DECANATURA DE DIVISIÓN DE EDUCACIÓN ABIERTA Y A DISTANCIA
FACULTAD DE CIENCIAS Y TECNOLOGÍAS
TEORIA DE GRAFOS
Código SAC (16887)
TEÓRICO PRÁCTICA
EVALUACIÓN DISTANCIA 2020-2

OBJETIVOS DE LA EVALUACIÓN

- Identificar las características de sistemas que pueden ser representados y estudiados desde la teoría de grafos
- Comprender los principios sobre los cuales la teoría de grafos representa, evalúa y analiza sistemas discretos
- Aplicar algoritmos de optimización de caminos propios de la teoría de grafos.

VALORACIÓN DEL ESPACIO ACADÉMICO

Este espacio académico se valorará de acuerdo con lo estipulado en el Capítulo VII del Reglamento Estudiantil Particular de la VUAD. Así:

Evaluación Teórica:
-La Evaluación en línea (o presencial) tiene un valor del 50% de la calificación total del espacio académico.
-La Evaluación Distancia tiene un valor del 40% de la calificación total del espacio académico.
-Foro académico tiene un valor del 5% de la calificación total del espacio académico.
-Chat académico tiene un valor del 5% de la calificación total del espacio académico
Evaluación Teórico práctico:
Evaluación presencial tiene un valor del 50%;
Evaluación a distancia tiene un valor del 25%;
Evaluación práctica valor del 25%.

Evaluación Práctico:
Evaluación práctica tendrá un valor del 100%.

Nota: Tenga en cuenta que la adecuada citación, bajo las Normas APA 6ª edición, hace parte de los criterios de evaluación. Por ende, todo punto o trabajo que contenga fragmentos o en su totalidad no corresponda a su autoría o no tenga la citación adecuada tendrá como consecuencia la anulación del punto o evaluación.

ACTIVIDADES A DESARROLLAR

Estimados estudiantes:
Antes de comenzar el desarrollo de la Evaluación Distancia se le recomienda ir al Aula Virtual y descargar la "Rúbrica de Evaluación", ya que allí se encuentran claros los criterios a partir de los cuales se realizará la valoración y ponderación de las respuestas dadas a cada una de las preguntas. **¡Éxitos!**
Actividades a Desarrollar
Adoptado como Referencia:
https://runestone.academy/runestone/static/pythoned/Graphs/toctree.html

1. Primer Momento de Evaluación. El problema del giro del caballo

El rompecabezas del giro del caballo se juega en un tablero de ajedrez con una sola pieza de ajedrez, el caballo. El objetivo del rompecabezas es encontrar una secuencia de movimientos que permitan al caballo visitar cada cuadrado del tablero exactamente una vez. Una de esas secuencias se llama "gira". El rompecabezas del giro del caballo ha fascinado a los jugadores de ajedrez, matemáticos y científicos por igual durante muchos años. Se sabe que la cota superior del número de giras legales posibles para un tablero de ajedrez de ocho por ocho es 1.305×10351.305×1035; sin embargo, hay incluso un número mayor de posibles callejones sin salida. Claramente éste es un problema que requiere algo de buen cerebro, algo de verdadera potencia computacional, o ambos.
Aunque los investigadores han estudiado muchos algoritmos diferentes para resolver el problema del giro del caballo, una búsqueda de grafos es uno de los más fáciles de entender y programar. Una vez más vamos a resolver el problema utilizando dos pasos principales:

• Representar como un grafo los movimientos legales de un caballo en un tablero de ajedrez.

• Usar un algoritmo de grafos para encontrar una ruta de longitud filas×columnas−1filas×columnas−1 donde cada vértice del grafo se visite exactamente una vez.

1.1 Construcción del grafo del giro del caballo

Para representar como un grafo el problema del giro del caballo usaremos las dos ideas siguientes: Cada cuadrado en el tablero de ajedrez puede representarse como un nodo del grafo. Cada movimiento legal del caballo puede representarse como una arista del grafo. La Figura 1 ilustra, en un grafo, los movimientos posibles de un caballo y las aristas correspondientes.

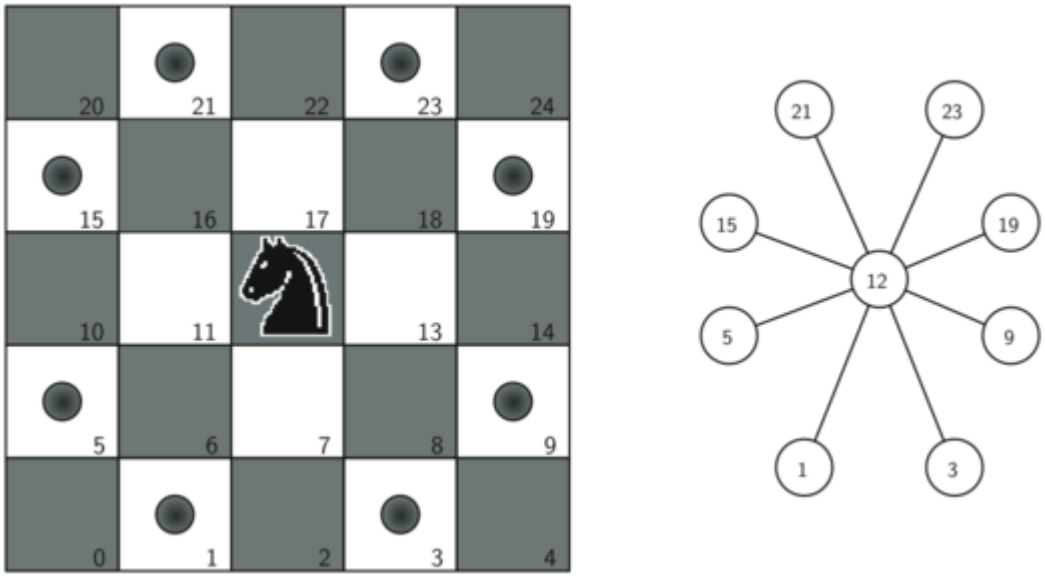

Figura 1: Movimientos legales para un caballo ubicado en el cuadrado 12, y el grafo correspondiente

Se puede usar la función en Python mostrada en el Programa 1 para construir el grafo completo de un tablero n-por-n, . La función grafoDelCaballo da una pasada por el tablero completo. En cada cuadrado del tablero la función grafoDelCaballo llama a una función auxiliar, generarMovLegales, para crear una lista de movimientos legales para esa posición en el tablero. Todos los movimientos legales se convierten en aristas del grafo. Otra función auxiliar pos_A_Id_Nodo convierte una posición en el tablero, dada originalmente en términos de una fila y una columna, en un número de vértice lineal similar a los números de vértice mostrados en la Figura 1.

