# Algoritmos y Estructuras de Datos III
# TP3

26 de junio de 2015



| Integrante | LU | Correo electrónico |
|---|---|---|
| Martin Baigorria | 575/14 | martinbaigorria@gmail.com |
| Federico Beuter | 827/13 | federicobeuter@gmail.com |
| Juan Rinaudo | 864/13 | jangamesdev@gmail.com |
| Mauro Cherubini | 835/13 | cheru.mf@gmail.com |



**Reservado para la cátedra**

| Instancia | Docente | Nota |
|---|---|---|
| Primera entrega | | |
| Segunda entrega | | |

# Índice

# 1. Introducción

## 1.1. Definiciones

Antes de enunciar el problema a resolver en este trabajo practico, es necesario definir algunos conceptos.
Sea $G = (V, E)$ un grafo simple:

**Definición** Un conjunto $I \subseteq V$ es un *conjunto independiente* de $G$ si no existe ningún eje de $E$ entre los vértices de $I$. Es decir, los ejes de $I$ no están conectados por las aristas de $G$.

**Definición** Un conjunto $D \subseteq V$ es un *conjunto dominante* de G si todo vértice de $G$ esta en $D$ o bien tiene al menos un vecino que esta en $D$.

**Definición** Un conjunto *conjunto independiente dominante* de $G$ es un conjunto independiente que a su vez es dominante del grafo G. Desde un conjunto independiente dominante se puede acceder a cualquier vértice del grafo $G$ con solo recorrer una arista desde uno de sus vértices.

**Definición** Un *Conjunto Independiente Dominante Mínimo* (CIDM) es el conjunto independiente dominante de $G$ de mínima cardinalidad.

## 1.2. Introducción

En 1979, Garey y Johnson probaron que el problema de encontrar el CIDM de un grafo es un problema NP-Hard[1]. El objetivo del trabajo es utilizar diferentes técnicas algorítmicas para resolver este problema. En un principio diseñaremos e implementaremos un algoritmo exacto para el mismo. Dada la complejidad del problema, luego propondremos diferentes algoritmos heurísticos para llegar a una solución que sea lo suficientemente buena a fines prácticos en un tiempo razonable.

Si recordamos el problema 3 del TP1, podemos ver claramente que el mismo es un caso particular del problema del conjunto dominante mínimo. En este problema se imponía cierta estructura sobre el grafo en el que se efectuaba la búsqueda. El grafo en si no era completo, dado que cada casilla era representada por un nodo, y un caballo no podía acceder a los nodos adyacentes. El movimiento de los caballos se modelaba con aristas entre nodos. Este no es un caso del CIDM dado que la solución optima al problema (la menor cantidad de caballos para cubrir el tablero) no necesariamente era independiente. Por lo tanto, al buscar la solución estaríamos buscando el CDM del grafo.

## 1.3. Maximalidad y dominancia

Las siguientes proposiciones serán útiles a lo largo del trabajo:

**Proposición 1.1** *Sea $M$ un conjunto independiente maximal de $G$. $\forall v \in G.V$, si $v \notin M \implies \exists u \in M$ tal que $u$ es adyacente a $v$.*

**Demostración** Por absurdo. Sea M un conjunto independiente maximal y $v \notin G.V$. $\nexists u \in M$ tal que $u$ es adyacente a $v$. Por lo tanto, puedo agregar $v$ a $M$ y el conjunto va a seguir siendo independiente. Esto es absurdo, dado que el conjunto era maximal.

**Proposición 1.2** *Dado $G(V, E)$, todo conjunto independiente maximal es un conjunto independiente dominante.*

**Demostración** Sea $M$ un conjunto independiente maximal. Dado $v \in G.V$, por la propiedad anterior, si $v \notin M \implies \exists u \in M$ tal que $u$ es adyacente a $v$. Por lo tanto, si $v \notin M$ entonces $v$ tiene algún vecino que esta en $M$. Esto significa que $M$ es dominante.

[1]M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman and Company, San Francisco (1979).

## 1.4. Modelado

Muchos problemas se pueden modelar con grafos y se pueden resolver mediante la búsqueda del conjunto independiente dominante mínimo.

### 1.4.1. Planificador Urbano

Supongamos que un planificador urbano esta diseñando una ciudad con muchos barrios. Con el objetivo de proveer un buen sistema de salud para los habitantes, el planificador determina que cada barrio debe tener que cruzar a lo sumo un barrio para acceder a un hospital publico. Aquí podemos modelar a cada barrio con un vértice, y representar la adyacencia entre barrios con una arista. Al obtener el CIDM, obtenemos la ubicación y la mínima cantidad de hospitales públicos necesarios para cumplir con los objetivos del planificador.

