Universidad Católica
San Pablo

Conjunto Dominante

Docente: Rensso Mora Colque

Integrantes:

Sebastian Gonzalo Postigo Avalos

Frank Roger Salas Ticona

Joaquin Casusol Escalante

Fabián Santiago Concha Sifuentes

Paolo Rafael Delgado Vidal

UCSP- Universidad Católica San Pablo
27 de junio de 2022

1. Abstract

En nuestro día a día trabajamos e interactuamos con grafos más de lo que creemos, y en ciertas situaciones aparecen problemas los cuales no podemos o no sabemos como resolver de la mejor manera como por ejemplo, imaginemos que trabajamos en un prestigioso museo o que simplemente queremos cuidar a nuestra familia y a nuestros bienes materiales y optamos por colocar cámaras para poder estar más tranquilos y tener todo perfectamente vigilado, pero aquí viene el problema, como colocamos esas cámaras, en que lugares, cuales son estos lugares estratégicos para los cuales las cámaras van a poder abarcar la mayor cantidad de espacio vigilado, pues bien, aquí entra el conjunto dominante de un grafo optimizándolo para usar la menor cantidad de vértices (en este caso cámaras) y llegar a todo el grafo (todo el espacio que queremos vigilar) el objetivo de esta investigación es presentar propiamente este problema, así como maneras óptimas para resolver esta naturaleza de problemas que nos podemos encontrar en un futuro de nuestras vidas.

2. Introducción

Un conjunto dominante se puede definir como un subconjunto de vértices **R** del conjunto de vértices **V** de un grafo **G** si y solo si todos los vértices en **V**-**R** son adyacentes a por lo menos un vértice en **R**.

Para tener una mejor claridad, vamos a definir este subconjunto R, primero tenemos que $R \subseteq V$ (esto debido a que R es un subconjunto de los vértices V de el grafo G). Entonces debemos tener en cuenta que este conjunto dominante tiene la particularidad de que todo vértice que no está en R es adyacente por lo menos a un vértice de R, por esto podríamos decir que con el conjunto R de vértices, podemos llegar a todos los vértices del grafo. Seguido a esto podemos incluir la idea de grafos conexos en nuestro problema del conjunto dominante expresando que un conjunto dominante es conexo si los vértices que están en R forman un subgrafo conexo.

Ahora bien, vamos a denotar al número dominante de un grafo como $\gamma(G)$, el cual en resumidas cuentas es el número de vértices del conjunto dominante de cardinal mínimo de G. De aquí llegamos al problema de optimización del conjunto dominante de G, el cual va a consistir en hallar un conjunto dominante de cardinal mínimo.

2.1. Ejemplo

Si tenemos un grafo **G** con el conjunto de vértices **V** $=\{v_1, v_2, v_3, v_4, v_5\}$ como se puede visualizar en la figura [1] y queremos comprobar si el subconjunto de **V**

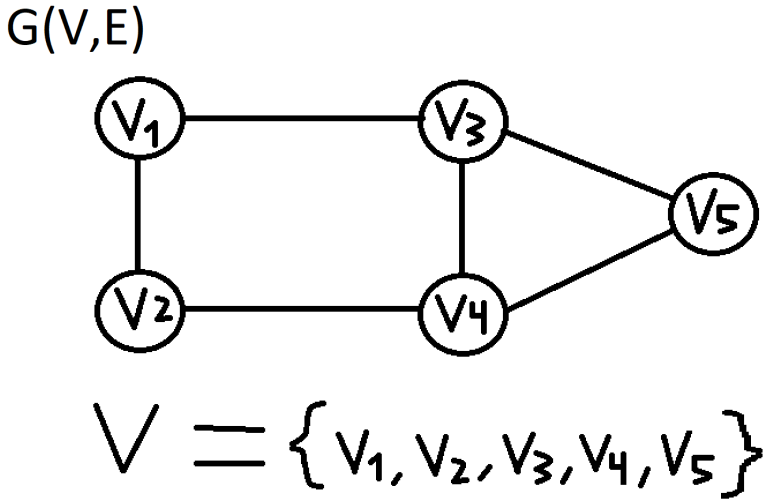
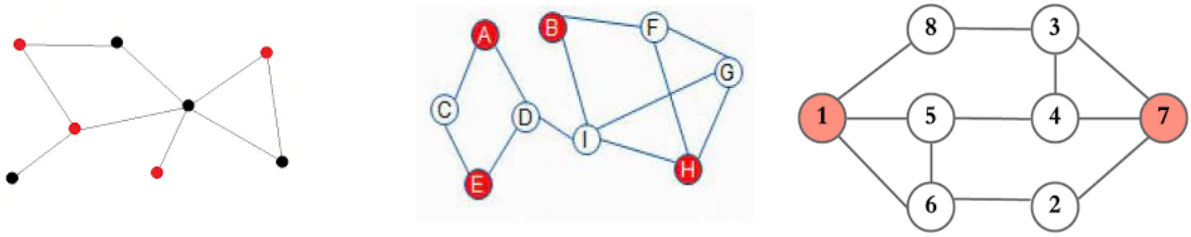


Figura 1: Ejemplo.