Programa 1

```
from pythoned.grafos import Grafo

def grafoDelCaballo(tamanoTablero):

    grafoCbllo = Grafo()

    for fil in range(tamanoTablero):

        for col in range(tamanoTablero):

            idNodo = pos_A_Id_Nodo(fil,col,tamanoTablero)

            posicionesNuevas = generarMovLegales(fil,col,tamanoTablero)

            for e in posicionesNuevas:
```

```
            nid = pos_A_Id_Nodo(e[0],e[1],tamanoTablero)
            grafoCbllo.agregarArista(idNodo,nid)
    return grafoCbllo
def pos_A_Id_Nodo(fila, columna, tamano_del_tablero):
    return (fila * tamano_del_tablero) + columna
```

La función generarMovLegales (Programa 2) toma la posición del caballo en el tablero y genera cada uno de los ocho movimientos posibles. La función auxiliar coordLegal (Programa 2) asegura que un movimiento particular que se genere todavía esté aún dentro del tablero.

Programa 2

```
def generarMovLegales(x,y,tamanoTablero):
    nuevosMovimientos = []
    desplazamientosEnL = [(-1,-2),(-1,2),(-2,-1),(-2,1),
                ( 1,-2),( 1,2),( 2,-1),( 2,1)]
    for i in desplazamientosEnL:
        nuevoX = x + i[0]
        nuevoY = y + i[1]
        if coordLegal(nuevoX,tamanoTablero) and \
                    coordLegal(nuevoY,tamanoTablero):
            nuevosMovimientos.append((nuevoX,nuevoY))
    return nuevosMovimientos
def coordLegal(x,tamanoTablero):
    if x >= 0 and x < tamanoTablero:
        return True
```

```
else:
    return False
```

La Figura 2 muestra el grafo completo de los posibles movimientos en una tablero de ocho por ocho. Hay exactamente 336 aristas en el grafo. Note que los vértices correspondientes a las aristas del tablero tienen menos conexiones (movimientos legales) que los vértices del centro del tablero. Una vez más podemos ver cuán ralo es el grafo. Si el grafo estuviera completamente conectado, habría 4,096 aristas. Dado que sólo hay 336 aristas, la matriz de adyacencia estaría llena sólo en un 8.2 por ciento.

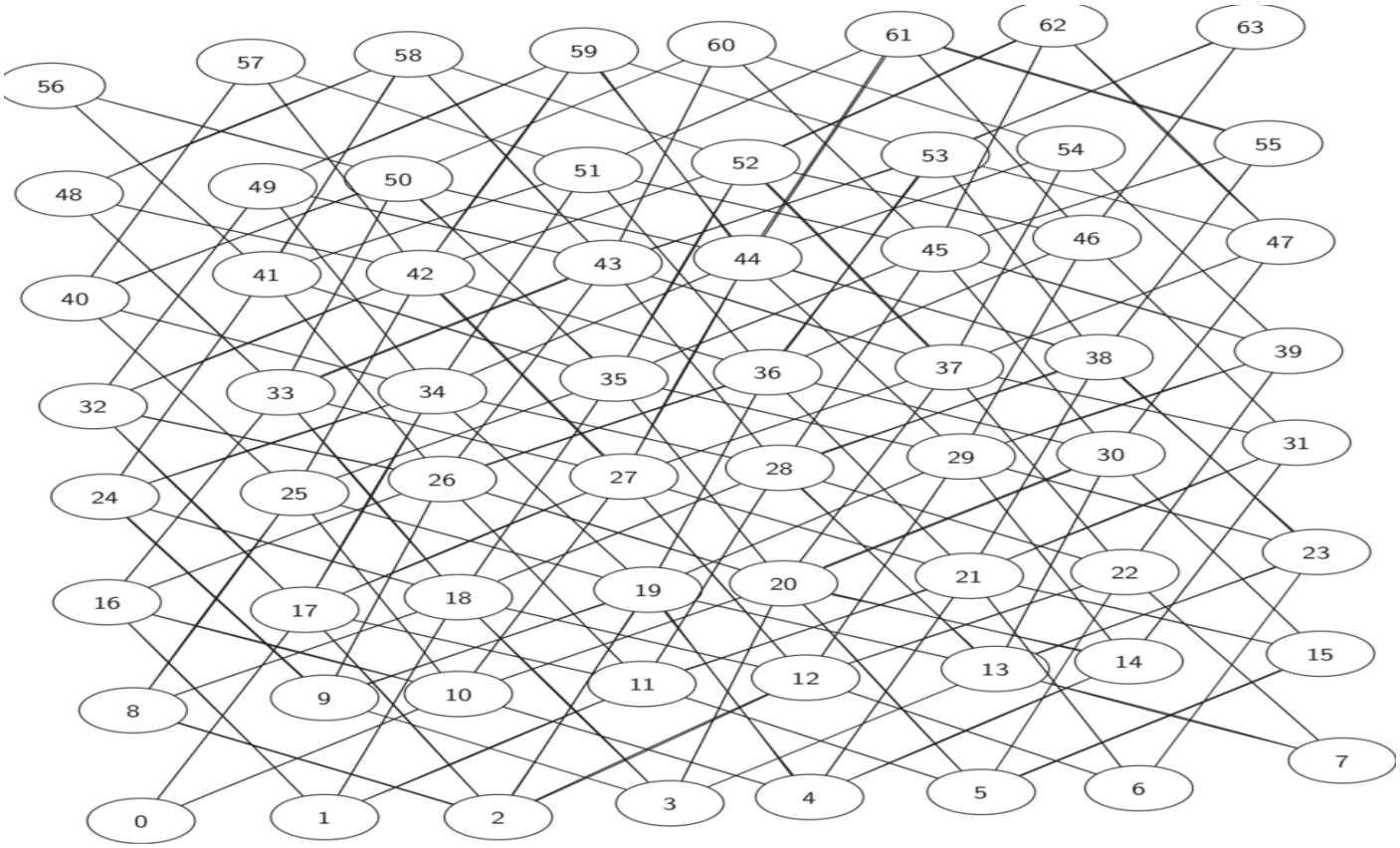

Figura 2. Posibles movimientos en una tablero de ocho por ocho

Actividad a Desarrollar. Implementar el TAD grafo en Lenguaje de Programación Python o Java.