# 2. Algoritmo Exacto

## 2.1. Algoritmo

Utilizando backtracking, recorremos todas los conjuntos dominantes independientes y luego seleccionamos el de menor cardinalidad. Representamos al grafo con un arreglo $graph[n]$ de nodos. Cada nodo tiene los siguientes atributos:

1. adj: Lista de nodos adyacentes al nodo actual.

2. degree: Grado del nodo actual.

3. added: Bool que indica si el nodo ha sido agregado al conjunto que representa el cubrimiento.

4. reachable: Bool que indica si el nodo actual puede ser alcanzado desde un nodo perteneciente al cubrimiento.

Comenzamos definiendo la función $backtracking$, que lo que hace es tomar un nodo del grafo, y luego considera los casos en los que el nodo pertenece o no a un posible cubrimiento. En caso de agregar el nodo al cubrimiento, todos los nodos adyacentes al mismo son ignorados en futuras llamadas recursivas. Si consideráramos los nodos adyacentes, romperíamos la independencia de los cubrimientos y ademas no solo incrementaría la complejidad del código sino que también el tiempo de ejecución del mismo.

## 2.2. Podas y estrategias

Para poder resolver el problema lo mas rápido posible, en primer lugar buscamos una forma rápida de verificar si un conjunto solución encontrado es independiente. En vez de tener que verificarlo, decidimos forzar la independencia por construcción. Esto se logro evitando los nodos adyacentes a los que ya agrego el algoritmo al potencial conjunto solución. De esta forma mantenemos la independencia del conjunto y evitamos tener que agregar innecesariamente muchos nodos.

Otro problema importante es verificar si los nodos seleccionados forman un cubrimiento. Esto lo resolvimos simplemente haciendo que la función backtracking lleve la cuenta del total de nodos alcanzables por el cubrimiento. Si ese numero es igual al numero total de nodos, significa que llegamos a un cubrimiento. De esta manera evitamos funciones auxiliares que tengan que verificar si los nodos seleccionados hasta ahora forman un cubrimiento, y a su vez sabemos que por construcción el mismo es independiente.

Ademas, antes de comenzar la búsqueda agregamos todos los vértices de $d(v) = 0$ al conjunto solución final. Esto se debe a que estos vértices necesariamente estarán en la solución. Es muy simple probar esto, dado que si no lo estuvieran, algún vértice adyacente debería estar en el conjunto para que lo cubra. Sin embargo, tal vértice no existe.

Una poda muy común que también hemos implementado es la de la solución local actual. Dada una solución posible (que aun no sabemos si es la mínima), si en el estado actual del algoritmo se esta considerando un numero de vértices que no le puede ganar a esta solución, ignoramos esa rama del árbol de estados posibles.

## 2.3. Complejidad

## 2.4. Complejidad Espacial

Para la representación del grafo, utilizamos un arreglo de nodos. Cada nodo tiene una lista de adyacencia. Por lo tanto, la complejidad espacial de nuestro algoritmo es de $\mathcal{O}(n + 2m)$, donde $n$ es la cantidad total de vértices y $m$ la cantidad total de aristas.

## 2.5. Complejidad Temporal

Nuestro algoritmo, sin considerar las podas, recorre cada conjunto independiente dominante una vez. Cada vez que encuentra uno, lo guarda en una estructura auxiliar en $\mathcal{O}(n)$. Si todos los nodos tienen grado 0, son agregados automaticamente, y el algoritmo resuelve el problema en $n$ iteraciones. En el peor de los casos, el algoritmo recorre todos los conjuntos independientes y dominantes, comenzando con el de mayor cardinalidad. Cada vez que lo encuentra, actualiza la estructura donde guardamos la solución. Para que esto suceda, en realidad todos los conjuntos dominantes deben tener diferente cardinalidad, cosa que en general no sucede. Como todo conjunto tiene $2^n$ subconjuntos, utilizaremos esto para acotar la cantidad de veces que actualiza la solución local. Seguramente hay una cota teórica mucho mejor.

Por otro lado, recorremos cada vértice y sus aristas adyacentes una vez por iteración. Aunque por construcción forzamos la independencia de los vértices, para poder acotar la complejidad supongamos que no ignora ninguna ramificación. Por lo tanto, la cantidad de nodos recorridos esta acotada por $2^n$. Esto significa que el algoritmo pertenece a $\mathcal{O}(n \times 2^n)$.

