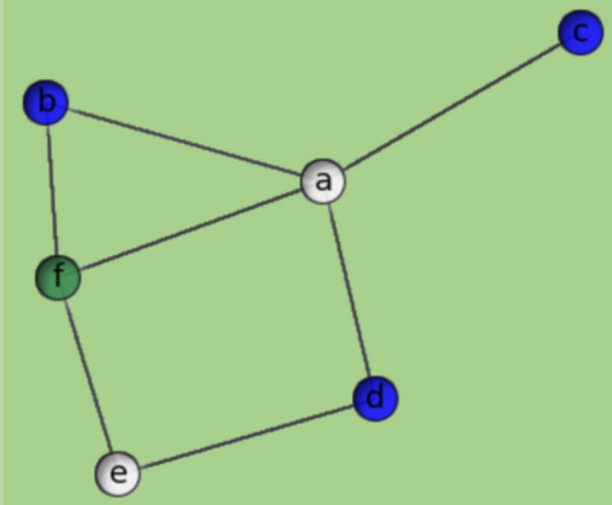
INTELIGENCIA ARTIFICIAL 2018

Proyecto final: Problema de Coloración de grafos



Ciencias de la Computación

Semestre Enero-Mayo 2018

ALAN RODRIGO LOPEZ MALDONADO

Facultad de Matemáticas

UADY

8-5-2018

INTELIGENCIA ARTIFICIAL

Alan Rodrigo López Maldonado

Reporte Proyecto final: Coloración de grafos

Algoritmos evolutivos: Algoritmo genético

Contenido: página

1. INTRODUCCIÓN

1.1. Descripción del problema

Uno de los problemas mas conocidos de Teoría de Grafos es el problema de coloración de grafos. Se trata de asignar etiquetas denominadas colores a los elementos de un grafo. Para este proyecto se implementará un algoritmo que permita colorear los vértices de un grafo. El objetivo del problema es asignar colores a los vértices de un grafo, de tal forma que los vértices adyacentes no compartan el mismo color. Cada grafo debe tener un número mínimo de colores necesarios, el cual se denomina número cromático, del grafo. La importancia del problema es que la coloración de grafos tiene múltiples aplicaciones y puede utilizarse para resolver muchos problemas de asignación. Entre sus aplicaciones se encuentran: el coloreado de mapas (vertices=regiones, aristas=fronteras), calendarización de examenes finales (vertices=clases, colores=dias, aristas=estudiante tomando ambas asignaturas), Distribución de asientos, Asignación de horarios universitarios, entre otros.

2. METODOLOGÍA

2.1. Método de implementación

El algoritmo para la coloración de grafos se implementó mediante el algoritmo genético, el cual consiste en una función recursiva que crea generaciones con posibles soluciones. Este algoritmo es considerado un algoritmo evolutivo puesto que, por cada recursión, mejora las posibles soluciones al problema dado (en este caso, un grafo) para encontrar una solución. El programa se realizó en el lenguaje de programación Python 2.7 usando las librerías: NetworkX (para visualizar el grafo) y Matplotlibt y Tkinter (para la interfaz gráfica). En cuanto a la estructura funcional del algoritmo se definieron varias

variables y funciones con el fin de obtener los datos del grafo recibido: el número y nombre de vertices, los nodos/vertices adyacentes, e información de los colores utilizados. Se requirieron utilizar listas (con n=numero de vértices, elementos) para contener las poblaciones resultantes del proceso del algoritmo genético y diccionarios para hacer la relación entre los vertices y colores. Posteriormente se programaron diversas funciones para realizar los procedimientos de: selección, torneo, cruzamiento, mutación, fitness y vialización del grafo. El proceso general del algoritmo genético es el siguiente: Inicialmente se crea una lista que contiene posibles soluciones (creadas aleatoriamente). Se seleccionan (mediante el Fitness) las mejores soluciones candidatas, se realiza cruzamiento y mutación para los elementos de la lista y finalmente se determina si algun elemento de la lista es una solución para el problema dado. Si lo es, devuelve la solución; si no, realiza otr recursión con los nuevos valores de la lista. Este es un proceso recursivo hasta que se encuentre alguna solución o bien hasta que se llegue a un número máximo de recursiones. Cada cromosoma de la generación, se llena con números entre 0 y 4 (número de colores). Cada indice de la lista corresponde a un Vértice, de tal forma que posteriormente se relaciona el indice (vértice) con el valor que tiene (color).

```
#Alan Rodrigo Lopez Maldonado

from Tkinter import *
import Grafo1
import GeneticFunctions
import GrafoSolucion


def RealizarAlgoritmoGenetico(NOMBREGRAFO):
    Grafo1.GrafoNetworkx(NOMBREGRAFO)
    Solucion= GeneticFunctions.MainFunction(NOMBREGRAFO)
    GrafoSolucion.GrafoNetworkx(NOMBREGRAFO, Solucion)
    MostrarMenuPrincipal()


def ObtenerNombresGrafos():
    lista=[]
    arch = open('ListaDeNombresDeGrafos.txt', 'r') #se abre solo para lectura
    text = file.read(arch)
    lista= text.split(",")
    arch.close()
    return lista



def MostrarMenuPrincipal():

    #ListadeGrafos=[ 'Grafo.txt' ,'Grafo.txt', 'Grafo.txt']
    ListadeGrafos=[]
    ListadeGrafos= ObtenerNombresGrafos()
    print "la lista de nombres de grafos es:"
    print ListadeGrafos
```