R_1=$\{v_1, v_2\}$ es un conjunto dominante de **G** solo tenemos que comprobar si todos los vértices en **V**-R_1=$\{v_3, v_4, v_5\}$ son por lo menos adyacentes a un vértice en R_1. En este caso v_3 es adyacente a v_1 por lo que cumple con la condición, v_4 es adyacente a v_2 por lo que también cumple con la condición y por ultimo v_5 no es adyacente a ningún vértice en R_1 por lo que no cumple con la condición y podemos confirmar que el subconjunto R_1 **no es un conjunto dominante**.

Ahora supongamos que tenemos el mismo grafo **G** pero esta vez queremos confirmar si el subconjunto de **V** R_2=$\{v_1, v_4\}$ es un conjunto dominante en **G**, para confirmar esto, usamos la misma prueba que antes con el subconjunto **V**-R_2=$\{v_2, v_3, v_5\}$ por lo que podemos ver que v_2 es adyacente a v_1 y v_4, v_3 es adyacente a v_1 y v_4. Por ultimo, v_5 es adyacente a v_4, por lo que, podemos confirmar que R_2 **es un conjunto dominante**.

Como ultimo ejemplo también usaremos el mismo grafo **G** pero esta vez queremos ver si el subconjunto de V $R_3 = \{v_3, v_4, v_5\}$ es un conjunto dominante, si sabemos que **V**-R_3=$\{v_1, v_2\}$ podemos concluir que v_1 es adyacente a v_3 y v_2 es adyacente a v_4 por lo que, R_3 **es un conjunto dominante**. Sin embargo como podemos ver en el anterior ejemplo R_2, R_3 no es el mínimo conjunto dominante del grafo **G** por lo que no podríamos usar este para resolver el problema del conjunto dominante.
A continuación algunos ejemplos mas sobre el conjunto dominante en los que el conjunto dominante de vértices estará resaltado.



2.2. Cobertura de vértices

La cobertura de vértices en un grafo se da cuando se escoge ciertos vértices pertenecientes a un grafo X, los cuales contienen todas las aristas de ese grafo. En el ámbito de la Ciencia de la Computación, existen dos problemas relacionados, los cuales son la mínima cobertura de un grafo y el problema de decisión de si se tiene un grafo existirá una cobertura con k vértices [1].

Algunos ejemplos fáciles de ilustrar sobre la cobertura de vértices pueden ser, todos los vértices del grafo, todos los vértices a los cuales estén conectados los vértices de otra cobertura. A continuación se muestran algunos ejemplos visuales de coberturas de grafos en la figura 2 donde los vértices coloreados de rojo representan la cobertura.

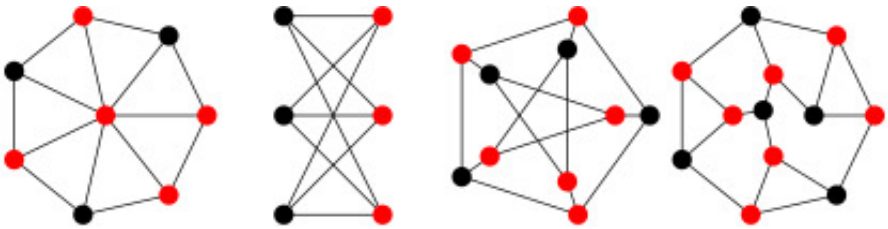


Figura 2: Ejemplos de cobertura de vértices.

Es bueno también tener en cuenta sus dos propiedades, siendo estas, el conjunto de vértices es un cubrimiento si y solo si el complemento de este es un conjunto independiente (un conjunto de vértices de un grafo los cuales no están conectados).

2.2.1. Cobertura mínima de vértices

Este problema trata de encontrar una cobertura tal que esta sea la que posea la menor cantidad de vértices posible, en tal sentido que este problema busca el resultado más óptimo posible. Por lo que este problema es bastante clásico en ámbitos de la optimización. Además este problema es sumamente complejo de aproximar [6], no es posible de aproximar a un factor menor a 2 debido a la *conjetura del juego único*. Por otra parte, se puede formular este problema como el número semi-entero de un

programa lineal cuyo programa dual linear es el problema máximo de apareamiento de grafos.