2. Segundo Momento de Evaluación. Implementación de la gira del caballo

El algoritmo de búsqueda que se usa para resolver el problema de la gira del caballo se denomina **búsqueda en profundidad** (**BEP**). Mientras que el algoritmo de búsqueda en anchura discutido en la sección anterior construye un árbol de búsqueda un nivel a la vez, una búsqueda en profundidad crea un árbol de búsqueda explorando una rama del árbol lo más profundamente posible. En esta sección veremos dos algoritmos que implementan una búsqueda en profundidad. El primer algoritmo que veremos resuelve directamente el problema de la gira del caballo al prohibir explícitamente que un nodo sea visitado más de una vez. La segunda implementación es más general, pero permite que los nodos sean visitados más de una vez a medida que se construye el árbol. La segunda versión se utiliza en las secciones subsiguientes para desarrollar algoritmos de grafos adicionales.

La exploración en profundidad del grafo es exactamente lo que necesitamos para encontrar una ruta que tiene exactamente 63 aristas. Veremos que cuando el algoritmo de búsqueda en profundidad encuentra un callejón sin salida (un lugar en el grafo donde no hay más movimientos posibles), él retrocede en el árbol al siguiente vértice más profundo que le permita realizar un movimiento legal.

La función giraCaballo recibe cuatro parámetros: n, la profundidad actual en el árbol de búsqueda; ruta, una lista de vértices visitados hasta el momento; u, el vértice en el grafo que deseamos explorar; y limite el número de nodos en la ruta. La función giraCaballo es recursiva. Cuando se llama a la función giraCaballo, ella verifica primero la condición del caso base. Si tenemos una ruta que contiene 64 vértices, regresamos de giraCaballo con un estado de True, indicando que hemos encontrado una gira exitosa. Si la ruta no es lo suficientemente larga, seguimos explorando un nivel más profundo eligiendo un nuevo vértice para explorar y llamando a giraCaballo recursivamente para ese vértice.

BEP también utiliza colores para realizar un seguimiento de qué vértices del grafo se han visitado. Los vértices no visitados son de color blanco, y los vértices visitados son de color gris. Habremos llegado a un callejón sin salida si todos los vecinos de un vértice particular han sido explorados y aún no hemos alcanzado nuestra longitud objetivo de 64 vértices. Cuando llegamos a un callejón sin salida debemos retroceder. El retroceso sucede cuando volvemos de giraCaballo con un estado de False. En la búsqueda en anchura utilizamos una cola para realizar un seguimiento de qué vértice se debe visitar a continuación. Dado que la búsqueda en profundidad es recursiva, estamos usando implícitamente una pila como ayuda para nuestro retroceso. Cuando volvemos de una llamada a giraCaballo con un estado de False, en la línea 11, permanecemos dentro del ciclo while y examinamos el siguiente vértice en listaVecinos.

Programa 3

```
from pythoned.grafos import Grafo, Vertice
def giraCaballo(n,ruta,u,limite):
    u.asignarColor('gris')
    ruta.append(u)
    if n < limite:
        listaVecinos = list(u.obtenerConexiones())
        i = 0
        hecho = False
        while i < len(listaVecinos) and not hecho:
            if listaVecinos[i].obtenerColor() == 'blanco':
                hecho = giraCaballo(n+1, ruta, listaVecinos[i], limite)
            i = i + 1
        if not hecho:  # prepararse para retroceder
```

```
            ruta.pop()
            u.asignarColor('blanco')
        else:
            hecho = True
        return hecho
```

Veamos un ejemplo sencillo de giraCaballo en acción. Usted puede consultar las figuras que se muestran a continuación para seguir los pasos de la búsqueda. Para este ejemplo asumiremos que la llamada al método obtenerConexiones en la línea 6 ordena los nodos en orden alfabético. Comenzamos llamando a giraCaballo(0,ruta,A,6).

giraCaballo comienza con el nodo A (Figura 3). Los nodos adyacentes a A son B y D. Como B está alfabéticamente antes de D, BEP selecciona a B para la expansión subsiguiente como se muestra en la Figura 4. La exploración de B ocurre cuando giraCaballo es llamada recursivamente. B es adyacente a C y D, por lo que giraCaballo elige explorar C a continuación. Sin embargo, como se puede ver en la Figura 5, el nodo C es un callejón sin salida pues no tiene nodos blancos adyacentes. En este punto cambiamos el color del nodo C de nuevo a blanco. La llamada a giraCaballo devuelve un valor de False. El retorno de la llamada recursiva efectivamente retrocede la búsqueda al vértice B (ver la Figura 6). El siguiente vértice por explorar de la lista es el vértice D, por lo que giraCaballo hace una llamada recursiva moviéndose al nodo D (ver la Figura 7). Desde el vértice D, giraCaballo puede continuar haciendo llamadas recursivas hasta llegar nuevamente al nodo C (ver la Figura 8, la Figura 9 y la Figura 10). Sin embargo, esta vez, cuando llegamos al nodo C, la prueba n < limite falla, por lo que sabemos que hemos agotado todos los nodos del grafo. En este punto podemos devolver True para indicar que hemos hecho una gira exitosa del grafo. Cuando devolvemos la lista, ruta tiene los valores [A, B, D, E, F, C], que es el orden que necesitamos para recorrer el grafo y visitar cada nodo exactamente una vez.