2.2. Función Fitness

Función Fitness recibe un cromosoma (solucion candidata) y retorna el número de conflictos, entre los vertices adyacentes, que tiene la población recibida. Cada cormosoma se define como sigue: se tiene una lista de N elementos, donde N es el número de vértices. Cada posición de la lista esta llenada con números entre 0 a NC, (NC=número de colores). Asi, los subindices de la lista (posiciones) corresponden a cada vértice, es decir, el indice 0 es el primer vértice, el indice 1, es el segundo vértice,... hasta indice (N-1) es el último vertice. Los valores de la lista corresponden al color asignado. Entonces, se utiliza un diccionario para relacionar el vertice correspondiente (indice de la lista) con el color asignado (valor de la lista). Luego otro diccionario para relacionar el valor asignado con el nombre del color. Posteriormente se recorre una lista que contiene los nodos adyacentes (aristas). Si los elementos i de la lista tienen el mismo color, se suma 1 a la variable fit (significa que los nodos adyacentes presentan conflicto). El mejor fitness es el tenga el menor número de coincidencia entre colores de vértices adyacentes. Entonces una solución al problema es aquella que su fitness (fit) sea igual a cero.

```
def Fitness(cromosoma, N,VERTICES, ADYACENCIA):
    fit=0
    ##primero se crea un diccionario para relacionar los vertices con sus colores
    i=0
    diccionario={}
    for i in range(0,N): #de cero hasta el numero de vertices
        diccionario[VERTICES[i]]= cromosoma[i]
    #luego se recorre la lista "ADYACENCIA" y se verifica si los valores
    #correspondientes del diccionario son iguales
    j=0
    for j in  range(0,len(ADYACENCIA)):
        colorvertice1=diccionario[ADYACENCIA[j][0]]
        colorvertice2=diccionario[ADYACENCIA[j][1]]
        if( colorvertice1 == colorvertice2):
            fit=fit+1
    return fit
```

2.3. Selección por Torneo

La función Selección recibe una lista “Generacion” (que contiene muchas posibles soluciones) y sirve para seleccionar a los mejores individuos de la Generación. Esto lo realiza mediante un torneo, que selecciona 3 cromosomas y luego calcula el Fitness de cada una, para decidir qué cromosoma cuenta con el menor Fitness y así decidir cuál es mejor. Se retorna la lista Generacion con los cromosmas seleccionados.

```
def Torneo(ven,N,VERTICES, ADYACENCIA):
    crom1= ven[0]
    crom2= ven[1]
    crom3= ven[2]
    ListaFitness=[ Fitness(crom1,N,VERTICES, ADYACENCIA),Fitness(crom2,N,VERTICES, ADYACENCIA), Fitness(crom3,N,VERTICES, ADYACENCIA) ]
    indice_ganador = ListaFitness.index( min(ListaFitness)) #se obtiene el indice correspondiente al cromosoma con el menor fitnes.
    crom_winner = ven[indice_ganador]   #se obtiene el cromosoma que correspnde al indice ganador
    return crom_winner


def Seleccion(listaGeneracion, M ,N,VERTICES, ADYACENCIA):
    Generacionseleccionada=[]
    contador=0
    while(contador != M):       #Se realizan M iteraciones para poder obtenr una generacion con M poblaciones
        ventana=[]
        crom1=listaGeneracion[contador%M]
        crom2=listaGeneracion[(contador+1)%M]
        crom3=listaGeneracion[(contador+2)%M]
        ventana=[crom1, crom2, crom3]
        crom_mejorFitness = Torneo(ventana,  N,VERTICES, ADYACENCIA)  #  !!SE REALIZA EL TORNEO!!  #la poblacion ganadora se agrega a l
        Generacionseleccionada.append(crom_mejorFitness)  #se va llenando una lista que contiene a las poblaciones ganadoras del torneo
        contador= contador +1

    return Generacionseleccionada
```

2.4. Cruzamiento

La función Cruzamiento recibe dos cromosomas (parent1, parent2) y realiza el Crossover (cruzamiento). Cada parent se divide a la mitad. La probabilidad de realizar el cruzamiento es de 70, por lo que si se recibe un número aleatorio menor igual a 70, se realiza el crossover, en el cual los parents intercambian tramas (el punto de corte se toma como la mitad). Posteriormente se llama a la función mutación para determinar si también se realiza la mutación a los elementos del parent. Finalmente se retornan los cromosmas hijos obtenidos. Nota: aunque no hubiera cruzamiento, los parents podrían mutar sus elementos.

```
def Cruzamiento(parent1, parent2, NC):
    punto_de_corte= 4
    mitad_parent1 =[]
    mitad_parent2 =[]
    Pc=random.randrange(100)
    if(Pc <= 70): #probabilidad del 0.7
        mitad_parent1=  parent1[punto_de_corte : ]
        mitad_parent2=  parent2[punto_de_corte : ]
        parent1= parent1[: punto_de_corte]
        parent2= parent2[: punto_de_corte]
        for elemento in mitad_parent2:
            parent1.append(elemento)
        for elemento in mitad_parent1:
            parent2.append(elemento)
        hijo1= Mutacion(parent1,NC)
        hijo2= Mutacion(parent2,NC)

        return [hijo1, hijo2]
    else:
        hijo1= Mutacion(parent1,NC)
        hijo2= Mutacion(parent2,NC)

        return[hijo1, hijo2]
```

2.5. Mutación

La función Mutación(cromsoma) recibe una individuo y recorre cada elemento de este, para determinar si algún elemento muta o no. La probabilidad de realizar la mutación es del 20%, por lo que si se obtiene un numero aleatorio (de entre 0 y 100) menor igual a 20, se muta el elemento, asignandole el valor: gen-(NC-1). Finalmente se retorna el cromosoma obtenido después del proceso de mutación.