# 3. Heurística Constructiva Golosa

## 3.1. Algoritmos

Para poder armar una heurística golosa para el problema del CIDM, en primer lugar hay que buscar un buen criterio para seleccionar que nodos pertenecerán al cubrimiento, dado los nodos que ya han sido agregados.

### 3.1.1. Por grado

Al principio decidimos implementar esta heurística utilizando un heap, ordenando los nodos por su grado. Luego, desencolamos del heap y vamos actualizando los flags de cada nodo a medida que son alcanzables. El algoritmo tiene $\mathcal{O}(n \times log(n) + m)$.

### 3.1.2. Scoring

Aunque este método con el heap es rápido, en realidad podemos mejorar la forma en la que seleccionamos los vértices. Este método consiste en tomar el numero de nodos adyacentes efectivos (score) a los que cada nodo puede acceder. Definimos a un nodo adyacente efectivo como un nodo que es adyacente y a su vez no puede ser accedido por otros nodos que ya pertenecen a la solución parcial en construcción. De esta forma, este criterio también nos garantiza la independencia del conjunto, dado que si tomamos dos nodos de la solución, por construcción no pueden ser adyacentes.

Cada nodo va a tener como atributos su score, un flag que indica si ha sido agregado y otro que indica si es alcanzable por el cubrimiento parcial actual.

El algoritmo va a iterar un arreglo de nodos $n^2$ veces. Cada vez que busquemos un nodo para agregar al conjunto, los iteraremos todos para buscar el de máximo score. Al identificarlo, actualizaremos los scores de los nodos adyacentes a los adyacentes del mismo. A priori parece que la complejidad de este nuevo algoritmo se podría mejorar de forma significativa utilizando algún otro tipo de estructura de datos.

## 3.2. Complejidad

El primer algoritmo resuelve el problema en $\mathcal{O}(n \times log(n) + m)$ simplemente ignorando la actualización de los scores, desencolando de un heap $n$ veces. Sin embargo, este criterio es a simple vista inferior que el de actualización de scores. Aquí hay un tradeoff entre hacer la mejor elección y la complejidad temporal del algoritmo.

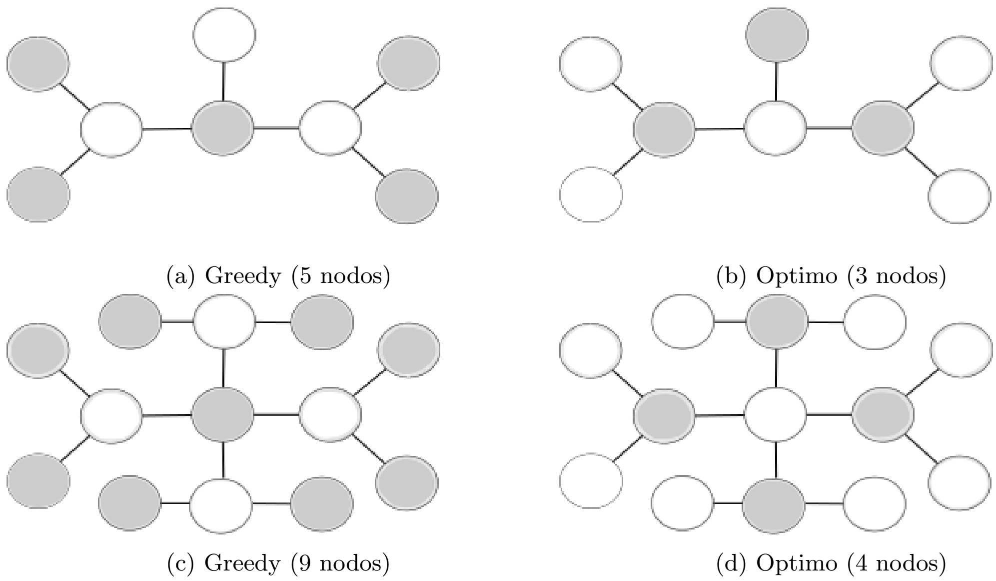
El algoritmo basado en el score recorre arreglo $n$ veces. A su vez, buscar los adyacentes de los adyacentes se hace $m$ veces. Luego actualizamos en total el score de $m$ nodos. Por lo tanto, el algoritmo tiene orden $\mathcal{O}(n^2 + 2 \times m)$.

