Tarea 6 – Diseño y análisis de algoritmos

Santiago Celis – 202111131

Samuel Goncalves Vergara - 202122595

Daniel Escalante Pérez – 202122384

Parte 1. Soluciones

1. La alcaldía de Bogotá decidió comprar unas cámaras de vigilancia para la ciudad que tienen la capacidad de realizar video en 360 grados. Esto permite ubicarlas en intersecciones entre calles de manera tal que una cámara en una intersección permite vigilar al tiempo todas las calles que desemboquen a dicha intersección. Dado un mapa de calles e intersecciones de la ciudad, el distrito está interesado en saber el número mínimo de intersecciones en las que debe ubicar cámaras de manera tal que todas las calles queden vigiladas.

Especificación problema de optimización

E/S	Nombre	Tipo	Descripción
E	G	int[][]	Grafo no dirigido donde los nodos son las intersecciones y las calles los ejes.
S	S	Set of int	Subconjunto de vértices de G que representa las intersecciones donde se tiene que poner cámara.
S	i	nat	Número mínimo de intersecciones.

Precondición: $\{\forall i, j \mid 0 \leq i, j \leq |G| : G[i][j] = G[j][i]\}$

$$T(a, G) \equiv (|a| \leq |G|) \wedge (\forall k \in a| : 0 \leq k \leq |G| - 1) \\ \wedge (\forall i, j | 0 \leq i, j \leq |G| - 1 \wedge G[i][j] = 1 : i \in a \vee j \in a)$$

Postcondición: $\{T(S, G) \wedge (|s| = i) \wedge (\forall c \in Set\ of\ int \mid T(c, G) : i \leq |c|)\}$

Especificación de problema de decisión asociado

E/S	Nombre	Tipo	Descripción
E	G	int[][]	Grafo no dirigido donde los nodos son las intersecciones y las calles los ejes.
E	x	nat	Cota número superior de intersecciones
S	b	boolean	¿Existe un subgrafo de tamaño menor o igual a x que cubra todas las calles?

Precondición: $\{\forall i, j | 0 \leq i, j \leq |G| : G[i][j] = G[j][i]\}$

Postcondición: $\{b \equiv \left(\exists p \in Set\ of\ int \middle| : T(p, G) \wedge (|p| \leq x)\right)\}$

Especificación de problema de verificación asociado

E/S	Nombre	Tipo	Descripción

E	G	int[][]	Grafo no dirigido donde los nodos son las intersecciones y las calles los ejes.
E	x	nat	Cota número superior de intersecciones
E	v	Set of int	Intersecciones a verificar
S	b	boolean	¿Las intersecciones de v cubren todos los ejes y el número de intersecciones es menor o igual a x?

Precondición: $\{\forall i, j \mid 0 \leq i, j \leq |G| : G[i][j] = G[j][i]\ \}$

Postcondición: $\{b \equiv (T(v, G)\ \wedge\ |v| \leq x)\}$

```java
public static boolean VerificacionIntersecciones(int[][] grafo, int x, Set<Integer> v){

    /*Se verifica que el numero de interesecciones sea menor o igual a la cota ingresada por parámetro
     * y que el el tamaño del conjunto de vertices sea menor o igual al numero de vértices del grafo de netrada para ver si es subgrafo,
     * sino lo lo cumple se retorna false.
     */
    if (v.size() > x || v.size() > grafo.length){
        return false;
    }
    // Se verifica que todos los elementos del conjunto de vertices del subgrafo pertenezcan al grafo orginal
    for (int in: v){
        if (in<0 || in>grafo.length-1){
            return false;
        }
    }
    /*
     * Se recorre todos los ejes del grafo verficando que para cada eje (pareja de vértices), al menos uno de los dos vértices que componen cada eje
     * este en el conjunto de intersecciones ingresado por parámetro.
     * Si un eje no cumple esto, es que no hay un vértice que lo cubra por lo que se retorna false. De lo contrario todos los ejes estan cubiertos
     * y se retoran true al final
     */
    for (int i = 0; i<grafo.length;i++){
        for (int j = 0; j<grafo[i].length;j++){
            if((grafo[i][j] == 1 && (!v.contains(i) && !v.contains(j))) && i != j){
                return false;
            }
        }
    }
    return true;
}
```

Recorrido en grafos:

Espacio de estados: Posibles subconjuntos (subgrafos) del grafo original.