2.2.2. Problema de cubrimiento de vértices

Diferenciándose del anterior este problema no es de optimización sino de decisión. Este problema es uno de los 21 problemas de Karp siendo este un problema bastante clásico y recurrente en ámbitos de la teoría de complejidad computacional, es bastante usado también para demostrar que otros problemas son NPC, como es el caso en este documento, y como prueba para problemas de tipo NP-hard [1].

2.2.3. Aplicaciones

El problema de optimización sirve para muchos problemas teóricos como, en un centro comercial que este interesado en instalar cámaras de seguridad buscando la menor cantidad de cámaras en un circuito cerrado, cubriendo callejones y conectando todas las habitaciones [2]. También es usado para determinar la repetición de secuencias de ADN.

3. Demostración

3.1. NP y NP-completo

Los problemas que forman parte de la clase NP son aquellos que pueden ser resueltos por un algoritmo no determinista en tiempo polinomial, lo cual también puede ser expresado como los problemas que pueden ser resueltos por una máquina de Turing no determinista. Se debe de tener en cuenta que los problemas de la clase P, están incluidos en NP porque todo problema que pueda ser resuelto por un algoritmo determinista en tiempo polinomial puede ser resuelto por un algoritmo no determinista también en tiempo polinomial. Existe un problema que hasta la fecha no tiene una solución y prueba, y este es saber si $P = NP$.

Los problemas NP-completos pueden ser clasificados como tales si, estos problemas pueden ser verificables en tiempo polinomial y existe un algoritmo de fuerza bruta el cual encuentre dicha solución probando cada solución posible, además debe cumplir que el problema puede ser usado por cualquier otro problema para simularlo. Es por esto que los problemas de tipo NPC son los problemas más complicados de entre todos los problemas cuyas soluciones son verificables en tiempo polinomial. Si se encontrase alguna solución determinista en tiempo polinomial para un problema NPC entonces se podría encontrar la solución de cualquier otro problema de verificación polinomial [1].

3.2. Demostración de que el problema es NPC

Ahora, antes de la explicación de los algoritmos y Heurísticas encontrados para la resolución de este problema, vamos a demostrar que es un NP-Completo, lo que vamos a hacer es demostrar que el Conjunto Dominante se reduce a la Cobertura de Vértices, esto lo vamos a representar como: **Conjunto Dominante** $\propto$ **Cobertura de Vértices** [1].

Conjunto Dominante: Dado un número natural k y un grafo G(V,E) donde V es el conjunto de vértices y E es el conjunto de aristas,¿Existe un conjunto dominante de G de tamaño k? Para resolver esta premisa vamos a hacer uso de un problema NP-Completo, el problema de cobertura de vértices, y vamos a cambiar la pregunta de la siguiente forma:

Cobertura de Vértices: Dado un número natural k y un grafo G(V,E) donde V es el conjunto de vértices y E el conjunto de aristas, ¿Existe una cobertura de vértices de G de tamaño k? Tendremos el siguiente Grafo como entrada para resolver el problema planteado. Figura [3]

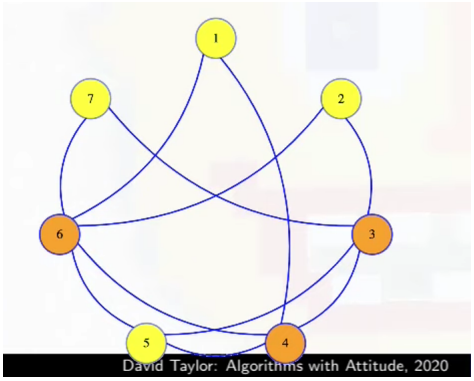


Figura 3: Grafo principal.

Seguido a esto vamos realizar una transformación en tiempo polinomial, primero vamos por comodidad a mover visualmente los vértices y aristas del grafo y posteriormente vamos a proceder con la parte más interesante de esta transformación, lo que procederemos a hacer será agregar un vértice, al que vamos a llamar vértice nuevo, por cada arista que une dos vértices originales, con vértices originales nos referimos a todos los que no son nuevos, entonces digamos que vamos a crear un nuevo vértice por cada arista que nos encontramos. Tenemos claro que esto se da en tiempo polinomial debido a que cada vértice que hemos agregado lo vamos a agregar dependiendo de cada arista que tenga el Grafo, entonces la complejidad del algoritmo de transformación del Grafo va a depender del número de aristas que éste contenga[8]. La nueva entrada nos quedaría algo así: Figura [4]:

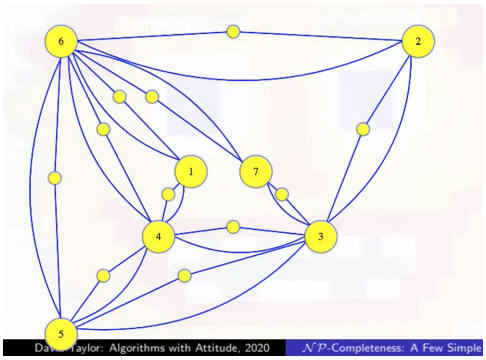


Figura 4: Grafo modificado.

