Universidad de Talca
Facultad de Ingeniería
Departamento de Ciencias de la Computación

Estructuras computacionales discretas

Tarea 2 - Informe

Erik Regla
eregla09@alumnos.utalca.cl

15 de Julio del 2014

1. Introducción

Dado un grafo, interesa encontrar ciertas propiedades de este, como por ejemplo si este es un *grafo conexo*, un *grafo bipartito*[1], o bien reportar si este grafo es *euleriano*[2]. Para los primeros dos problemas, se presenta una solución utilizando una modificación al algoritmo de búsqueda en anchura[3]. Para el tercer problema, se revisa el *Algoritmo de Tucker*[Tuc76] y el *Algoritmo de Hierhozer*[HW73] para resolver este problema.

2. Análisis del problema

2.1. TDA grafo

```
TDA Graph:
        Graph();
        Graph(const Graph& target);
        Graph(int vertex_count, bool directed);
        ~Graph();
        int V() const;
        int E() const;
        bool is_directed() const;
        int insert( int source,  int target);
        int remove( int source,  int target);
        bool edge( int source,  int target) const;
        std::vector<int> neighbours(int target);
        int get_random_neightbour(int target);
```

[1] Grafo en el cual se puede separar como dos conjuntos (A, B) en donde cada nodo de un conjunto A solo referencia elementos del conjunto B y viciversa.

[2] Que se puede recorrer todos sus vértices pasando una y solo una vez por cada una de sus aristas.

[3] Más conocido como *BFS*, acrónimo de *Breadth-first search*

2.1.1. Implementación

Para efectos de implementación, se escogió una TDA grafo general [Sed02], el cual soporta tanto grafos dirigidos como no dirigidos. Para esta se presentan las siguientes observaciones:

- Se eligió por implementar subyacentemente a la TDA un vector de mapas para almacenar la información de cada una de las aristas. La motivación recae en que si bien el uso de mapas para representar grafos constituye un costo doble en comparación a una lista de adyacencia, en el caso de encontrar aristas paralelas no es necesario almacenar k aristas en la lista. Esta es simulada gracias a `neighours`, la cual se encarga de interpretar el mapa en su representación final como vector unidimensional de vecinos, lo cual reduce enormemente la cantidad de datos a tener que cargar en RAM al momento de almacenar el grafo.
- De lo referido anteriormente, se obvia que en caso de búsqueda, esta se realiza en tiempo constante (en promedio) dado que el costo de acceder a la posición de un arreglo es constante, y también lo son las operaciones sobre mapas, por tanto, se puede decir que el acceso a los datos contenidos en este es constante.
- Para el caso de la obtención de los k vecinos a un nodo, se toma ventaja de la implementación de mapas, sin embargo, como se tiene que retornar un arreglo con todos los vecinos, esta operación tiene costo máximo V, en el caso promedio es $2\frac{|E|}{|V|}$ el cual es constante.
- La inserción y eliminacion discriminan entre un grafo dirigido o no, además de que la cuenta de aristas es calculada bajo la forma $|E|$ en el caso de que este sea dirigido y como $\frac{|E|}{2}$ en el caso que no lo sea, esto último en consecuencia de que la representación almacena dos veces la referencia al nodo.

2.2. Verificación de conectividad y biparticiones en grafos

2.2.1. Búsqueda en anchura

BFS es un algoritmo de búsqueda en grafos el cual cuenta con la particularidad de entregar un árbol en el cual todos sus niveles corresponden a la distancia de ese vértice a la del nodo en el cual se inició *BFS*[CSRL01]. Utilizando la implementación planteada en el Algoritmo 1, se pueden extraer las siguientes conclusiones.

- Si S contiene los mismos elementos que G, entonces el grafo es completamente conexo.
- Los elementos de G son buscados en orden de profundidad.
- El tiempo de ejecución para *BFS* es en el peor caso $O(|E|)$.

Por simple inspección del Algoritmo 1, combinando este con el mismo sistema de marcado que utiliza el algoritmo utilizado en la generación de los *Árboles Rojo-Negro*, es obvio que obtenemos al mismo tiempo las biparticiones de un grafo dado además de su verificación, la cual en caso

Algoritmo 1 BFS

Precondición: G es un grafo conexo, R es un nodo de G
Postcondición: *nil* si el elemento no es encontrado, T en caso contrario.

```
1: crear cola Q
2: crear conjunto S
3: insertar R en Q y en S
4: mientras Q no vacía hacer
5:    T ← Q.pop()
6:    si T es el elemento a buscar entonces
7:       devolver T
8:    si no
9:       para nodo N contenido en T no contenido en S hacer
10:         encolar N en Q
11:         insertar N en S
12:      fin para
13:   fin si
14: fin mientras
15: devolver nil
```

de ser negativa, finaliza inmediatamente el Algoritmo 2. También sabemos que si un grafo no es completamente conexo, se cumple que:

$$|V| > |V_{BFS}| \tag{1}$$

Donde, V son los nodos del grafo G y V_{BFS} es el resultado de *BFS* (Algoritmo 1) en algún nodo aleatorio de G. Ahora que se tienen los nodos que no están presentes en el primer resultado de la búsqueda, los cuales se denominarán X. Para cada conjunto resultante de X tenemos un nuevo grupo de nodos, en vista de este hecho, la cantidad de iteraciones de *BFS* necesarias para poder reducir a 0 la cantidad de elementos de X es en consecuencia la cantidad de componentes de G.