Condición de satisfabilidad: $T(a, G)\ \wedge\ |a| \leq x$

Estado inicial: Conjunto vacío

Función de sucesores: Dado un estado (subconjunto) se le agrega un nuevo vértice a ese subconjunto

$$suc(c) = \{c \cup s\ |:\ s \notin c \wedge s \in V\}$$

No hay ciclos porque por cada estado se crece por uno en el tamaño del conjunto de conjuntos, estos nunca decrecen por lo que no se devuelven y no hay ciclos.

$$viable(c) = |c| \leq x$$

2. Se diseñó un curso intensivo de un día que se va a dictar varias veces para atender a todos los estudiantes interesados. Se definieron unos días posibles en los que se va a dictar el curso y se

pidió a los interesados en el curso que seleccionaran los días en los que tendrían disponibilidad para tomarlo. Se desea encontrar el mínimo número de días en los que se debe dictar el curso para atender a todos los estudiantes interesados.

Especificación problema de optimización

E/S	Nombre	Tipo	Descripción
E	m	Set<Set<{Nat U {0}}>>	Conjunto de conjuntos donde los conjuntos representan días y el contenido son números que representan a los estudiantes.
E	n	Set<{Nat U {0}}>	Conjunto con números que representan los estudiantes.
S	c	{Nat U {0}}	Cantidad de días necesarios.
S	dias	Set<Set<{Nat U {0}}>>	Conjunto con los subconjuntos de los días necesarios.

Precondición: $\{\big(\forall g \in m\big|: (\forall i \in g|: i \in n)\big)\}$

Valid(h)≡ $\{(\forall k \ \in dias|: k \ \in m) \wedge (\cup j \ \in dias \ = \ n)\}$

Postcondición: $\{|dias| = c \wedge \text{Valid(dias)} \wedge (\forall y \in \text{Set} < \text{Set} < \text{Nat U } \{0\} >> |\text{valid(y)}: |\text{y}| \geq \text{c})\}$

Especificación de problema de decisión asociado

E/S	Nombre	Tipo	Descripción
E	m	Set<Set<{Nat U {0}}>>	Conjunto de conjuntos donde los conjuntos representan días y el contenido son números que representan a los estudiantes.
E	n	Set<{Nat U {0}}>	Conjunto con números que representan los estudiantes.
E	z	Nat U 0	Cantidad máxima de días necesarios
S	r	Boolean	Existe una cantidad menor a Z de días tal que puedan ir todos los interesados.

Precondición: $\{\big(\forall g \in m\big|: (\forall i \in g|: i \in n)\big) \wedge 0 \leq z \leq |m|\}$

Postcondición: $\{r \equiv (\exists y \in \text{Set} < \text{Set} < \text{Nat U } \{0\} >> |\text{valid(y)}: |\text{y}| \leq \text{z})\}$

Especificación de problema de verificación asociado

E/S	Nombre	Tipo	Descripción
E	m	Set<Set<{Nat U {0}}>>	Conjunto de conjuntos donde los conjuntos representan días y el contenido son números que representan a los estudiantes.
E	n	Set<{Nat U {0}}>	Conjunto con números que representan los estudiantes.
E	z	Nat U 0	Cantidad máxima de días necesarios.
E	P	Set<Set<{Nat U {0}}>>	Conjunto de conjuntos con números que puede representar una posible solución.
S	r	Boolean	P es un conjunto de días de tamaño menor a z talque todos los interesados puedan ir.