Ahora bien. tenemos nuestra entrada transformada para poder aplicar el método de resolución. Hay que tener en cuenta ciertos hechos que ocurren en este grafo, los cuales son sumamente importantes, primero podemos ver que cada nuevo vértice añadido, une dos vértices originales y también reemplaza por decirlo así una arista original. Entonces aquí entra el concepto de Conjunto dominante, para lo cuál buscaremos 3 vértices que dominen a todos los nuevos vértices, cuando los encontremos, que en nuestro caso van a ser los vértices 6 4 y 3, nos damos cuenta que estos también dominan a los otros vértices, a los vértices originales, entonces hemos dado con un conjunto dominante [8]. Figura [5]

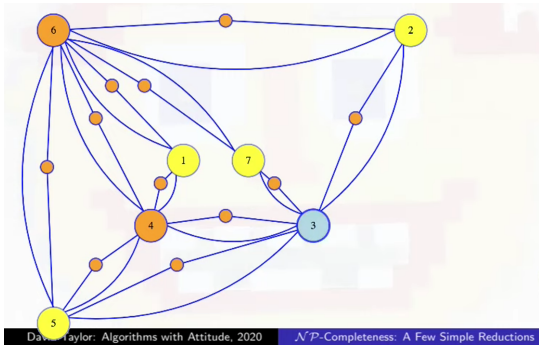


Figura 5: Conjunto Dominante.

Entonces, si somos lo suficientemente observadores, nos damos cuenta que con esta configuración de 3 vértices que pertenecen al conjunto dominante del grafo, también tenemos una configuración de cobertura de vértices, es decir, con estos 3 vértices llegamos a todas las aristas originales del grafo. Lo que trasladándolo de nuevo a la figura original del grafo, damos con que hemos encontrado 3 vértices con los que podemos formar una cobertura de vértices [8]. Figura [6]. Figura [7]

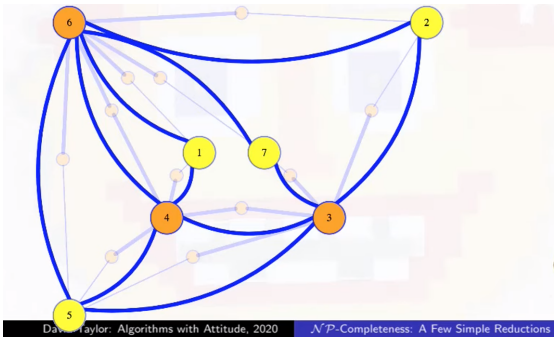


Figura 6: Cobertura de Vértices.

Dado que hemos podido resolver la cobertura de vértices utilizando una solución para el conjunto dominante, entonces podemos decir que El Conjunto Dominante es reducible en tiempo polinomial a la Cobertura de Vértices el cual es un problema NP-Completo. Por estas razones concluimos que el Conjunto Dominante es un NP-Completo y nuestra demostración habría acabado.

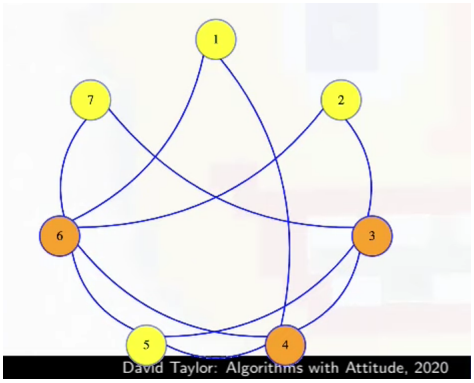


Figura 7: Cobertura de Vértices2.

4. Fuerza bruta

Podemos definir un algoritmo de fuerza bruta como un algoritmo que es capaz de encontrar la respuesta de un problema, no importa que tan complicado sea. Para conseguir esto, probara con todos los posibles caminos para llegar a una solución. Al mismo tiempo, esto significa que estos algoritmos son muy ineficientes ya que toman mucho tiempo para completar.