Notar que la forma en que buscamos el máximo es sumamente ineficiente. Esto se debe a que si utilizamos sort, luego es bastante difícil encontrar el nodo al que le debemos actualizar su respectivo score. A su vez, dado que en cada iteración actualizamos el score, mantener el orden es sumamente costoso. Es muy posible que exista una estructura de datos mucho mas eficiente para resolver este problema (una especie de heap dinámico). Knuth [2] seguramente nos querría pegar.

[2] El rey de las estructuras de datos.

## 3.3. Efectividad de la heurística

Nuestra heurística no siempre devuelve la solución optima. Considerar los siguientes ejemplos:



(a) Greedy (5 nodos)

(b) Optimo (3 nodos)

(c) Greedy (9 nodos)

(d) Optimo (4 nodos)

Figura 1: Ejemplos de nuestra heurística comparado con el optimo.

El peor caso es claramente el de la figura (c) y (d). Tenemos un nodo $v$ de grado $d(v) = n$, con sus nodos adyacentes de grado $d(u) = n - 1$. Si tenemos $c$ componentes conexas de ese tipo, utilizaremos $c \times (n \times (n-2)+1)$ nodos, cuando en realidad el optimo tiene $c \times n$ nodos.

# 4. Heurística de Búsqueda Local

## 4.1. Algoritmo

Antes de explicar nuestro algoritmo, comenzemos definiendo que es una heurística de búsqueda local. Para cada solución factible $s \in S$, se define $N(s)$ como el conjunto de soluciones vecinas de $s$. Un procedimiento de búsqueda local toma una solución inicial $s$ e iterativamente la mejora reemplazándola por otra solución mejor del conjunto $N(s)$, hasta llegar a un optimo local. El algoritmo se puede ver con el siguiente pseudocodigo:

```
procedure localSearch(G)
    s ← getInitialSolution(G)
    bool localSolution ← true
    while localSolution do
        localSolution ← false
        for all ŝ ∈ N(s) do
            if |ŝ| < |s| then
                s ← ŝ
                localSolution ← true
                break
```

En primer lugar hay que pensar que algoritmo utilizar en la función $getInitialSolution(G)$. Para esto, utilizamos cualquiera de las heurísticas constructivas golosas del paso anterior.

Luego, debemos identificar como construiremos las diferentes $s \in N(s)$, es decir, como construiremos la función que nos devuelve los vecinos de una solución parcial $N(S)$.

## 4.2. Vecindades

Para este algoritmo, utilizaremos los siguientes dos criterios para definir la vecindad de una solución $s$:

1. Primera vecindad: Para la primera vecindad simplemente tomamos un vértice que actualmente no pertenece a la solución local. Luego, quitamos todos sus vértices adyacentes y verificamos si tenemos una solución con menor cardinal.

2. Segunda vecindad: Para este criterio, lo que hacemos es buscar dos nodos que no pertenecen a la solución local. Los agregamos, quitamos sus nodos adyacentes, y verificamos si el nuevo conjunto es un cubrimiento de menor cardinal.

## 4.3. Complejidad

### 4.3.1. Primera vecindad

En una iteración, el primer algoritmo de vecindad agrega un nodo y luego quita sus adyacentes. Por lo tanto, en el peor caso una iteración tiene orden $\mathcal{O}(n \times \Delta(G)^2)$. Esto se debe a que se debe verificar que todos los nodos adyacentes a los que saque son adyacentes a algún otro nodo del conjunto en $\mathcal{O}(\Delta(G))$ para cada nodo adyacente $(\Delta(G))$. [3]

### 4.3.2. Segunda vecindad

En el segundo caso, probamos agregando todos los pares de nodos a la solución actual, quitando sus nodos adyacentes y verificando si luego es una solución. Para ello, repetimos el procedimiento de la primera vecindad, aunque con una particularidad adicional: debemos verificar que al ver los adyacentes de los adyacentes, los mismos no correspondan a los vértices que quite al agregar el otro vértice.

Entonces, al agregar un vértice, quito todos sus vértices adyacentes (para mantener la independencia). Luego verifico que los adyacentes de los adyacentes son alcanzables. Esto ahora se hace $\Delta(G)$ veces en $\mathcal{O}(\Delta(G)^2)$. Esto se debe a que al verificar sus nodos adyacentes, debo ver que no estén en la lista de vértices adyacentes del otro nodo que también agregue.

Este procedimiento lo repetimos para todo par de $v \notin S$. Podemos acotar esto de forma grotesca por $\binom{n}{2}$. Por lo tanto la complejidad total de una iteración es de $\mathcal{O}(\binom{n}{2} \times \Delta(G)^3)$. Esto se podría mejorar optimizando la búsqueda en la lista de adyacencia de un vértice a $\mathcal{O}(\binom{n}{2} \times \Delta(G)^2 \times log(\Delta(G)))$.

