queríamos, ya que tendremos que decodificar los valores y ordenar los vértices dependiendo de su prioridad representada por el número continuo, para después encontrar el color que está asignado a este vértice.
Las soluciones se ven de la siguiente manera:
[0.90.20.10.60.20.8] Donde el vértice en la posición 0 tiene 0.9 de prioridad, el vértice en la posición 1 tiene 0.2 de prioridad y así sucesivamente.
Los colores están fijos y son igual al número de vértices, por lo que todos tratan de tomar el primer color para que así todos tengan el mismo color y el número de colores utilizados sea 1, solo que cuando existen adyacencias deben tomar otro color distinto y así sucesivamente.

(b) 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?

Codificación binaria
Abordaremos el problema de la mochila, en donde tenemos una mochila con un limite de peso y distintos objetos con un peso fijo. El objetivo es maximizar el peso de la mochila decidiendo qué objetos metemos a la mochila sin que haya un excedente al límite de peso.
El peso de los objetos se toma completo.[Jai]

i. **Espacio de búsqueda**
El espacio de búsqueda estará dado por el número de objetos que tenemos para meter, serán vectores binarios donde cada número en la posición i nos indica si el objeto con el índice i está en la mochila (1) o no está (0).

ii. **Función objetivo**
La función objetivo busca maximizar el peso total que llevamos en la mochila sin que exceda al límite.

iii. **Tamaño del espacio de búsqueda**
El tamaño del espacio de búsqueda es 2^n [MT90], con n el número de bits en la codificación, dado que cada bit puede tener solamente 2 estados (0 o 1), y la codificación completa representa una combinación única de estos estados. Aunque cabe destacar que pueden existir soluciones que excedan el límite de peso en la mochila y no sean factibles, aún así siguen siendo parte del espacio de búsqueda ya que eso solo lo sabremos al evaluarlo en nuestra función objetivo.

iv. **Ejemplar**: Supongamos que tenemos una mochila con límite de peso 10 y además tenemos 4 objetos (x_0, x_1, x_2, x_3) con los siguientes pesos respectivamente $3, 4, 5, 6$

v. **Solución**: Una posible solución para este ejemplar sería el vector binario $(0, 1, 0, 1)$, el cual nos indicaría que dentro de la mochila se encuentran los objetos x_1 y x_3 y la suma de sus pesos sería $4 + 6 = 10$ lo cual maximiza exactamente el límite de la mochila.
Otra posible solución sería el vector $(1, 1, 0, 0)$ el cual generaría un peso de $3 + 4 = 7$ el cual no excede el límite de la mochila.

vi. **¿Qué tipo de mapeo induce la representación?**
La representación induce un mapeo uno a uno, donde la codificación binaria se relaciona con una única forma de elegir qué objetos meter.

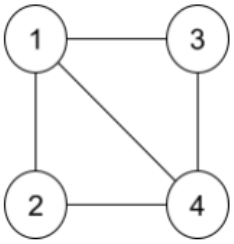
Vector de valores discretos
Abordaremos el problema de coloración de gráficas, donde nos enfocamos en colorear una gráfica minimizando el número de colores utilizados, teniendo en cuenta que dos vértices adyacentes no pueden tener el mismo color. Cada celda del vector representa un vértice y el valor almacenado en ella representa el color que tiene tal vértice.

i. **Espacio de búsqueda**
El espacio de búsqueda son todas las posibles combinaciones de colores para cada vértice

ii. **Función objetivo**
La función objetivo busca encontrar el número mínimo de colores para colorear una gráfica

iii. **Tamaño del espacio de búsqueda**
El tamaño del espacio de búsqueda es $n!$, con n el número de vértices, puesto que podemos representar todas las posibles combinaciones de los colores asignados a los vértices. [MR97]

iv. **Ejemplar**:



v. **Solución**: $[1, 2, 2, 3]$
Nos indica que el primer vértice se colorea con el color 1, el segundo y tercer vértice se colorea con el color 2, puesto que no son adyacentes, como el vértice 4 es adyacente a todos los anteriores, es necesario colorearlo con un color distinto a los ya utilizados, con el color 3.

vi. **¿Qué tipo de mapeo induce la representación?**
La representación induce un mapeo de uno a muchos, es decir, con una mismo vector de valores discretos, se puede representar la solución de dos o más gráficas, no es necesario que sea la más óptima, sino que cumpla con la condición del problema.

Permutaciones Retomaremos el problema de la mochila, en donde tenemos una mochila con un límite de peso y distintos objetos con un peso fijo. El objetivo es maximizar el peso de la mochila decidiendo qué objetos metemos a la mochila sin que haya un excedente al límite de peso. En este caso, para representar las soluciones utilizando permutaciones; ordenaremos los objetos dependiendo de sus características, en este caso, el peso.
El objeto en el primer lugar de la permutación es el que tiene mayor prioridad para entrar a la mochila, no podremos agregar más objetos a la mochila si eso implica exceder el límite de peso.

i. **Espacio de búsqueda**
Todas las posibles permutaciones de los objetos. Cada permutación representa un orden específico para considerar los objetos.

ii. **Función objetivo**
La función objetivo busca maximizar el peso total que llevamos en la mochila sin que exceda al límite.

iii. **Tamaño del espacio de búsqueda**
El tamaño del espacio de búsqueda es $n!$, con n igual a la cantidad de objetos disponibles.

iv. **Ejemplar**: Supongamos que tenemos 5 objetos con diferentes pesos y características. El problema sería determinar en qué orden considerar estos objetos para maximizar el valor total sin sobrepasar el límite de peso de la mochila. Consideramos 10 el límite de peso de la mochila.
-1. Lápices 1
-2. Cuadernos 4

-3. Lonchera 6
-4. Lapicera 3
-5. Colores 2

v. **Solución**: Si los objetos están por peso en orden ascendente, es posible maximizar el número de objetos dentro de la mochila, es decir, la solución se puede representar como sigue:
$[1, 5, 4, 2, 3]$

vi. **¿Qué tipo de mapeo induce la representación?**
La representación induce un mapeo uno a uno, cada permutación de elementos de una prioridad única.

References

[Jai] Sonoo Jaiswal. Daa: 0/1 knapsack problem - javatpoint.

[MR97] Michael Molloy and Bruce Reed. Graph coloring and applications. *Providence: American Mathematical Society*, 1997.

[MT90] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley Sons, 1990.