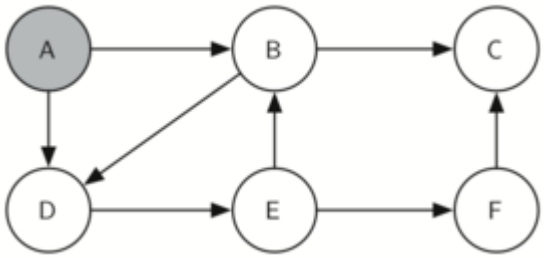

Figura 3: Comenzar con el nodo A

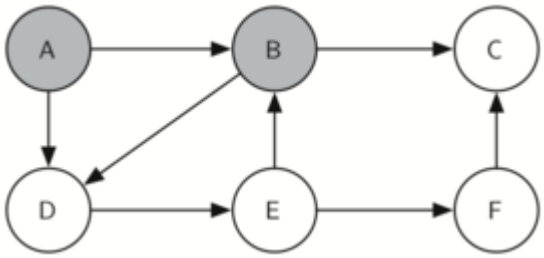

Figura 4: Explorar B

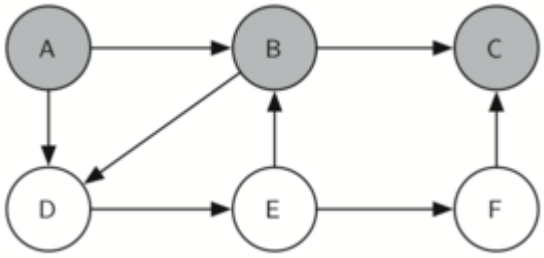

Figura 5: El nodo C es un callejón sin salida

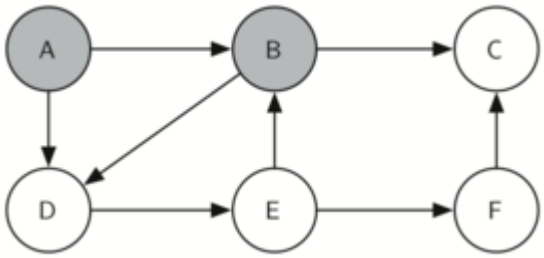

Figura 6: Retroceder a B

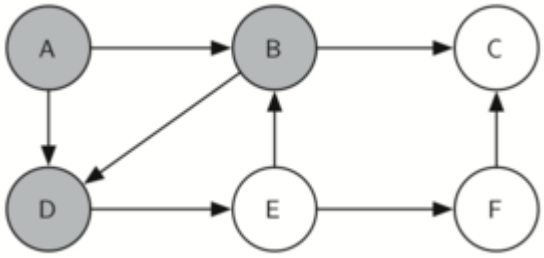

Figura 7: Explorar D

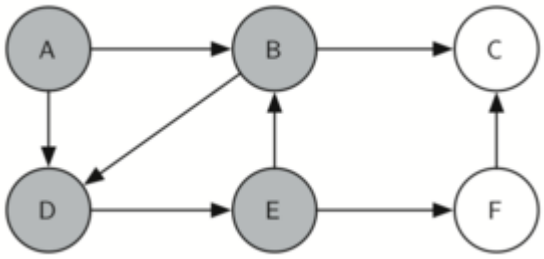

Figura 8: Explorar E

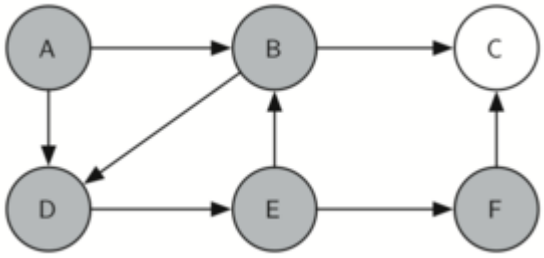

Figura 9: Explorar F

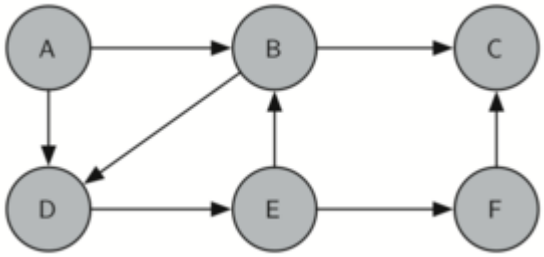

Figura 10: Finalización

La Figura 11 muestra cómo luce una gira completa en un tablero de ocho por ocho. Hay muchas giras posibles; algunas son simétricas. Con algunas modificaciones se pueden hacer giras circulares que comienzan y terminan en el mismo cuadrado.

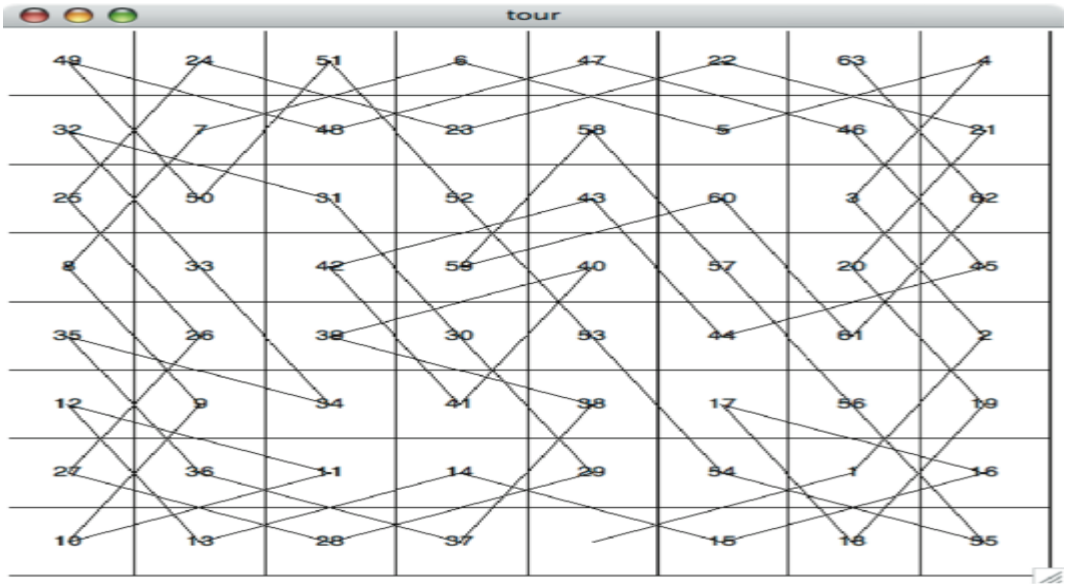

Figura 11: Una gira completa en el tablero

Actividad a Desarrollar. Implementar la matriz de adyacencia en Lenguaje de Programación Python o Java