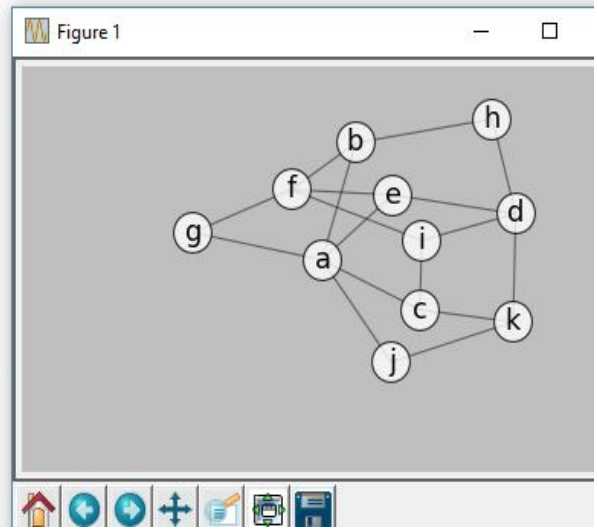
```
def Mutacion(cromosoma, NC):
    index=0
    for gen in cromosoma:
        Pm=random.randrange(100)
        if(Pm <= 20): #probabilidad del 0.2
            cromosoma[index] = abs(gen- (NC-1))
        index=index+1
    return cromosoma #retorna el cromosoma con los genes mutados
```

2.6. NetworkX para grafos

NetworkX es un paquete de Python para la creación, manipulación y estudio de la estructura, dinámica y funciones de redes complejas. NetworkX es una biblioteca de Python para estudiar gráficos y redes. Es un software gratuito lanzado bajo la licencia BSD-new. NetworkX incluye muchas funciones e instalaciones de generador de gráficos para leer y escribir gráficos en muchos formatos. En este proyecto, NetworkX se utilizó únicamente para visualizar el grafo. Para poder dibujar el grafo, NetworkX Incluye Matplotlib y una interfaz para usar el paquete de software de código abierto Graphviz. Estos forman parte del paquete networkx.drawing.

```
import os, sys
import networkx as nx
import matplotlib.pyplot as plt
import pylab

def ObtenerVerticesGrafo(nombre):
    lista=[]
    arch = open(nombre, 'r')
    linea = file.readline(arch)
    lista= linea.split(",")
    arch.close()
    lista.pop(len(lista)-1)
    return lista

def ObtenerAdyacenciaGrafo(nombre):
    lista=[]
    pardevertices=[]
    arch = open(nombre, 'r')

    lineal = file.readline(arch)
    for linea in arch:
        pardevertices= linea.split(",")
        pardevertices.pop(len(pardevertices)-1)
        lista.append(pardevertices)

    arch.close()
    return lista
```

```
def ConstruirGrafo(listaNodos, ADYACENCIA):
    G=nx.Graph()
    G.add_nodes_from(listaNodos)

    for i in range(0,len(ADYACENCIA)):
        ady1=ADYACENCIA[i][0]
        ady2=ADYACENCIA[i][1]
        G.add_edge(ady1,ady2)

    pos = nx.spring_layout(G)
    nx.draw_networkx_nodes(G,pos ,nodelist=listaNodos,node_color='w',node_size=500, alpha=0.8)
    labels={}

    for i in range(0,len(listaNodos)):
        labels[listaNodos[i]]=listaNodos[i]

    nx.draw_networkx_labels(G,pos ,labels,font_size=16)
    nx.draw_networkx_edges(G, pos ,width=1.0,alpha=0.5)
    print "Nodos: ", G.number_of_nodes(), G.nodes()
    print "Enlaces: ", G.number_of_edges(),G.edges()
    plt.axis('off')
    plt.show()


def GrafoNetworkx(NOMBREGRAFO):
    Nodos=ObtenerVerticesGrafo(NOMBREGRAFO)
    NodosAdyacentes=ObtenerAdyacenciaGrafo(NOMBREGRAFO)
    print Nodos
    print NodosAdyacentes
    ConstruirGrafo(Nodos, NodosAdyacentes)
```



2.7. Evaluación del algoritmo

Gracias a que el algoritmo, selecciona aleatoriamente valores entre 0 y el numero de colores (4), es posible, obtener una gran variedad de soluciones para colorear los vértices del grafo. A diferencia de otros algoritmos de coloración de grafos, el algoritmo genético no da preferencias a ciertos colores (por orden), asi pues cada vez que se corre el algoritmo se obtienen combinaciones distintas, y se pueden seleccionar desde el primer color hasta el último color, de una forma equilibrada, lo que permite obtener soluciones con colores mas distribuidos. El algoritmo genético encuentra la solución para gran variedad de grafos, por lo que se puede aplicar para problemas de asignación y coloreado de algunos mapas.