4.1. Explicación de código fuerza bruta

Primero el código recibiría una matriz de adyacencia que estará definida en un archivo.txt, después de leer los contenidos de este y transformar esos contenidos en una matriz 2d, entraremos en nuestro bucle principal que tendrá 2^n iteraciones, n siendo la cantidad de vértices que pertenecen a la matriz de adyacencia. Usaremos 2 a la potencia de n como rango en nuestro bucle porque de esta manera podemos recorrer todas las posibles combinaciones que tenemos con los vértices si representamos los números como si fueran bits y la posición de los bits como si fueran la posición de los vértices. Por ejemplo, con el numero 53 lo representamos en bits como 110101 en este caso eso significaría que los vértices=$\{v_1, v_3, v_5, v_6\}$ son los seleccionados por el numero 53, como podemos ver usamos los bits como si fueran booleanos para saber cuales son los vértices que tenemos que probar.

Dentro de este bucle que tendrá un gran rango ya que tenemos que probar con todos los posibles caminos, ahora solo nos falta saber si ese camino es una solución o no, para esto solo tenemos que saber que vértices de la matriz de adyacencia son adyacentes a la lista de vectores que conseguimos en el paso anterior. Por lo que, para saber que vértices son adyacentes haremos una multiplicación entre esta lista y la matriz de adyacencia (ambas matrices son booleanas). Después, tendremos que iterar en la matriz resultante de esa multiplicación y asegurarnos de que esta no contenga ningún 0, si la matriz devuelve todos sus elementos como mayores a 0 entonces significa que nuestra lista de vértices cubre a todos los demás vértices por lo que tenemos un conjunto dominante.

Por ultimo, ahora solo debemos identificar el conjunto dominante mínimo para resolver el problema del conjunto dominante, para esto solo tenemos que almacenar el valor mínimo y sobrescribir este valor si encontramos uno que sea menor, y con esto podemos concluir la explicación del código de fuerza bruta.

4.2. Código de fuerza bruta

Listing 1: Código de fuerza bruta.

```
#include <iostream>
#include <fstream>
using namespace std;

/*Multiplicamos array[n] por la matriz de adyacencia[n][n]
guardamos los resultados en result[n].*/
void multiply(int* res, const int* array, const int**
    adyacenciaArray, const int n) {
     for (int i = 0; i < n; i++) {
         res[i] = 0;
```

```
        for (int j = 0; j < n; j++) {
            res[i] = res[i] + (array[j] * adyacenciaArray[j][i]);
        }
    }
}


bool isDominatingSet(const long x, const int n, const int**
    adyacenciaArray) {
    int *Conjunto_v= new int[n];
    int *Vertices_cubiertos= new int[n];
    long Bit = 1;
    //Calculamos que vertices pertenecen al valor x.
    for (int i = 0; i < n; i++) {
        if ((x & Bit) != 0) {
            Conjunto_v[i] = 1;
        }
        else {
            Conjunto_v[i] = 0;
        }
        Bit = Bit * 2;
    }
    /*Hacemos una multiplicacion entre el array y matriz de
        adyacencia
    de esta manera sacamos los vertices que estan cubiertos.*/
    multiply(Vertices_cubiertos, Conjunto_v, adyacenciaArray, n);

    /*Ahora comprobamos que todos los vertices esten en el array
    si no lo estan, no tenemos un conjunto dominante.*/
    for (int i = 0; i < n; i++) {
        if (Vertices_cubiertos[i] == 0) {
            return false;
        }
    }
    return true;
}

//Devuelve el numero de bits que contienen 1 en el valor x.
int Bitsen1(long x, int n) {
    long Bit = 1;
    int res = 0;
    //Calculamos que vertices pertenecen al valor x.
    for (int i = 0; i < n; i++) {
        if ((x & Bit) != 0) {
            res++;
        }
        Bit = Bit * 2;
    }
    return res;
```

```
}

void print(long x, int n) {
    long mask = 1;
    //Calculamos que vertices pertenecen al valor x.
    cout << "{";
    for (int i = 0; i < n; i++) {
        if ((x & mask) != 0) {
            cout << i << "␣";
        }
        mask = mask * 2;
    }
    cout << "}" << endl;
}

int main() {
    /* Leemos un txt en el que tenemos nuestra
    matriz de adyacencia */
    int n;
    std::ifstream O;
    O.open("C:/Users/sebpost/Documents/ADA/NPC/test.txt");
    if (!O.is_open()) {
        std::cout << "No␣se␣puedo␣abrir␣el␣archivo";
    }
    O >> n;
    int** adjacencyMatrix;
    adjacencyMatrix = new int* [n];
    for (int i = 0; i < n; i++) {
        adjacencyMatrix[i] = new int[n];
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            O >> adjacencyMatrix[i][j];
        }
    }
    // Imprimimos nuestra matriz de adyacencia

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << (adjacencyMatrix[i][j]) << "␣";
        }
        cout << endl;
    }

    // sacamos una potencia de 2
    long pot_2 = 1;
    for (int i = 0; i < n; i++) {
        pot_2 = pot_2 * 2;
```

```
    }

    int Nbits = n;
    int res = pot_2 - 1;
    //este sera el bucle que tomara mucho tiempo
    for (long i = 1; i < pot_2; i++) {

        if (isDominatingSet(i, n, (const int**)adjacencyMatrix)) {
            {
                if (Bitsen1(i, n) < Nbits) {
                    Nbits = Bitsen1(i, n);
                    res = i;
                }
            }
        }
    }
    cout << "El conjunto dominante minimo es: ";
    print(res, n);
    cout << endl;
}
```