Cuarto Momento de Evaluación. Algoritmo de Prim del árbol de expansión

Para tercer algoritmo de grafos vamos se considera un problema al que se enfrentan los diseñadores de juegos en línea y los proveedores de radio por Internet. El problema es que quieren transferir eficientemente una pieza de información a todos y cada uno de los que puedan estar escuchando. Esto es importante en los juegos para que todos los jugadores conozcan la posición más reciente de cada uno de los otros jugadores. Es importante también en la radio por Internet para que todos los oyentes que estén sintonizados estén recibiendo todos los datos que necesitan para reconstruir la canción que estén escuchando. La Figura 9 ilustra el problema de la radiodifusión.

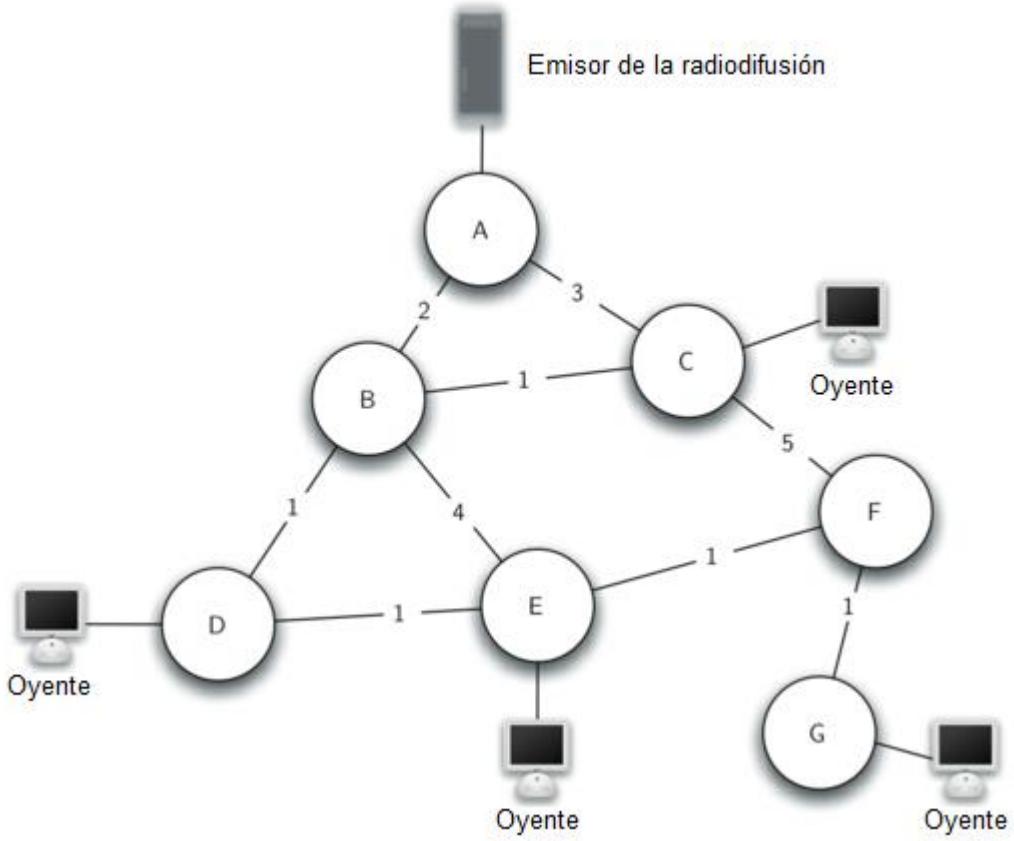

Figura 9: El problema de la radiodifusión

Hay algunas soluciones de fuerza bruta a este problema, así que mirémoslas primero para ayudar a entender mejor el problema de la radiodifusión. Esto también le ayudará a apreciar la solución que vamos a proponer cuando hayamos terminado. Para comenzar, el emisor de la radiodifusión tiene cierta información que todos los oyentes necesitan recibir. La solución más simple es que el emisor de la radiodifusión mantenga una lista de todos los oyentes y envíe mensajes individuales a cada uno. En la Figura 9 mostramos una pequeña red con un radiodifusor y algunos oyentes. Utilizando este primer

enfoque, se enviarían cuatro copias de cada mensaje. Suponiendo que se utilice la ruta de menor costo, veamos cuántas veces cada enrutador manejaría el mismo mensaje.

Todos los mensajes de la emisora pasan por el enrutador A, por lo que A ve las cuatro copias de cada mensaje. El enrutador C sólo ve una copia de cada mensaje para su oyente. Sin embargo, los enrutadores B y D verían tres copias de cada mensaje ya que los enrutadores B y D están en la ruta menos costosa para los oyentes 1, 2 y 3. Esto es un montón de tráfico extra si consideramos que el emisor de la radiodifusión debe enviar cientos de mensajes por segundo para una emisión de radio.

Una solución de fuerza bruta es que el emisor de la radiodifusión envíe una sola copia del mensaje de radiodifusión y deje que los enrutadores resuelvan las cosas. En este caso, la solución más sencilla es una estrategia llamada **inundación no controlada**. La estrategia de inundación funciona de la siguiente manera. Cada mensaje comienza con un valor de tiempo de vida (tdv) inicializado en un número mayor o igual al número de aristas entre el emisor de radiodifusión y su oyente más distante. Cada enrutador obtiene una copia del mensaje y pasa el mensaje a *todos* sus enrutadores vecinos. Cuando se transmite el mensaje, el tdv disminuye. Cada enrutador continúa enviando copias del mensaje a todos sus vecinos hasta que el valor tdv llegue a 0. Es fácil convencerse de que la inundación no controlada genera muchos más mensajes innecesarios que nuestra primera estrategia.