[3] $\Delta(G)$ denota el máximo grado de un vértice perteneciente al grafo.

# 5. Metaheurística GRASP

## 5.1. Algoritmo

GRASP (Greedy Randomized Adaptative Search Procedure) es una combinación entre una heuristica golosa aleatorizada y un procedimiento de búsqueda local. La metaheurística se puede ver con el siguiente pseudocódigo:

```
procedure GRASP(G, k)
    G bestSolution
    while !terminationCondition(G, k, bestSolution) do
        s ← randomGreedyHeuristic(G, k)
        s ← localSearch(G,s)
        if |s| < |bestSolution| then
            bestSolution ← s
```

De este procedimiento surgen dos preguntas, que en realidad son cosas que debemos definir. De donde proviene la aleatoriedad de la heurística greedy? Cual es criterio de terminación que utilizaremos?

### 5.1.1. Random Greedy Heuristic

1. Por cantidad: Para agregarle una componente aleatoria a `GRASP`, se propone fabricar en cada paso de la heurística constructiva golosa una *Lista Restricta de Candidatos* (RCL) y elegir aleatoriamente un candidato de esta lista. Para ello, decidimos crear la función `greedyHeapConstructiveRandomized(Node graph[], int n, int k)` que lo que hace es ir eligiendo los $k$ vértices con mayor grado utilizando un heap como estructura auxiliar.

2. Por valor: Al igual que en el criterio anterior, elegimos un candidato aleatorio de una lista desde un heap. Sin embargo, ahora un vértice esta en la lista de candidatos si y solo si el grado de cualquier nodo en la lista esta a una distancia de $k$ grados del vértice de mayor grado.

### 5.1.2. Criterios de terminación

1. No se encontró ninguna mejora en las ultimas $j$ iteraciones.

2. Se alcanzo un limite prefijado de $j$ iteraciones.

# 6. Codigo

## 6.1. containers.h

```
#ifndef DATA_STRUCTURES_H_
#define DATA_STRUCTURES_H_

#include <forward_list>

using namespace std;

struct Node {
    int degree;
    int score;
    bool added;
    bool reachable;
    forward_list<int> adj;

    Node() {
        degree = 0;
        score = 0;
        added = false;
        reachable = false;
    }
};

struct _Pair {
    int score;
    int id;

    _Pair(int _score, int _id) {
        score = _score;
        id = _id;
    }

    bool operator <(const _Pair& x) {
        return this->score < x.score;
    }
};

#endif
```

### 6.2. backtracking.cpp

```cpp
#include <iostream>
#include <forward_list>
#include "../containers.h"

using namespace std;

void backtracking(int current, int& n, int coveredNodes, int usedNodes, Node graph[],
    bool localSolution[], int& nodesUsedInSolution);

int main() {

    int n, m; // n: vertices, m: edges
    cin >> n >> m;

    Node graph[n]; // graph container
    bool localSolution[n];

    int u, v;
    for (int i = 1; i <= m; ++i) { // (u,v) edges
        cin >> u >> v;
        u--; // nodes are counted from 0 in array.
        v--;
        graph[u].adj.push_front(v);
        graph[v].adj.push_front(u);

        graph[u].degree++;
        graph[v].degree++;
    }

    int initialNodes = 0;
    for (int i = 0; i < n; ++i) { // add d(v)=0 nodes to cover.
        if (graph[i].degree == 0) {
            graph[i].added = true;
            graph[i].reachable = true;
            localSolution[i] = true;
            initialNodes++;
        }
    }

    int nodesUsedInSolution = n; // worst case scenario is n, that way I avoid
        setting all the array as true.

    backtracking(0, n, initialNodes, initialNodes, graph, localSolution,
        nodesUsedInSolution);

    // display solution
    cout << nodesUsedInSolution;
    for (int i = 0; i < n; ++i) {
        if (localSolution[i] == true) cout << " " << i + 1;
    }
    cout << endl;

    return 0;
}

void backtracking(int current, int& n, int coveredNodes, int usedNodes, Node graph[],
    bool localSolution[], int& nodesUsedInSolution) {
```

```
    if (current == n) return; // no nodes left to add.
    if (graph[current].reachable == true) return backtracking(current + 1, n, coveredNodes, usedNodes, graph, localSolution, nodesUsedInSolution);
    if (usedNodes + 1 == nodesUsedInSolution) return; // cant beat current solution

    int pushed = 0;
    forward_list<int> added; // save changes to graph to then restore
    graph[current].added = true;

    for (auto it = graph[current].adj.begin(); it != graph[current].adj.end(); ++it) {
        int adjNode = *it;
        if (graph[adjNode].reachable == false) { // node reaches these new vertices
            graph[adjNode].reachable = true;
            added.push_front(adjNode);
            ++pushed;
        }
    }

    int tempCoveredNodes = coveredNodes + pushed + 1;
    if (tempCoveredNodes == n) { // coverage found
        for (int i = 0; i < n; ++i) {
            localSolution[i] = graph[i].added;
        }
        nodesUsedInSolution = ++usedNodes;
    } else {
        backtracking(current + 1, n, tempCoveredNodes, usedNodes + 1, graph, localSolution, nodesUsedInSolution); // adding current element to coverage
    }

    // restore graph state
    graph[current].added = false;
    for (auto it = added.begin(); it != added.end(); ++it) {
        graph[*it].reachable = false;
    }

    backtracking(current + 1, n, coveredNodes, usedNodes, graph, localSolution, nodesUsedInSolution); // skip current node

}
```