Precondición: $\{\big(\forall g \in m\big|: (\forall i \in g|: i \in n)\big) \wedge 0 \leq z \leq |m|\}$

Postcondición:$\{ r \equiv (valid(P) \wedge |P| \leq z)\}$

```java
public boolean verificacionDias(Set<Set<Integer>> m, Set<Integer> n, int z, Set<Set<Integer>> P) {
    Set<Integer> U = new HashSet<>();
    // Revisa que los dias elegidos sean de los proporcionados al inicio.
    // Adicionalmente, crea una union U con todos los elementos proporcionados en la posible respuesta
    for (Set<Integer> dia: P){
        if (!m.contains(dia)) return false;
        U.addAll(dia);
    }
    // Revisa que la union sea igual a n
    if (!U.equals(n)) return false;
    //Revisa que se cumpla la cota del problema
    return P.size() <= z;


}
```

Recorrido en Grafos:

Espacio de estados: Todos los posibles subconjuntos de m.

Condición de satisfabilidad: $valid(P) \wedge |P| \leq z$

Estado inicial: Conjunto vacío

Función de sucesores: Dado un estado, los sucesores son todos los conjuntos en el que se agrega un conjunto (dia) que no se tenga ya en el estado.

Sucesores(x) $= \{xU\{k\} | k \in (i \in m : i \notin x)\}$

No hay ciclos porque por cada estado se crece por uno en el tamaño del conjunto de conjuntos, estos nunca decrecen por lo que no se devuelven y no hay ciclos.

Viable(x) = $(|x| \leq z)$

3. Juan quiere invitar a sus amigos a conocer su nuevo apartamento. Sin embargo, tiene la dificultad de que sus amigos son algo conflictivos y entonces sabe que varias parejas de amigos se han peleado entre ellos. Debido a esto, tomó la decisión de organizar dos reuniones. Desafortunadamente, Juan ya se dio cuenta que no es posible distribuir a sus amigos en dos grupos de tal manera que dentro de cada grupo no haya parejas de personas que se hayan peleado entre ellas. El objetivo entonces es desarrollar un programa que reciba los amigos de Juan y las parejas de amigos que tienen conflictos y distribuya los amigos en dos grupos, de tal modo que, para la mayor cantidad posible de parejas con conflicto, los dos miembros de la pareja queden en grupos separados.

Especificación problema de optimización

E/S	Nombre	Tipo	Descripción
E	A	Set<Int>	Conjunto de amigos.
E	p	Set<int[]>	Parejas de amigos que tienen conflictos entre sí.
S	g	Set<Integer>[]	Estructura que representa los dos grupos de amigos.
S	c	Int	Cantidad de parejas con conflictos que quedaron en el mismo grupo.

En este problema la entrada p, es un conjunto cuyos conjuntos son de tamaño dos y tienen los índices de los amigos con conflicto. Estos índices son los del conjunto de entrada a.

La salida g, es un arreglo de tamaño dos, que contiene dos conjuntos con los índices de los amigos que conforman cada grupo.

Precondición: $\{\forall\, X\ \in p, \nexists\, Y\ \in p\ |: (X[0] = Y[0]\ ,\wedge,\ X[1] = Y[1])\ \vee (X[1] = Y[0]\ ,\wedge,\ X[0] = Y[1])\}$

Postcondición:

$$\{(c\ =\ CT(g,p))\ \wedge\ (\forall\, x\ \in\ \text{Set} < \text{Integer} > []|\ RP(g,A) : c\ \leq\ CT(x,\ p))\}$$

RP (a, b) $\equiv \{\forall x \in a.get(0), \nexists y \in a.get(1)|: x = y\} \wedge (|b| = |a.\text{get}(0)| + |a.get(1)| \wedge a.\text{get}(0)\ U\ a.get(1) = b)\ \}$