5. Heurísticas Empleadas

En esta sección explicaremos las diferentes heurísticas y meta-heurísticas que utilizamos en el problema del conjunto dominante.

5.1. Tabu Search

Por mucho tiempo se ha intentado resolver problemas de optimización combinacional, en particular el problema del agente viajero, se han creado muchos procedimientos heurísticos para resolver este pero tabu search en particular se diferencia de los demás por su **generalidad** ya que no solo es capaz de resolver un problema especifico. Sino, es capaz de resolver muchísimos tipos de problemas.

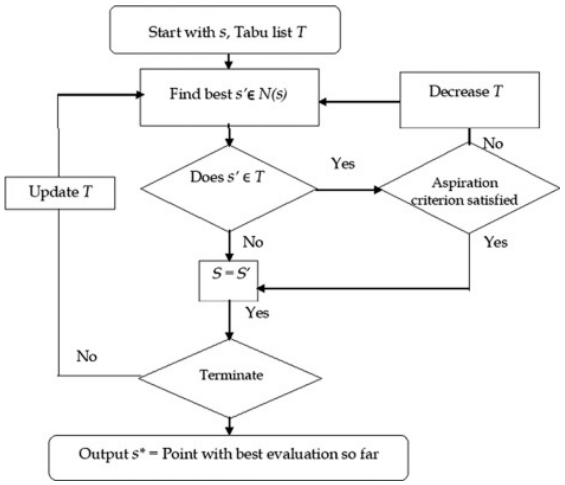
El método o estrategia de búsqueda *tabu search* es una meta-heurística, en otras palabras, es un método heurístico en el que se puede usar otros procedimientos heurísticos para conseguir mejores resultados, es una optimización para mejorar los métodos de búsqueda locales. Para poder realizar esta optimización *tabu search* introduce un tipo de memoria para guiar a el proceso, por lo que podemos ver esto como una búsqueda inteligente.
tabu search [3] esta hecho para trabajar con problemas de optimización combinacional de la siguiente manera:

$\min\{f(s)|s\epsilon X\}$ donde X es un conjunto de soluciones y f es una función de costo

Tabu search es un algoritmo iterativo que empieza desde la primera solución posible **s** con el objetivo de encontrar una solución **s*** que minimice **f** al menos aproximada-

mente. Para hacer esto se crea un conjunto de modificaciones posibles **M**, y **s'** que sera un **s** con una modificación **m** ($m\epsilon M$) y **N(s)** que sera un conjunto de soluciones que se pueden alcanzar de **s** si vemos esto como si fuera un grafo donde los vértices son **s** y **s'** estos dos estarían conectados y $s'\epsilon N(s)$.

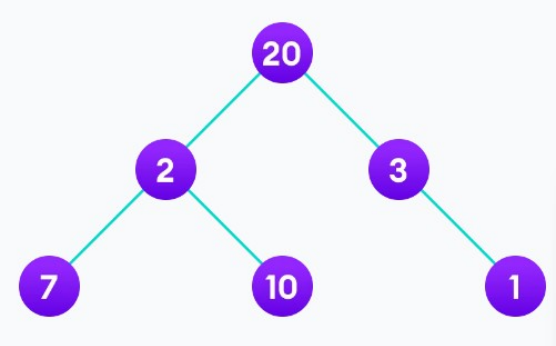
Por lo que, [3] en cada iteración estamos yendo de s a s' el problema con esto es que podemos movernos de s a una peor solución s', para prohibir estos movimientos el TS crea una lista de los movimientos prohibidos para llegar a una solución s* eficientemente.