3. Resultados

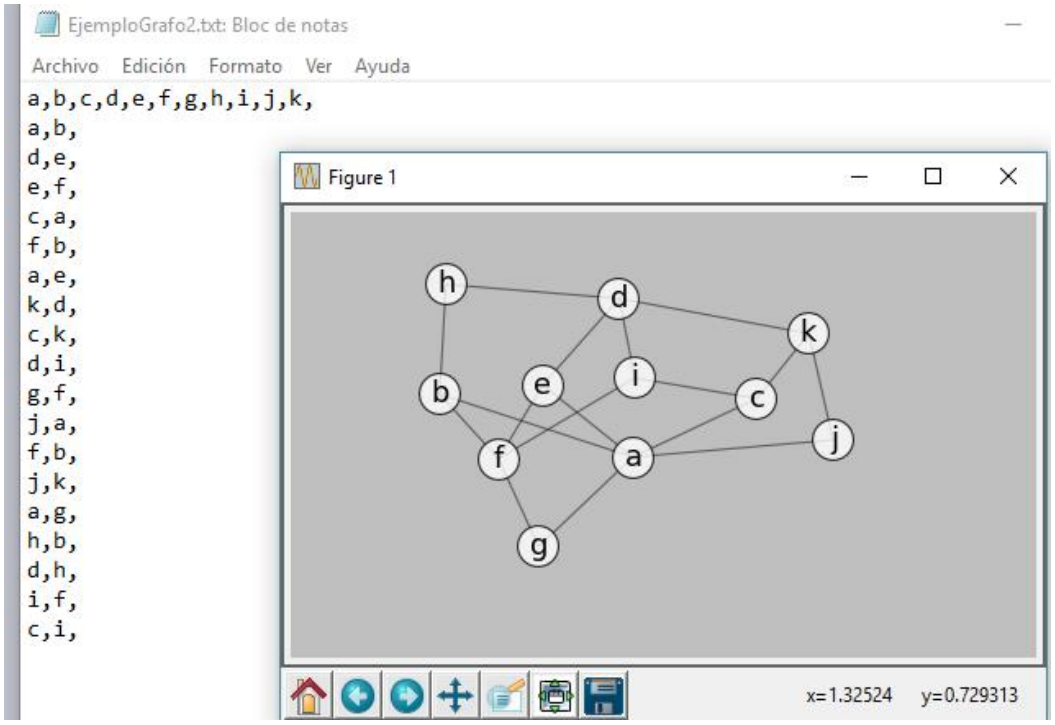
3.1. Vector Solución

El programa realiza la recursión hasta que algún cromosoma de la generación cumpla con los requisitos de solución, los cuales son: no puede haber vértices adyacentes con el mismo color. Al encontrar la solución, el vector retornado, se utilliza para mostrar el grafo y se imprime la información resultante (lista, número de generaciones realizadas, solucion, y datos del grafo) en pantalla para que el usuario pueda observar el comportamiento del algoritmo génetico. Nota el algoritmo no siempre encuentra la solución ya que depende de el grafo recibido y del número de colores establecidos (en este caso 4). Cuando el algoritmo sobrepasa el numero de iteraciones (50 generaciones), se imprime un mensaje informando que no se han podido colorear los vertices del grafo correctamente. Por ejemplo, el vector: [2, 1, 0, 2, 3, 2, 0, 0, 3, 1, 3, 0, 2, 3] es una solución para un grafo dado, y cada posición del vector corresponde a un vértice, mientras que el valor, es el color que se le asigna a tal vértice. Por ejemplo, la el índice 0 es el vértice “a” y tiene color “w” (white-blanco). El índice 1 es el vértice “b” y tiene color “blue” (azul).

```
 ¡¡¡ SE HA ENCONTRADO UNA SOLUCION !!!

El numero de iteraciones es:
29
[2, 1, 0, 2, 3, 2, 0, 0, 3, 1, 3, 0, 2, 3]
a = w
b = b
c = r
d = w
e = #2E8B40
f = w
g = r
h = r
i = #2E8B40
j = b
k = #2E8B40
l = r
m = w
n = #2E8B40
```

3.2. Base de datos

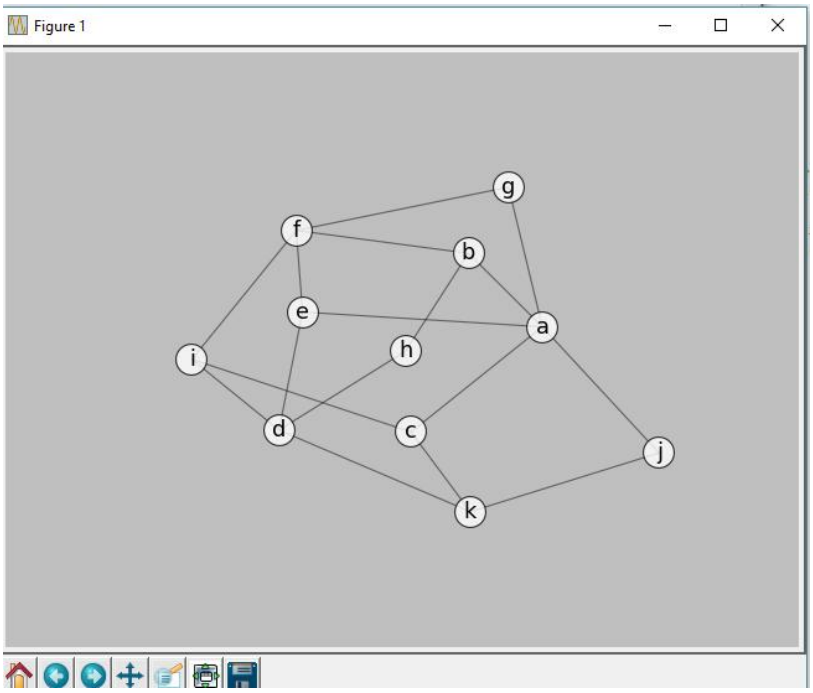
Se utilizan documentos de texto (.txt) para guardar la información que utiliza el programa: los colores, los nombres de los grafos existentes, los datos del grafo. El programa cuenta con funciones que permiten leer y traducir los datos del grafo para posteriormente usar esa información durante la ejecución del algoritmo. El grafo debe tener un formato específico: en la primera linea del txt se deben enlistar los nombres de los vertices separados por comas y finalizando el listado con una coma. Los pares de vértices adyacentes deben enlistarse en cada linea siguiente del txt, igualente finalizando la linea con coma (para cada par). Finalmente debe dejarse solo un renglón vacio (salto de linea) despues de enlistar todos los vértices adyacentes. El siguiente es un ejemplo de un documento de texto que cumple con el formato establecido, junto con el grafo al que corresponde:



3.3. **Visualización de los grafos**

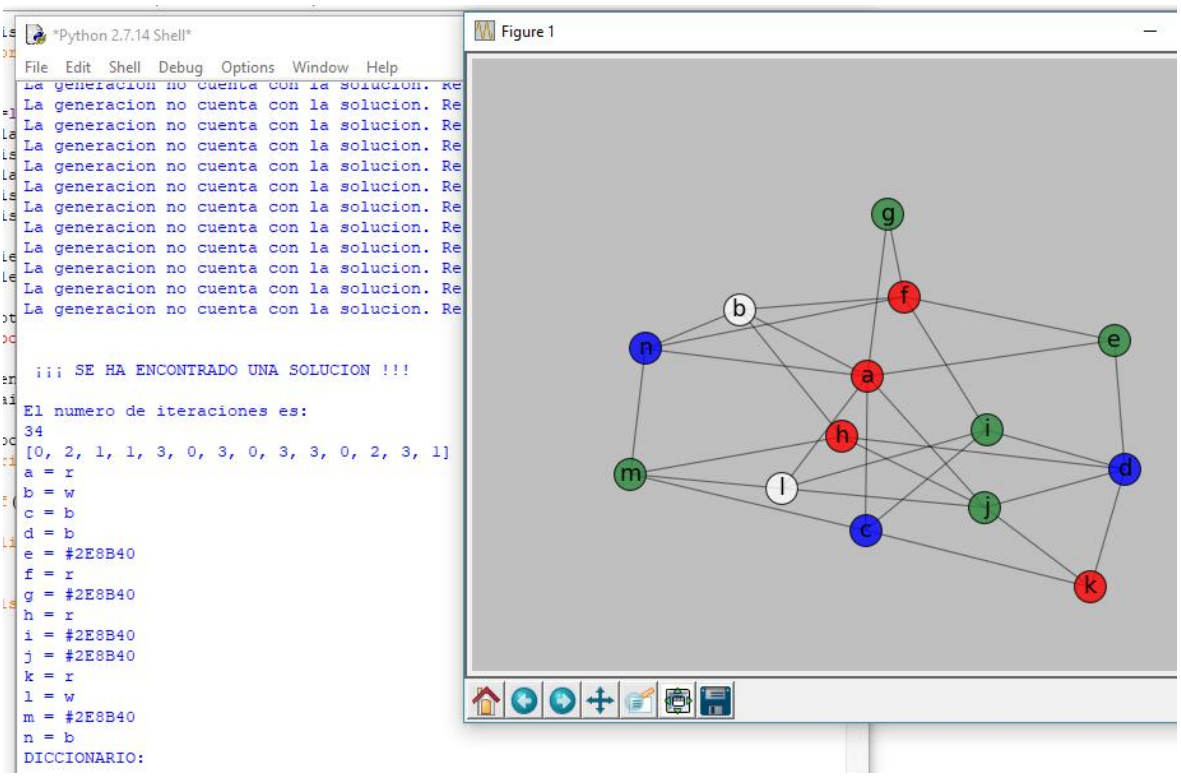
NetworkX permite visualizar los grafos. G.nx.Graph() crea el objeto grafo, incialmente esta vacio. G.add_nodes_from() agrega los nodos al grafo. Recibe una lista de vértices (obtenidas del archivo txt). G.add_edge() añade las aristas (que corresponden a los vertices adyacentes). nx.draw_network_ permite dibujar los elementos del grafo con las propiedades asignadas (color, tamaño, posición). Finalmente se utiliza matplotlib para mostrar el grafo en pantalla.

```
def ConstruirGrafo(listaNodos, ADYACENCIA):
    G=nx.Graph()
    G.add_nodes_from(listaNodos)

    for i in range(0,len(ADYACENCIA)):
        ady1=ADYACENCIA[i][0]
        ady2=ADYACENCIA[i][1]
        G.add_edge(ady1,ady2)

    pos = nx.spring_layout(G)
    nx.draw_networkx_nodes(G,pos ,nodelist=listaNodos,node_color='w',node_size=500, alpha=0.8)
    labels={}

    for i in range(0,len(listaNodos)):
        labels[listaNodos[i]]=listaNodos[i]

    nx.draw_networkx_labels(G,pos ,labels,font_size=16)
    nx.draw_networkx_edges(G, pos ,width=1.0,alpha=0.5)
    print "Nodos: ", G.number_of_nodes(), G.nodes()
    print "Enlaces: ", G.number_of_edges(),G.edges()
    plt.axis('off')
    plt.show()
```