CT (a, b) = $\{k\ |: (\forall\, X\ \in\ b\ \wedge k\ =\ 0|\ x\ =\ X.get(0)\ \wedge\ y\ =\ X.get(1) : ((x,\ y\ \in\ a.get(0))\ \vee (x,\ y\ \in\ a.get(1)))\ \rightarrow\ k\ +1)\}$

Especificación de problema de decisión asociado

E/S	Nombre	Tipo	Descripción
E	A	Set<Int>	Conjunto de amigos.
E	p	Set<int[]>	Parejas de amigos que tienen conflictos entre sí.
E	num	Int	Cota maxima de cantidad de parejas con conflicto en un mismo grupo.
S	b	Boolean	¿Existe una distribución de amigos en dos grupos, de tal manera que la cantidad de parejas con conflicto en un mismo grupo sea menor o igual num?

Precondición: $\{\forall\, X\ \in p, \nexists\, Y\ \in p\ |: (X[0] = Y[0]\ ,\wedge,\ X[1] = Y[1])\ \vee (X[1] = Y[0]\ ,\wedge,\ X[0] = Y[1])\}$

Postcondición: $\{b\ \equiv\ ((\exists\, X\ \in\ \text{Set} < \text{Integer} > []|RP(X,A) : c\ =\ CT(X,p) \wedge\ c\ \leq\ num)\ )\}$

Especificación de problema de verificación asociado

E/S	Nombre	Tipo	Descripción
E	A	Set<Int>	Conjunto de amigos.
E	p	Set<int[]>	Parejas de amigos que tienen conflictos entre sí.
E	s	Set<Integer>[]	Estructura que representa los dos grupos de amigos a verificar.

E	num	Int	Cota de cantidad de parejas con conflicto en un mismo grupo.
S	b	Boolean	¿La distribución de amigos s tiene una cantidad menor o igual que num de parejas con conflicto en un mismo grupo?

Precondición: $\{\forall\ X\ \in p, \nexists\ Y\ \in p\ |: (X[0] = Y[0]\ ,\wedge,\ X[1] = Y[1])\ \vee (X[1] = Y[0]\ ,\wedge,\ X[0] = Y[1])\}$

Postcondición: $\{b\ \equiv (c\ =\ CT(s,p)\ \wedge\ c\ \leq\ num\ \wedge RP(s,A))\}$

```java
public static boolean verificacionParejas(Set<Integer> A, Set<int[]> p, Set<Integer>[] s, int num){
    boolean b = false;
    int count = 0;
    //Se verifica que la respuesta sea válida
    for(Integer amigo: A){
        boolean primero = false;
        for(Set<Integer> grupo: s){
            if(grupo.contains(amigo) && !primero){
                primero = true;
            } else if (grupo.contains(amigo) && primero){
                return false;
            }
        }
    }
    if(A.size() == (s[0].size() + s[1].size())){
        //Se recorre cada pareja con conflicto
        for(int[] pareja: p){
            //Se busca la pareja en cada grupo
            for(Set<Integer> grupo: s){
                //Si ambos amigos están en el mismo grupo, se aumenta el valor de count.
                if((grupo.contains(pareja[0])) && (grupo.contains(pareja[1]))){
                    count++;
                }
            }
        }
        //Se compara count con num
        if(count <= num){
            b = true;
        }
    }
    return b;
}
```

Recorrido en Grafos:

Espacio de estados: Todas las posibles distribuciones de amigos en A en un arreglo de dos sets.

Condición de satisfactibilidad: $c\ =\ CT(s,p)\ \wedge\ c\ \leq\ num \wedge\ RP(s,A)$

Estado inicial: Distribución con dos grupos vacíos, es decir, un arreglo con dos sets vacíos.

Función sucesora: Dado un estado representado por un array de tamaño dos, con dos sets cada uno de tamaño a y b, los sucesores serían los N posibles arreglos de tamaño dos con arreglos de tamaño a + 1 y b, o a y b + 1 respectivamente.