5.2. Algoritmo Codicioso - Greedy Algorithm

Los algoritmos Greedy son aquellos que permiten resolver un problema de optimización al dividir el problema en sub-problemas que sigan la estructura del original, es decir que sean problemas que sigan buscando la solución al problema general.[7] De este modo se tendrá una solución óptima para cada sub-problema, bajo un criterio y la información disponible en ese momento sin preocuparse si el mejor resultado actual traerá el resultado óptimo general[5].

En la siguiente imagen se presenta el problema de encontrar el camino más largo del árbol:

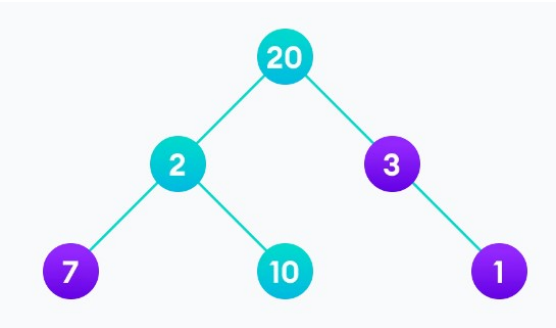


De acuerdo con la definición de los algoritmos Greedy, se buscará la mejor solución para el primer paso, es decir, ir desde la raíz (20), al hijo más pesado en ese nivel (3), y finalmente al siguiente nodo hijo (1), dando como resultado el siguiente resultado:

$$20 + 3 + 1 = 24$$

Sin embargo, entendemos que no es la solución correcta, pues existe otro recorrido que cumple lo solicitado óptimamente:

$$20 + 2 + 10 = 32$$



Esto implica que el algoritmo nunca revierte la decisión anteriormente tomada, incluso si la elección es incorrecta. Funciona en un enfoque de arriba hacia abajo (top-down)[4]. Así, al finalizar cada sub-solución, se tendrá un conjunto de soluciones que resultarán en la posible solución óptima del problema general, porque siempre se buscará la mejor opción local, y así, intentar producir el mejor resultado global.

5.3. Explicación de código

Primero, nuestro código leerá e interpretara archivos.txt, estos archivos tendrán los vértices y aristas del grafo en el que queremos hallar el conjunto dominante. Segundo, agregamos los vértices del grafo a un vector de vértices para tener su id ,grado y un booleano para saber si esta dominado o no y crearemos un entero x para ir almacenando el vértice de mayor grado (si dos o mas vértices comparten el mayor grado, se escogerá uno al azar). Tercero, entramos en un bucle y no saldremos de este hasta que todos los pesos de todos los vértices del grafo sean 0, para llegar a esto usaremos x (que almacena el vértice de mayor grado) para bajarle el peso a sus vértices de adyacencia por uno. Por ejemplo, si el vértice v_1 es adyacente a los vértices v_2 y v_3 que tienen los grados 1 y 2 respectivamente, si el vértice de mayor grado ya esta dominado(el booleano nos dirá eso) entonces sus vértices de adyacencia también estarán dominados y su peso se reducirá en uno.Por ultimo, al final del bucle los vértices tendrán los grados 0 y 1 respectivamente. Seguimos iterando hasta que todos los vértices tengan peso 0 y tendremos nuestro conjunto mínimo dominante.

5.4. Código de heurística

Listing 2: Código main.cpp.

```
#include <iostream>
#include <fstream>
#include "Graph.h"
#include "Algorithms.h"

using namespace std;

int main() {

    grafo* Grafo;
    ifstream O;
    O.open("C:/Users/sebpost/Documents/ADA/NPC/g-sample.txt");

    Grafo = Grafo->creatGraph(O);
    conjunto_vertices solucion = heuristica_greedy(Grafo);

    cout << "Solucion:␣" << "\t" << solucion.numero_dominante() <<
       (solucion.es_conjuntoDominante() ? "␣Conjunto␣dominante" : "
       ␣No␣hay␣conjunto␣dominante") << endl;

    delete(Grafo);
    O.close();
```

```
    return 0;
}
```