3.4. Muestras correctas (soluciones)

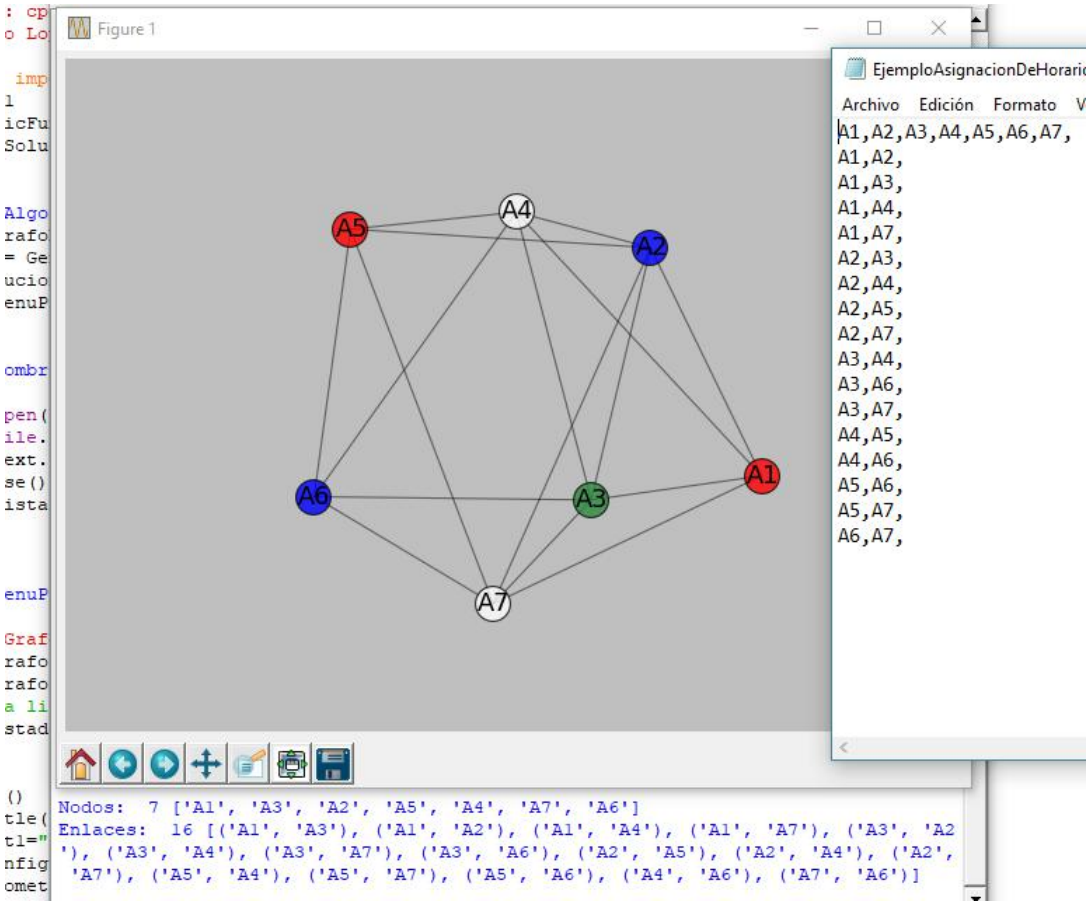
EjemploGrafo3



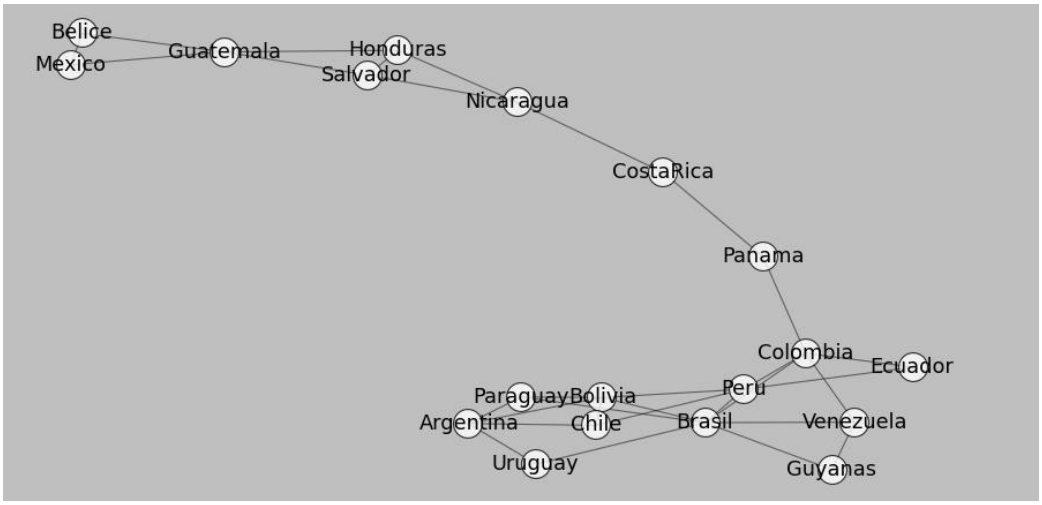
Ejemplo de asignación de dias para exámenes finales

El jefe de una escuela tiene que programar las fechas de los exámenes finales correspondientes a 7 asignaturas, que (A1, A2, A3, A4, A5, A6 y A7) de modo que se puedan distribuir en 4 dias de la semana. Se sabe que los siguientes pares de asignaturas tienen estudiantes en común: {A1, A2}, {A1, A3}, {A1, A4}, {A1, A7}, {A2, A3}, {A2, A4}, {A2, A5}, {A2, A7}, {A3, A4}, {A3, A6}, {A3, A7}, {A4, A5}, {A4, A6}, {A5, A6}, {A5, A7} y {A6, A7}. El problema de la programación de las fechas de exámenes puede modelarse medante un grafo, donde los colores representan los 4 dias de la semana, los vértices representan los exámenes de las asignaturas y los pares de adyacencia son los estudiantes que cursan ambas