La solución a este problema radica en la construcción de un **árbol de expansión** de ponderación mínima. Formalmente definimos el árbol de expansión mínimo TT para un grafo G=(V,E)G=(V,E) como sigue. TT es un subconjunto acíclico de EE que conecta todos los vértices de VV. Se minimiza la suma de las ponderaciones de las aristas de TT.
En la Figura 10 se muestra una versión simplificada del grafo de radiodifusión, resaltando las aristas que forman un árbol de expansión mínimo para el grafo. Ahora para solucionar nuestro problema de radiodifusión, el emisor de la radiodifusión simplemente envía a la red una sola copia del mensaje de radiodifusión. Cada enrutador envía el mensaje a cualquier vecino que forme parte del árbol de expansión, excluyendo al vecino que acaba de enviarle el mensaje. En este ejemplo A reenvía el mensaje a B. B reenvía el mensaje a D y C. D reenvía el mensaje a E, el cual lo reenvía a F, y este último lo reenvía a G. Ningún enrutador ve más de una copia de cualquier mensaje, y todos los oyentes interesados ven una copia del mensaje.

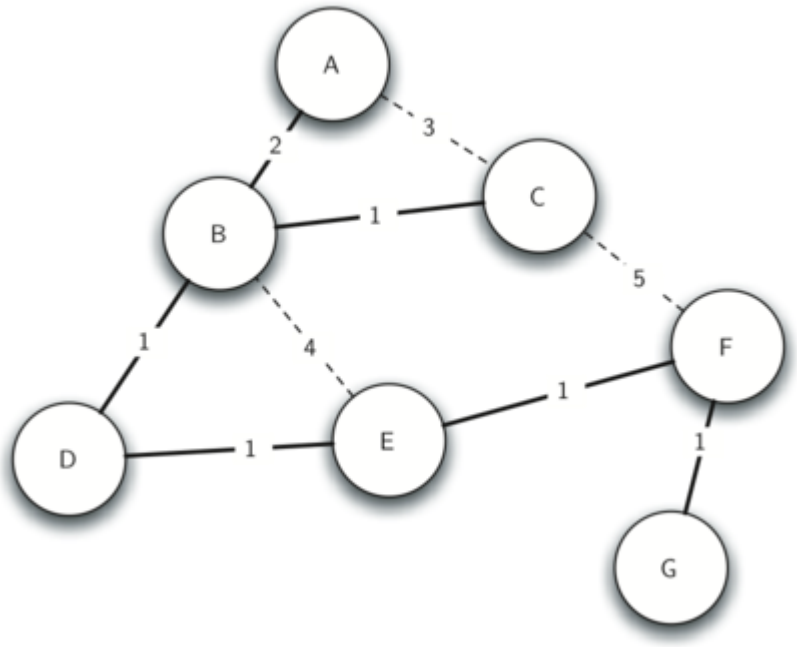

Figura 10: Árbol de expansión mínimo para el grafo de difusión

El algoritmo que usaremos para resolver este problema se llama el algoritmo de Prim. El algoritmo de Prim pertenece a una familia de algoritmos llamados “algoritmos codiciosos” porque en cada paso elegiremos el siguiente paso más barato. En este caso, el siguiente paso más barato es seguir la arista que tenga la ponderación más baja. Nuestro último paso es desarrollar el algoritmo de Prim.

La idea básica en la construcción de un árbol de expansión es la siguiente:

```
Mientras que T no sea aún un árbol de expansión
    Encontrar una arista que sea segura para agregarla al árbol
    Agregar la nueva arista a T
```

El truco está en el paso que nos lleva a “encontrar una arista que sea segura”. Una arista segura se define como cualquier arista que conecta un vértice que está en el árbol de expansión con un vértice que no está en el árbol de expansión. Esto asegura que el árbol seguirá siendo siempre un árbol y que, por lo tanto, no tendrá ciclos.

El código en Python para implementar el algoritmo de Prim se muestra en el Programa 2. El algoritmo de Prim es similar al algoritmo de Dijkstra porque ambos usan una cola de prioridad para seleccionar el siguiente vértice que se agregará al grafo en crecimiento.

Programa 2

```
from pythoned.grafos import ColaPrioridad, Grafo, Vertice

def prim(G,inicio):
    cp = ColaPrioridad()
    for v in G:
        v.asignarDistancia(sys.maxsize)
        v.asignarPredecesor(None)
    inicio.asignarDistancia(0)
    cp.construirMonticulo([(v.obtenerDistancia(),v) for v in G])
    while not cp.estaVacia():
        verticeActual = cp.eliminarMin()
        for verticeSiguiente in verticeActual.obtenerConexiones():
          nuevoCosto = verticeActual.obtenerPonderacion(verticeSiguiente)
          if verticeSiguiente in cp and nuevoCosto<verticeSiguiente.obtenerDistancia():
              verticeSiguiente.asignarPredecesor(verticeActual)
              verticeSiguiente.asignarDistancia(nuevoCosto)
              cp.decrementarClave(verticeSiguiente,nuevoCosto)
```

La siguiente secuencia de figuras (Figura 11 a Figura 17) muestra el algoritmo en funcionamiento sobre nuestro árbol de ejemplo. Comenzamos con A como vértice inicial. Las distancias a todos los otros vértices se inicializan en infinito. Observando a los vecinos de A podemos actualizar las distancias a dos de los vértices adicionales B y C porque las distancias a B y C, a través de A, son menores que infinito. Esto mueve a B y C al frente de la cola de prioridad. Actualizamos los enlaces a los predecesores para B y C haciendo que apunten a A. Es importante tener en cuenta que todavía no hemos añadido formalmente a B ni a C al árbol de expansión. Un nodo sólo se considera parte del árbol de expansión cuando es eliminado de la cola de prioridad.

Puesto que B tiene la menor distancia, examinamos después a B. Examinando a los vecinos de B vemos que D y E pueden ser actualizados. Tanto D como E obtienen nuevos valores de distancia y se actualizan

sus enlaces a los predecesores. Pasando al siguiente nodo en la cola de prioridad encontramos a C. El único nodo al que C es adyacente está todavía en la cola de prioridad, es F, por lo tanto podemos actualizar la distancia a F y ajustar la posición de F en la cola de prioridad.