### 6.3. greedy.cpp

```cpp
#include <iostream>
#include <algorithm>
#include <stdlib.h>
#include "../containers.h"

using namespace std;

#define E_INVALID_PARAMETER 0

/* Gready Constructive Randomized Heuristic for MIDS
 * Using a heap, this function builds a MIDS by picking vertices
 * randomly from the top k vertices with the highest degree.
 *
 * @param graph[] Array of nodes.
 * @param n Size of graph.
 * @param k Parameter that indicates from how many nodes to
 *          pick randomly
 * @return Nodes used in solution set.
 */
int greedyHeapConstructiveRandomized(Node graph[], int n, int k) {

    if (k == 0) return E_INVALID_PARAMETER;

    vector<_Pair> currentPicks;
    vector<_Pair> heap;
    int nodesUsed = 0;

    for (int i = 0; i < n; i++) {
        if (graph[i].degree == 1) {
            graph[i].added = true;
            nodesUsed++;
        } else {
            graph[i].added = false;
            graph[i].reachable = false;
            heap.push_back(_Pair(graph[i].score, i));
        }
    }
    make_heap(heap.begin(), heap.end());

    int i = 0;
    while (i < k && i < (int) heap.size()) {
        _Pair p = heap.front();
        currentPicks.push_back(p);
        pop_heap(heap.begin(), heap.end());
        heap.pop_back();
        i++;
    }

    while (currentPicks.size() > 0) {
        int id = rand() % currentPicks.size();
        // cout << "picked id " << id << endl;
        _Pair p = currentPicks.at(id);
        currentPicks.erase(currentPicks.begin() + id);

        // cout << "node id " << p.id << endl;

        if (heap.size() > 0) {
```

```
            // cout << heap.size() << endl;
            _Pair p2 = heap.front();
            currentPicks.push_back(p2);
            pop_heap(heap.begin(), heap.end());
            heap.pop_back();
        }

        if (graph[p.id].reachable == true) continue;

        graph[p.id].added = true;
        nodesUsed++;

        for (auto it = graph[p.id].adj.begin(); it != graph[p.id].adj.end(); ++it) {
            int adjNode = *it;
            graph[adjNode].reachable = true;
        }

    }

    return nodesUsed;
}

/* Gready Constructive Randomized Heuristic for MIDS
 * Using a heap, this function builds a MIDS by picking vertices
 * randomly from the vertices that are k degrees away from the
 * available vertex with the highest degree.
 *
 * @param graph[] Array of nodes.
 * @param n Size of graph.
 * @param k Parameter that indicates from how many nodes to
 *          pick randomly
 * @return Nodes used in solution set.
 */
int greedyHeapConstructiveRandomized2(Node graph[], int n, int k) {

    if (k == 0) return E_INVALID_PARAMETER;

    vector<_Pair> currentPicks;
    vector<_Pair> heap;
    int nodesUsed = 0;

    for (int i = 0; i < n; i++) {
        if (graph[i].degree == 1) {
            graph[i].added = true;
            nodesUsed++;
        } else {
            graph[i].added = false;
            graph[i].reachable = false;
            heap.push_back(_Pair(graph[i].score, i));
        }
    }
    make_heap(heap.begin(), heap.end());

    int i = 0;
    int degree = heap.front().score;
    while(heap.front().score > degree - k && i < (int) heap.size()) {
        _Pair p = heap.front();
        currentPicks.push_back(p);
        pop_heap(heap.begin(), heap.end());
```

```cpp
        heap.pop_back();
        i++;
    }

    while (currentPicks.size() > 0) {
        int id = rand() % currentPicks.size();
        // cout << "picked id " << id << endl;
        _Pair p = currentPicks.at(id);
        currentPicks.erase(currentPicks.begin() + id);

        degree = currentPicks.at(0).score; // update degree

        // cout << "node id " << p.id << endl;

        if (heap.size() > 0 && heap.front().score > degree - k) {
            // cout << heap.size() << endl;
            _Pair p2 = heap.front();
            currentPicks.push_back(p2);
            pop_heap(heap.begin(), heap.end());
            heap.pop_back();
        }

        if (graph[p.id].reachable == true) continue;

        graph[p.id].added = true;
        nodesUsed++;

        for (auto it = graph[p.id].adj.begin(); it != graph[p.id].adj.end(); ++it) {
            int adjNode = *it;
            graph[adjNode].reachable = true;
        }

    }

    return nodesUsed;
}

/* Greedy Constructive Heuristic for MIDS
 * Using a heap, this function builds a MIDS by picking
 * the highest degree vertex repeatedly.
 *
 * @param graph[] Array of nodes.
 * @param n Size of graph.
 * @return Nodes used in solution set.
 */
int greedyHeapConstructive(Node graph[], int n) {

    vector<_Pair> heap;
    int nodesUsed = 0;

    for (int i = 0; i < n; i++) {
        if (graph[i].added == false)
            heap.push_back(_Pair(graph[i].score, i));
    }
    make_heap(heap.begin(), heap.end());

    for (int i = 0; i < n; i++) {
        _Pair p = heap.front();
        pop_heap(heap.begin(), heap.end());
```

```
        heap.pop_back();

        if (graph[p.id].reachable == true) continue;

        graph[p.id].added = true;
        nodesUsed++;

        for (auto it = graph[p.id].adj.begin(); it != graph[p.id].adj.end(); ++it) {
            int adjNode = *it;
            graph[adjNode].reachable = true;
        }
    }

    return nodesUsed;
}

/* Greedy Constructive Heuristic for MIDS
 * This function builds a MIDS by picking vertices by score.
 * The score is defined as the number of effective reachable
 * vertices given the vertices that have already been picked.
 *
 * @param graph[] Array of nodes.
 * @param n Size of graph.
 * @return Nodes used in solution set.
 */
int greedyConstructive(Node graph[], int n) {

    int nodesUsedInSolution = 0;

    for (int i = 0; i < n; ++i) {

        int greatest = 0;
        int score = 0;
        bool flag = false;

        // search for max score.
        for (int j = 0; j < n; ++j) {
            if (graph[j].reachable == true) continue;
            if (graph[j].score >= score) { // can be improved here!
                greatest = j;
                score    = graph[j].score;
                flag = true;
            }
        }

        if (!flag) break; // no more nodes to search.

        graph[greatest].added = true;
        graph[greatest].reachable = true;

        // update adyacent nodes of reachable nodes' scores.
        for (auto it = graph[greatest].adj.begin(); it != graph[greatest].adj.end(); ++it) {
            int adjNode = *it;
            graph[adjNode].reachable = true;
            for (auto it2 = graph[adjNode].adj.begin(); it2 != graph[adjNode].adj.end(); ++it2) {
                graph[*it2].score--;
            }
```

```
            }

            nodesUsedInSolution++;
        }

        return nodesUsedInSolution;
}
```