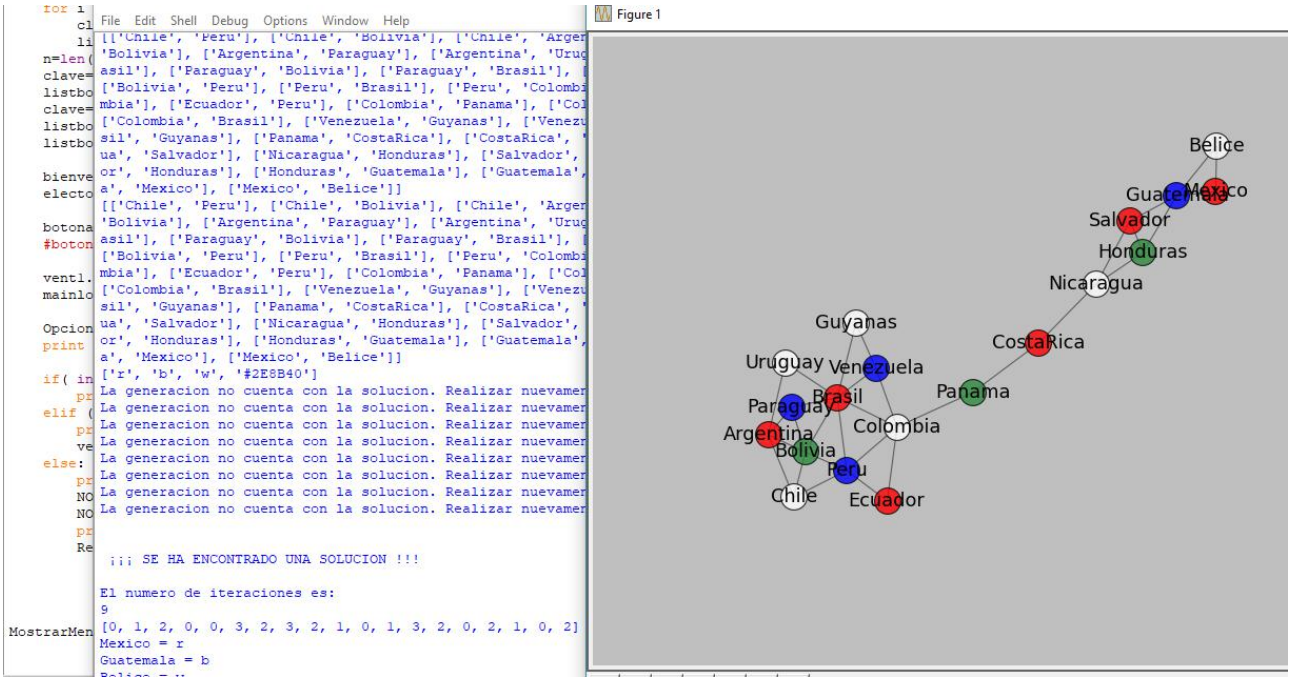
asignaturas. ¿De que manera pueden distribuirse los días para realizar todos los exámenes de modo que ningún estudiante tenga dos exámenes el mismo día?



Ejemplo Coloreado del mapa de latinoamerica



Se desea colorear el mapa de latinoamerica con el fin de que cada pais que comparta frontera con otro pais, sea de diferente color. En el Grafo no se incluyen los paises del Caribe ya que la mayoria no comparte fronteras terrestres.