Listing 3: Código algoritmos.h.

```
#include "Algorithms.h"

struct vertice {
    int id;
    int peso;
    bool dominado;

    //Constructor con solo el id
    vertice(int id) : id(id) {}
    //Constructor con el id, peso y booleano dominado.
    vertice(int id, int w, bool cvd) : id(id), peso(w), dominado(
       cvd) {}

    bool operator>(const struct vertice v) const {
        return peso > v.peso;
    }

    bool operator==(const vertice& p) const {
        return p.id == id;
    }
};

/*Esta funcion retorna el grado del vertice*/
int grado(grafo* grafo, int vertice) {
    int i = 0;

    for (auto it = grafo->begin(vertice); it != grafo->end(vertice)
       ; (it)++)
    {
        i++;
    }

    return i;
}

//escoge el vertice de mayor grado.
int escogerVertice(vector<vertice>& grados) {
    //ordenamos los vertices por su grado
    sort(grados.begin(), grados.end(), std::greater<vertice>());
    //si no tiene grado
    if ((grados[0]).peso == 0)
    {
        return -1;
    }
```

```
    return (grados[0]).id;
}

//
void ajustar_pesos(grafo* grafo, vector<vertice>& peso, int v) {

    //quitamos el peso del vertice i ya que este ahora estara
       siendo dominado.
    auto vertice_i = find(peso.begin(), peso.end(), vertice(v));
    if (vertice_i != peso.end()) {
        vertice_i->peso = 0;
    }

    //actualizamos los pesos de los vertices de adyacencia.
    for (auto j = grafo->begin(v); j != grafo->end(v); (j)++)
    {
        //recorremos los vertices de adyacencia del vertice i.
        auto vertice_j = find(peso.begin(), peso.end(), vertice((*j
           ).conseguir_destino()));
        if (vertice_j != peso.end())
        {
            if ((*vertice_j).peso > 0)
            {
                //si el vertice i no esta dominado bajamos el peso
                   de sus vertices adyacentes
                if (!(*vertice_i).dominado)
                {
                    (*vertice_j).peso--;
                }

                /*si el vertice j no esta dominado entonces
                   cambiamos su estado a dominado*/
                if (!(*vertice_j).dominado)
                {
                    (*vertice_j).dominado = true;
                    (*vertice_j).peso--;

                    for (auto k = grafo->begin(v); k != grafo->end(
                       v); (k)++)
                    {
                        /*Buscamos vertices conectados al vertice
                           destinatario y en caso se encuentre uno
                        reducimos el peso en 1*/
                        auto vertice_k = find(peso.begin(), peso.
                           end(), vertice((*k).conseguir_destino())
                           );
                        if (vertice_k != peso.end())
                        {
```

```
                            if ((*vertice_k).peso > 0)
                            {
                                (*vertice_k).peso--;
                            }
                        }
                    }
                }
            }
        }
    }
    //cambiamos el estado del vertice i a dominado.
    (*vertice_i).dominado = true;
}


conjunto_vertices heuristica_greedy(grafo* grafo) {

    conjunto_vertices aux(grafo);
    vector<vertice> pesos;

    /*en este bucle estamos agregando los vertices del grafo
    a un vector de vertices para tener su id ,grado y un booleano
    para saber si esta dominado*/
    for (int i = 0; i < grafo->get_num_v(); i++)
    {
        pesos.push_back(vertice(i, grado(grafo, i), false));
    }

    int nv = 0;

    /*vtx lo igualamos al vertice con mayor grado del vector
    pesos, el bucle termina cuando todos los vertices tengan peso
       0.*/
    while ((nv = escogerVertice(pesos)) != -1)
    {
        //en vertex tememos un vector que guarda los vertices
           dominantes.
        aux.verticesDominantes(nv);
        //modifica los pesos de todo el grafo luego de sacar el
           vertice dominante.
        ajustar_pesos(grafo, pesos, nv);
    }

    for (int x = 0; x < aux.ObtenerTamanoConjunto(); x++)
    {
        if (aux.getVertexSet()[x] == 1)
            cout << x << endl;
    }
```

```
    return aux;
}
```

Referencias

[1] Thomas H. Cormen y col. *Introduction to Algorithms*. 2nd. The MIT Press, 2001. ISBN: 0262032937.

[2] Irit Dinur y Samuel Safra. «On the hardness of approximating minimum vertex cover». En: *Annals of mathematics* (2005), págs. 439-485.

[3] C. N. Fiechter. *A parallel tabu search algorithm for large traveling salesman problems*. 1990. URL: https://core.ac.uk/download/pdf/82688573.pdf.

[4] *Greedy algorithm*. URL: https://www.programiz.com/dsa/greedy-algorithm#:~:text=A%20greedy%20algorithm%20is%20an,in%20a%20top%2Ddown%20approach..

[5] *Greedy algorithms*. URL: https://brilliant.org/wiki/greedy-algorithm/.

[6] Subhash Khot y Oded Regev. «Vertex cover might be hard to approximate to within 2- ε». En: *Journal of Computer and System Sciences* 74.3 (2008), págs. 335-349.

[7] Gerson Lázaro-Universidad Francisco de Paula Santander. *Algoritmos Greedy*. URL: https://www.youtube.com/watch?v=AtdFmnOf-7s.

[8] David Scot Taylor. *Algorithms with Attitude*. URL: https://www.youtube.com/watch?v=u5W32YxmnL8&ab_channel=AlgorithmswithAttitude.