2.3. Busqueda de ciclos eulerianos en grafos

2.3.1. Descripción

Un ciclo euleriano se define como un circuito en donde se recorren todas las aristas de un nodo, solo pasando como máximo una sola vez sobre cada arista. Se le conoce como *grafo euleriano* (sus aristas se pueden recorrer utilizando un circuito simple) si y solo si todos los grados de sus vértices son pares y pertenecen a un solo componente conexo [Tuc76]. Esto resume a nuestro problema en verificar si el grafo cumple con el teorema antes mencionado para luego buscar un circuito en este. Este problema ya fue abordado con anterioridad por Tucker, quien presenta un método para poder encontrar el circuito en un grafo euleriano. Tucker presenta el Algoritmo 4 como solución a este problema, el cual consiste en separar los nodos que contienen más de dos vértices, en nodos que

Algoritmo 2 BFS Híbrido: Versión 1

Precondición: G es un grafo conexo, R es un nodo de G

Postcondición: **falso** si es que G no es bipartito, M_1 y M_2 conjuntos de la bipartición de G

```
1: crear cola Q
2: crear mapa M_1, M_2
3: insertar R en Q y en M_1
4: B ←verdadero
5: mientras Q no vacía hacer
6:    T ← Q.pop()
7:    si T es el elemento a buscar entonces
8:       devolver T
9:    si no
10:      para nodo N contenido en T no contenido en M_1 ni en M_2 hacer
11:         si N no contenido en M_1 ni en M_2 entonces
12:            encolar N en Q
13:            si B = verdadero entonces
14:               insertar N en M_1
15:               B ← falso
16:            si no
17:               insertar N en M_2
18:               B ← verdadero
19:            fin si
20:         si no
21:            si B = verdadero ∧ N = M_1 entonces
22:               devolver falso
23:            si no, si B = falso ∧ N = M_2 entonces
24:               devolver falso
25:            fin si
26:         fin si
27:      fin para
28:   fin si
29: fin mientras
30: devolver verdadero
```

Algoritmo 3 Verificación de conectividad

Precondición: G es un grafo conexo

Postcondición: N es el numero de componentes de G

```
1: N ← 0
2: G_temp ← G
3: mientras G_temp no vacío hacer
4:    G_temp ← BFS(G_temp)
5:    N ← N + 1
6: fin mientras
7: devolver N
```

solo contengan a lo más dos, luego encontrar ciclos simples y eliminarlos uno por uno para luego reensamblarlos de manera ordenada.

El resultado de este algoritmo es un set de vértices, incidentes el uno al otro. Luego repetidamente combinar cada uno de los sets obtenidos de modo de unirlos en una cadena cíclica, nuestro circuito. Aplicando inducción simple sobre la definición misma de grafo euleriano, podemos verificar que este algoritmo efectivamente encuentra un circuito.

Algoritmo 4 Algoritmo de Tucker

Precondición: G es un grafo conexo euleriano
Postcondición: C circuito euleriano de G

almacenar en G_1 pares de vértices de G de modo que
G_1 solo contenga vertices de grado 2, C_i número de componentes de G_1
mientras verdadero hacer
  si $C_i = 1$ **entonces**
    detener $C = G_i$
  si no
    encontrar dos componentes X, X' de G_i que compartan el vértice v_i en común.
    formar circuito C_{i+1}, comenzando desde v_1 atravezando X, X' y terminando en v_i.
  fin si
  $G_{i+1} = G_i - \{X, X'\} \cup C_{i+1}$
  $T = C_{i+1}$
  incrementar i
  $C_i \leftarrow$ numero de componentes de G_i
fin mientras
devolver C

Sin embargo, esta definición por más intuitiva que pueda parecer, no parece ser la más adecuada para poder ser implementada en un tiempo relativamente corto, como también ser explicada en profundidad a alguna persona que no tenga conocimientos previos. Por ese motivo, se elige como algoritmo a implementar el algoritmo de Hierholzer [HW73].

2.3.2. Algoritmo de Hierholzer

Hierholzer propuso en una publicación póstuma que para encontrar un ciclo euleriano en un grafo, basta con trazar un camino del cual se tiene conocimiento que no se atravezará dos veces por el mismo vértice, luego, cada vez que se llega a un ciclo cerrado, comenzar a regresarse por los nodos para luego encontrar otro. Cada arista elegida para regresarse debe ser descartado del grafo original, de modo que al final solo quedarán un grafo completamente desconexo. Entonces, las aristas descartados forman el circuito.