Ahora examinemos los vértices adyacentes al nodo D. Encontramos que podemos actualizar E y reducir la distancia a E de 6 a 4. Cuando hacemos esto cambiamos el enlace al predecesor en E para que apunte a D, preparándolo así para ser injertado en el árbol de expansión, pero en un lugar diferente. El resto del algoritmo continúa como se esperaría, agregando cada nuevo nodo al árbol.

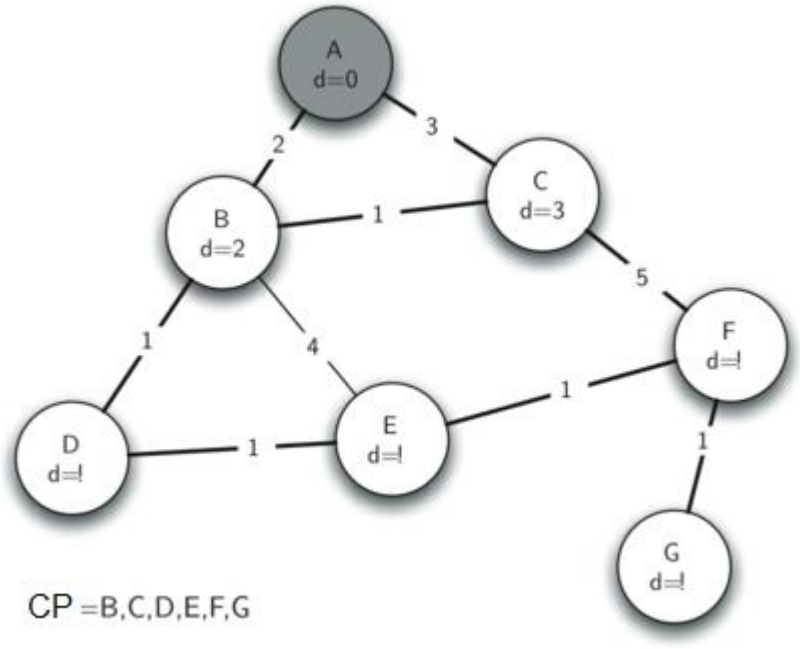

Figura 11: Seguimiento al algoritmo de Prim

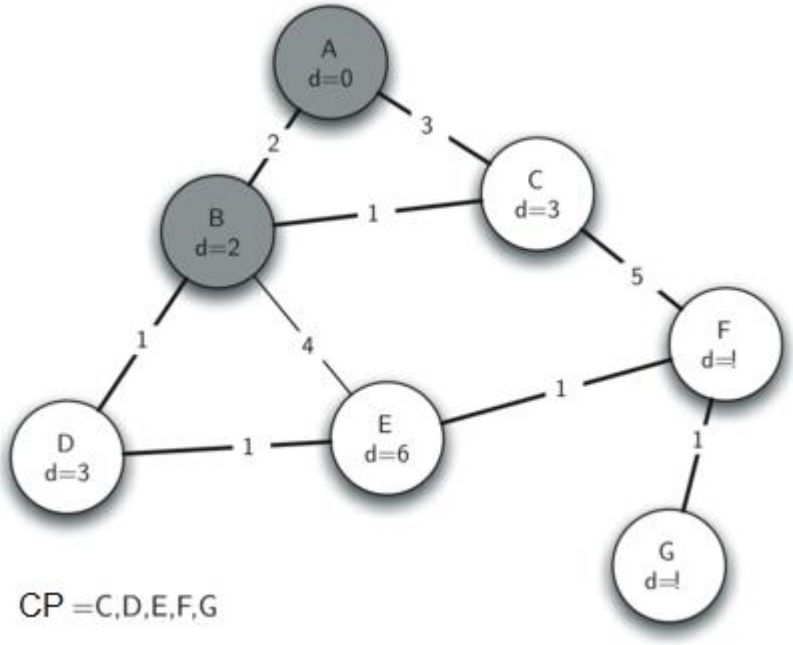

Figura 12: Seguimiento al algoritmo de Prim