### 6.4. local.cpp

```cpp
#include <iostream>
#include <forward_list>
#include <algorithm>
#include <stdlib.h>
#include "local.h"

using namespace std;

int localSearch(Node graph[], int n, int nodesUsedInSolution) {

    int currentNodes = nodesUsedInSolution;

    for (int i = 0; i < n; ++i) {
        if (graph[i].added == true || graph[i].degree == 1) continue; // search for a node not in S.
        currentNodes++;

        bool reachable;

        for (auto it = graph[i].adj.begin(); it != graph[i].adj.end(); ++it) {

            if (graph[*it].added == false) continue; // we already know its reachable.

            reachable = false; // flag that indicates if all removed nodes are reachable.

            int adjNode = *it;
            currentNodes--;

            for (auto it2 = graph[adjNode].adj.begin(); it2 != graph[adjNode].adj.end(); ++it2) {
                if (adjNode == *it2) continue;
                if (graph[*it2].added == true) { // if the adj to the adj is added, can remove safely.
                    reachable = true;
                    break;
                }
            }
            if (reachable == false) break;
        }

        if (reachable == true && currentNodes < nodesUsedInSolution) { // build graph once we know we can improve it.
            graph[i].added = true;
            for (auto it = graph[i].adj.begin(); it != graph[i].adj.end(); ++it) {
                graph[*it].added = false;
            }
            nodesUsedInSolution = currentNodes;
            i = 0; // s <- s'
        } else {
            currentNodes = nodesUsedInSolution;
        }
    }

    return nodesUsedInSolution;
}
```

```
int localSearch2(Node graph[], int n, int nodesUsedInSolution) {

    int currentNodes = nodesUsedInSolution;

    for (int i = 0; i < n; ++i) {

        // find index of two nodes not in S.
        if (graph[i].added == true || graph[i].degree == 1) continue;

        int j;
        for (j = i + 1; j < n; j++) { // search for a second node
            if (graph[j].added == true || graph[j].degree == 1) continue;

            if (j == n) break; // no pair found

            // cout << "i: " << i << " j: " << j << endl;

            // check if S with these 2 nodes and without adj nodes is a 'better' cover.
            currentNodes = currentNodes + 2;
            bool reachable;

            for (auto it = graph[i].adj.begin(); it != graph[i].adj.end(); ++it) {

                reachable = false; // flag that indicates if all removed nodes are reachable.

                int adjNode = *it;

                if (graph[*it].added == true) currentNodes--;

                for (auto it2 = graph[adjNode].adj.begin(); it2 != graph[adjNode].adj.end(); ++it2) {
                    if (adjNode == *it2) continue;
                    if ((graph[*it2].added == true || *it2 == j) && !belongsTo(graph[j].adj, *it2)) { // if the adj to the adj is added, can remove safely.
                        reachable = true;
                        break;
                    }
                }
                if (reachable == false) break;
            }

            if (reachable == true) {

                for (auto it = graph[j].adj.begin(); it != graph[j].adj.end(); ++it) {

                    reachable = false; // flag that indicates if all removed nodes are reachable.

                    int adjNode = *it;

                    if (graph[*it].added == true && !belongsTo(graph[i].adj, *it)) currentNodes--;