Dado que G es un grafo euleriano, es obvio que $\forall v \in V$ posee un número par de aristas, lo que implica que independiente del sentido del circuito, se puede utilizar cualquier nodo como punto de partida. Del mismo modo, $\forall C_i \in C$ cumple que está contenido en G, la prueba del Algoritmo 5 nuevamente se puede obtener vía inducción simple sobre el circuito.

Algoritmo 5 Algoritmo de Hierholzer

Precondición: G es un grafo conexo euleriano
Postcondición: C circuito euleriano de G

1: elegir $v \in V$
2: trazar circuito C_0 sobre $G - \{C\}$ comenzando en
   v, en cada paso, agregar la arista atravezada a C.
3: $i \leftarrow 0$
4: **mientras** $|C_i| \neq |G|$ **hacer**
5: elegir vértice $v_i \in C_i$ que sea incidente a algún vértice no contenido en C_i
6: generar circuito C'_i comenzando en v_i en el grafo $G - C_i$.
7: construir $C_{i+1} = C'_i + C_i$ comenzando por v_{i-1}
8: **fin mientras**
9: **devolver** C

Algoritmo 6 Implementación del algoritmo de Hierholzer

Precondición: G es un grafo conexo euleriano
Postcondición: C circuito euleriano de G

1: G_t copia de G
2: S pila vacía
3: S $\leftarrow v \in V$
4: **mientras** $|S| > 0$ **hacer**
5: **si** $\exists (a, b) \in G - \{C\}$ **entonces**
6: insertar b en S
7: $G_t \leftarrow G_t - \{(a, b)\}$
8: **si no**
9: insertar $S.top()$ en C
10: **fin si**
11: **fin mientras**
12: **devolver** C

Para efectos de implementación, se utiliza el el Algoritmo 6, el cual utiliza una pila para almacenar el nodo que actualmente se está visitando y una copia del grafo a resolver, el cual será usado para *marcar* las aristas por las cuales pasamos previamente. A modo de reducir el consumo de memoria, C es impreso de manera inmediata y no es almacenado. Por simple inspección del algoritmo anterior, no cabe duda que el tiempo de ejecución para este algoritmo es del $O(|E|+|V|)$, dado que es necesario recorrer todos los vertices y todas las aristas.

3. Modo de ejecución

3.0.3. Dependencias

- G++ 4.8
- makefile

Para generar los binarios se utiliza makefile. Siguiendo el estándar, también está disponible una opción `clean`.

```
make all           #genera todos los binarios
make euler         #genera solo binario para ciclo euleriano
make conexo        #genera solo binario para prueba de conectividad
make bipartito     #genera solo binario para verificar biparticiones
```

La ejecución de estos sigue el patron especificado en el enunciado, refiérase a el para mayor información.

4. Pruebas

Los resultados de las pruebas para el archivo `test.in` están en las Figuras 1, 2 y 3

5. Notas

- Las fuentes de *Blumenkranz*, el programa que implementa los algoritmos anteriormente descritos, está alojado en `https://bitbucket.org/eregla/ecd2014-1/` al igual que la prueba y tarea anterior.
- El programa está licenciado bajo una licencia abierta MIT.

```
./cicloEuleriano < test.in
```

4 4 1 0 3 2	NO
2 1 0	NO
3 2 1 2 2 0 2 2 0 1	NO
5 2 1 3 4 0 3 2 4 2 1 3 4 0 1 2 4 2 1 3	NO
6 1 3 0 5 2 4 3 5 4 1 0 2 1 2 1 2	SI Ciclo = 3 0 1 5 2 4 1 2 3

Figura 1: Salida de pruebas ara ciclos eulerianos

```
./bipartito < test.in
```

4 4 1 0 3 2	NO
2 1 0	SI partU = 0 partV = 1
3 2 1 2 2 0 2 2 0 1	NO
5 2 1 3 4 0 3 2 4 2 1 3 4 0 1 2 4 2 1 3	NO
6 1 3 0 5 2 4 3 5 4 1 0 2 1 2 1 2	NO

Figura 2: Salidas para pruebas de detección de biparticiones

```
./conexo < test.in
```

4 4 4 1 0 3 2	El numero de componentes es 2
4 2 1 0	El numero de componentes es 1
4 3 2 1 2 2 0 2 2 0 1	El numero de componentes es 1
4 5 2 1 3 4 0 3 2 4 2 1 3 4 0 1 2 4 2 1 3	El numero de componentes es 1
6 1 3 0 5 2 4 3 5 4 1 0 2 1 2 1 2	El numero de componentes es 1

Figura 3: Salidas para pruebas de conectividad

Referencias

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[HW73] Carl Hierholzer and Chr Wiener. Ueber die mÃ¶glichkeit, einen linienzug ohne wiederholung und ohne unterbrechung zu umfahren. *Mathematische Annalen*, 6(1):30–32, 1873.

[Sed02] Robert Sedgewick. *Algorithms in C++ Part 5: Graph Algorithms*, volume 5. Addison Wesley, 3 edition, 2002.

[Tuc76] Alan Tucker. A new applicable proof of the euler circuit theorem. *The American Mathematical Monthly*, 83(8):pp. 638–640, 1976.