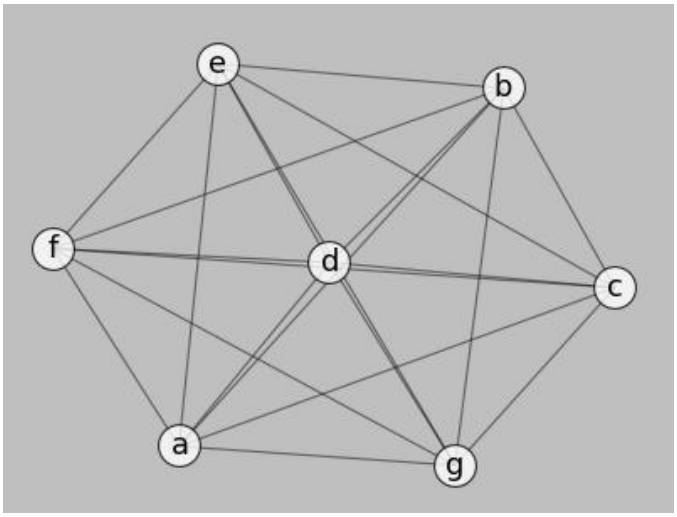


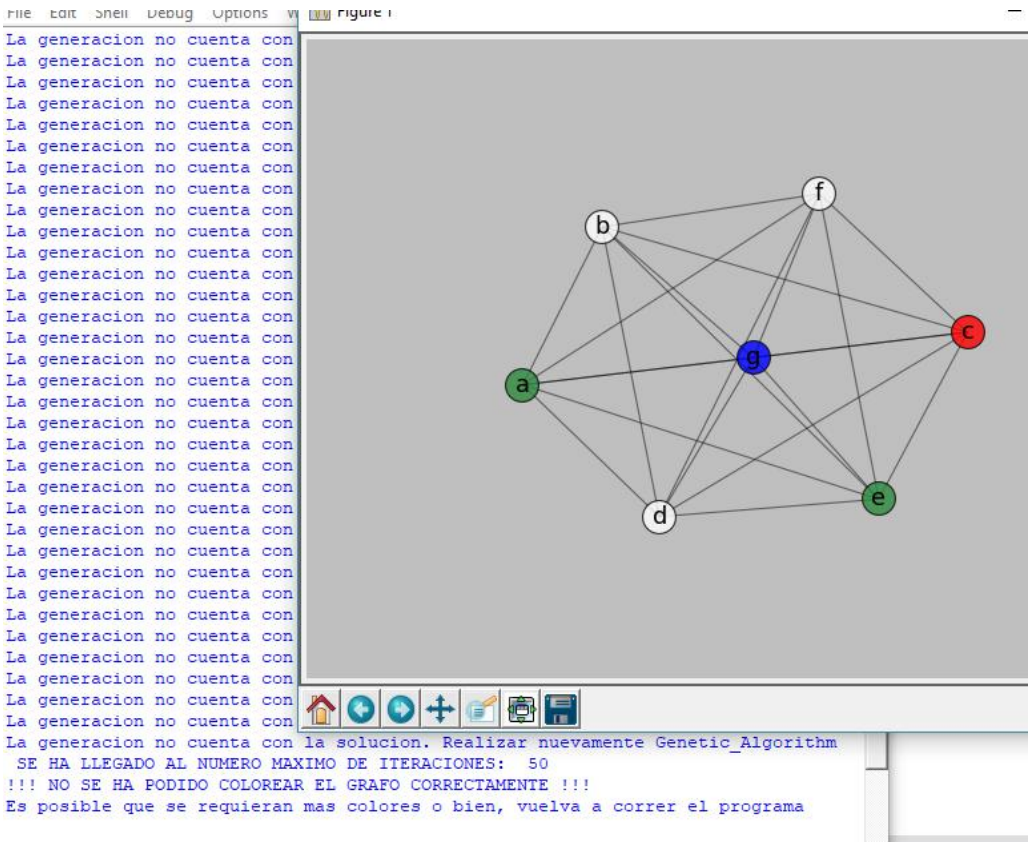
Para todos estos ejemplos, las soluciones fueron correctas pues todos los vértices adyacentes del grafo tienen diferente color.

3.5. Muestras incorrectas y errores

Debido a que solo se establecieron solo 4 colores, el algoritmo no encuentra solución para ciertos grafos, principalmente para los que son grafos completos (todos los vértices se unen entre sí). Este problema es debido a que no se implementó un algoritmo que calcule el número cromático (minimo número de colores necesarios) para colorear los vértices del grafo.El programa No puede encontrar la solucion para grafos completos de 5 o mas vértices.

EjemploError: Grafo completo de 7 vértices





Como se observa existen vértices adyacentes que tienen el mismo color, por lo que la solución no es correcta. Esto es debido a que no alcanza con solo 4 colores para poder colorear los vértices del grafo sin que exista conflictos.

4. Conclusión

4.1. Desempeño del algoritmo

El algoritmo es capaz de encontrar alguna solución para un gran número de grafos; sin embargo, no es el mejor en cuanto a eficiencia, ya que puede realizar muchas recursiones y no encontrar una solución correcta rápidamente (alcanza máximo de recursiones posibles), sobretodo para problemas complejos. Esto depende en gran medida del tamaño de la muestra, de la cantidad máxima de recursiones y de la generación de números aleatorios al inicio del programa. Si los números aleatorios son malos, las generaciones serán malas y al algoritmo genético le será dificil evolucionar para obtener mejores soluciones. Pero si por el contrario, las muestras inciales resultan ser aceptables, el algoritmo génetico podrá encontrar la solución mas rápido. Para que el algoritmo tenga un mejor desempeño y sirva una mayor variedad de grafos es necesario programar un algoritmo que calcule el mínimo número de colores que se requieren. Asimismo, la función Fitness podría mejorarse para realizar menos operaciones y poder encontrar una solución de mejor manera.

4.2. Explicación de errores encontrados

Los errores son debido a que, para un grafo, se puede exceder el número de colores necesarios de los ya establecidos (4 colores), lo que ocasiona que los vértices del grafo no se coloreen adecuadamente. El programa tambien puede fallar cuando el grafo es muy complejo, y/o cuando se alcanza el Máximo número de iteraciones posibles.

4.3. Conclusiones personales

Es muy interesante la forma en la que trabaja el algoritmo genético ya que cada nueva recursión presenta individuos (cromosomas) mejorados y con más posibilidades de ser una solución. La aplicación del algoritmo genético, para el problema de coloreado los grafos, ofrece una gran variedad de soluciones para un mismo problema, que no se puede lograr con otros algoritmos. Cada vez que se corre el algoritmo se pueden obtener combinaciones distintas con colores más distribuidos. El programa es capaz de resolver problemas medianamente complejos con un número de recursiones aceptable. Este algoritmo puede tener gran relevancia en aplicaciones de la vida real para resolver determinados problemas de asignación, como por ejemplo, la asignación de horarios, asignación de fechas de examen, etc. El programa puede mejorarse con una interfaz más intuitiva, ordenada y programando un algoritmo para obtener el número cromático que permita expander la variedad de grafos a resolver.