Sucesores(x): $\{x\ \in \text{Set} < \text{Integer} > []|: k\ \in A, (x.get(0) \cup k)\ \vee (x.get(1) \cup k)\}$

Parte 2. Implementación

```java
public Set<Set<Integer>> algoAprox (Set<Set<Integer>> m, Set<Integer> n){
    Set<Set<Integer>> res = new HashSet<>();
    // Itera agarrando el conjunto que este en m con mayor intersección con n.
    // Se borran esos elementos de n
    // Se añade a la respuesta el conjunto elegido
    // iteradamente vacia el n y cubre con los conjuntos en res el conjunto n.
    while (!n.isEmpty()){
        Set<Integer> elegido = hallarMaxInter(m, n);
        n.removeAll(elegido);
        res.add(elegido);
    }
    return res;
}

public Set<Integer> hallarMaxInter(Set<Set<Integer>> m, Set<Integer> n){
    int max = 0;
    Set<Integer> maxSet = new HashSet<>();
    // Revisa todos los sets y busca el de mayor intersección con n.
    for (Set<Integer> set: m){
        int suma = 0;
        for (Integer i : set) if (n.contains(i)) suma++;
        if (suma >= max) {
            max = suma;
            maxSet = set;
        }
    }
    return maxSet;
}
```

Bono:

```java
public Set<Set<Integer>> recorrido(Set<Set<Integer>> m, Set<Integer> n , int z) {
    Set<Set<Integer>> inicio = new HashSet<>();
    Queue<Set<Set<Integer>>> fila = new LinkedList<Set<Set<Integer>>>();
    fila.add(inicio);
    // Hace un recorrido con una fila donde revisa si el primero de la fila cumple, sino revisa los sucesores y añade
    // los que sean viables. Hace este proceso hasta vaciar la fila.
    while (!fila.isEmpty()) {
        Set<Set<Integer>> actual = fila.poll();
        if (sat(m, n, actual, z)) return actual;
        List<Set<Set<Integer>>> sucesores = sucesores(actual, m);
        for (Set<Set<Integer>> s: sucesores) {
            if (viable(s, z)) fila.add(s);
        }
    }

    return null;
}
```

```java
public boolean sat(Set<Set<Integer>> m, Set<Integer> n, Set<Set<Integer>> actual, int z) {
    return valid(m, n, actual) && viable(actual, z);
}

// Esta función encuentra todos los sucesores a un estado dado, busca los conjuntos que todavia no tiene en la respuesta y
// crea una lista de vertices con los vertices iguales al estado pero añadiendo uno de los conjuntos que no se tiene aun.
public List<Set<Set<Integer>>> sucesores(Set<Set<Integer>> x, Set<Set<Integer>> m) {
    List<Set<Integer>> posibles = new ArrayList<>();
    List<Set<Set<Integer>>> res = new ArrayList<>();
    for (Set<Integer> set: m) {
        if (!x.contains(set)) posibles.add(set);
    }
    for (int i = 0; i < m.size() - x.size(); i++){
        Set<Set<Integer>> nuevo = new HashSet<>(x);
        nuevo.add(posibles.get(i));
        res.add(nuevo);
    }
    return res;
}
```

```java
// La función revisa si una respuesta es valida al verificar que sean conjuntos validos y que la union de n.
public boolean valid(Set<Set<Integer>> m, Set<Integer> n, Set<Set<Integer>> dias) {
    Set<Integer> u = new HashSet<>();
    for (Set<Integer> set: dias) {
        if (!m.contains(set)) return false;
        u.addAll(set);
    }
    if (!u.equals(n)) return false;
    return true;
}


// Revisa la viabilidad del estado al verificar que el tamaño siga dentro de la cota.
public boolean viable(Set<Set<Integer>> x, int z) {
    return x.size() <= z;
}
```