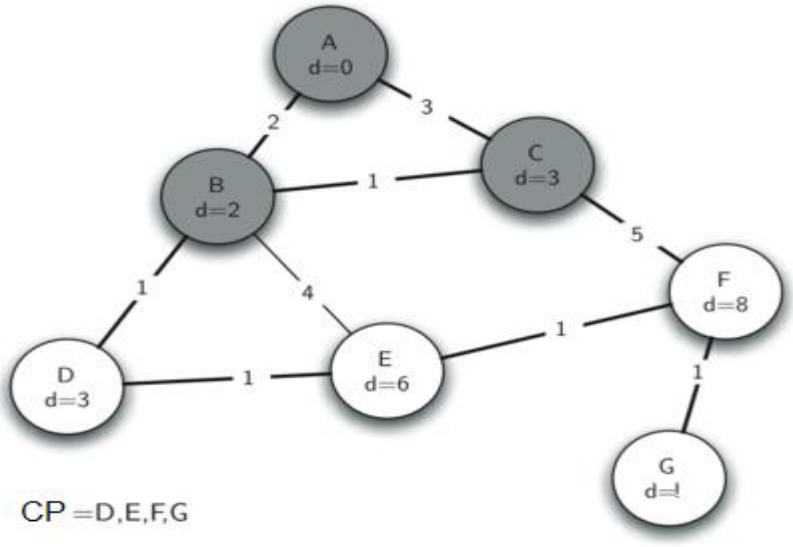

Figura 13: Seguimiento al algoritmo de Prim

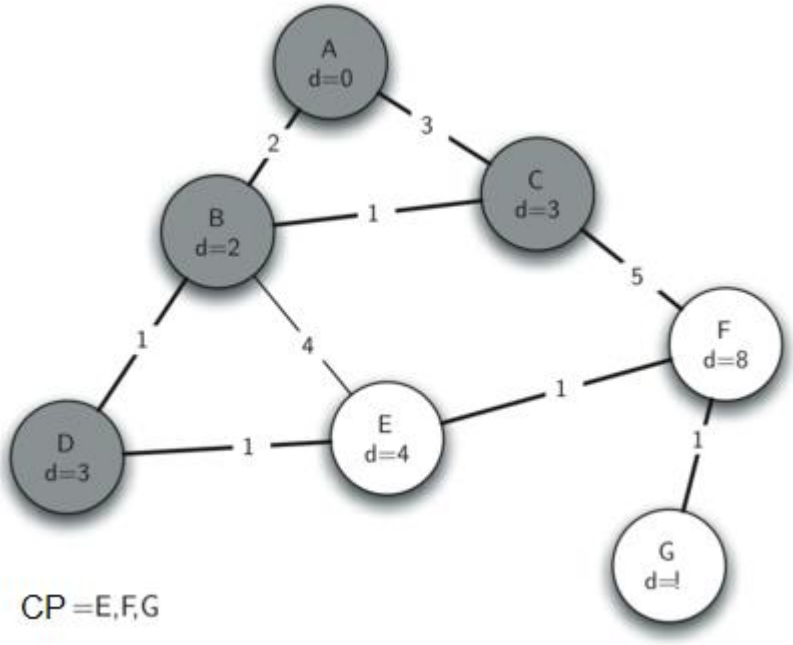

Figura 14: Seguimiento al algoritmo de Prim

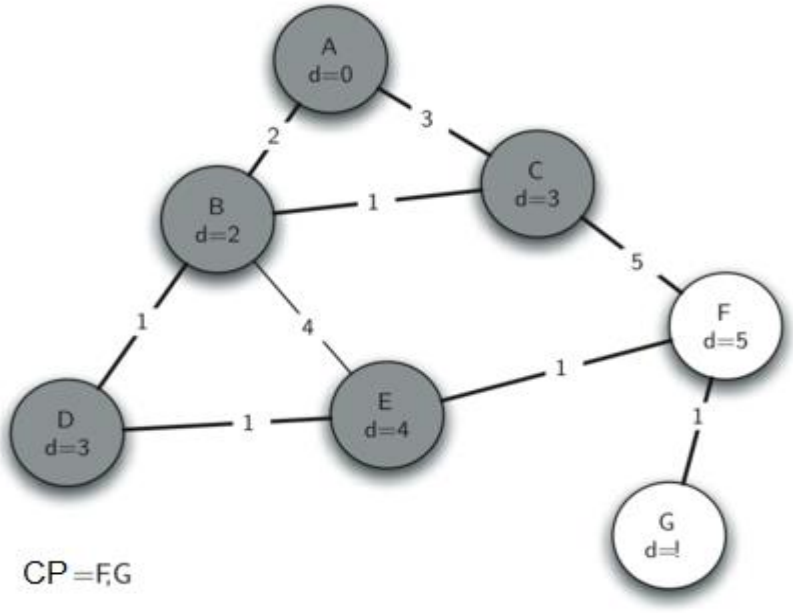

Figura 15: Seguimiento al algoritmo de Prim

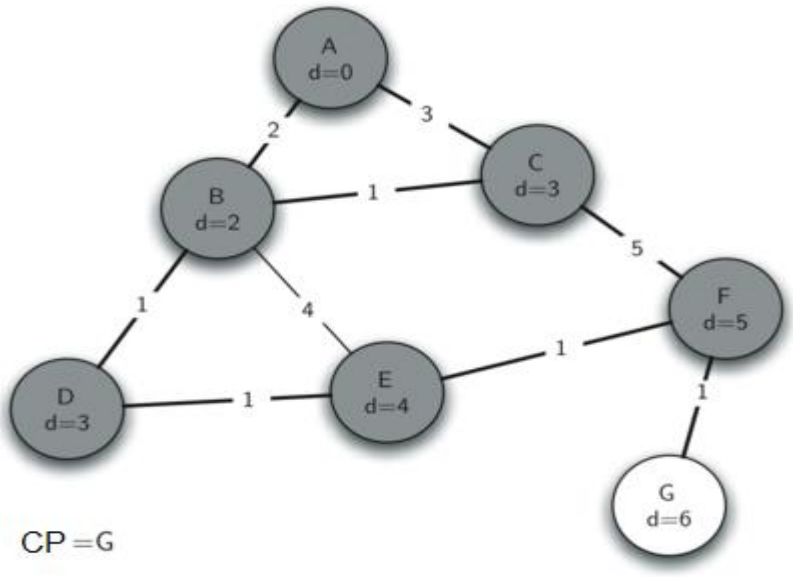

Figura 16: Seguimiento al algoritmo de Prim

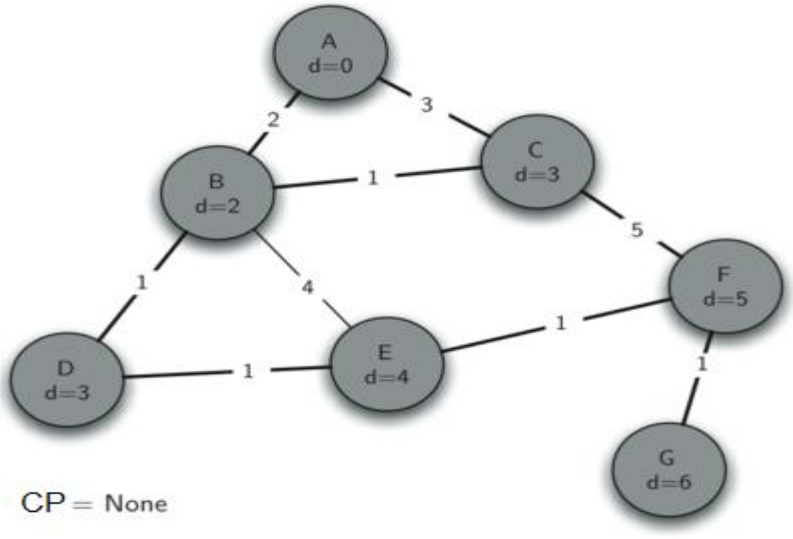

Figura 17: Seguimiento al algoritmo de Prim

Actividad a Desarrollar. Implementar el algoritmo de Prim en Lenguaje de Programación Python o Java