```

```
                    for (auto it2 = graph[adjNode].adj.begin(); it2 != graph[adjNode].adj.end(); ++it2) {
                        if (adjNode == *it2) continue;
                        if ((graph[*it2].added == true || *it2 == i) && !belongsTo(graph[i].adj, *it2)) { // if the adj to the adj is added, can remove safely.
                            reachable = true;
                            break;
                        }
                    }
                    if (reachable == false) break;
                }
            }

            if (reachable == true && currentNodes < nodesUsedInSolution) { // build graph once we know we can improve it.
                // cout << "currentNodes: " << currentNodes << " nodesUsedInSolution: " << nodesUsedInSolution << endl;
                graph[i].added = true;
                graph[j].added = true;
                for (auto it = graph[i].adj.begin(); it != graph[i].adj.end(); ++it) {
                    graph[*it].added = false;
                }
                for (auto it = graph[j].adj.begin(); it != graph[j].adj.end(); ++it) {
                    graph[*it].added = false;
                }
                nodesUsedInSolution = currentNodes;
                i = 0; // s <- s'
            } else {
                currentNodes = nodesUsedInSolution;
            }
        }
    }

    return nodesUsedInSolution;
}

bool belongsTo(forward_list<int> adj, int x) {
    for (auto it = adj.begin(); it != adj.end(); ++it) {
        if (*it == x) return true;
    }
    return false;
}
```

### 6.5. grasp.cpp

```cpp
#include <iostream>
#include <forward_list>
#include <algorithm>
#include <stdlib.h>
#include "../greedy/greedy.h"
#include "../local/local.h"

using namespace std;

void displaySolution(Node graph[], int n, int nodesUsedInSolution);
int graspMIDSByIterations(Node graph[], int n, int j, int k, bool localSolution[]);
int graspMIDSByValue(Node graph[], int n, int j, int k, bool localSolution[]);

int main() {

    int n, m; // n: vertices, m: edges
    cin >> n >> m;

    Node graph[n]; // graph container
    bool localSolution[n];

    int u, v;
    for (int i = 1; i <= m; ++i) { // (u,v) edges
        cin >> u >> v;
        u--; // nodes are counted from 0 in array.
        v--;
        graph[u].adj.push_front(v);
        graph[v].adj.push_front(u);

        graph[u].degree++;
        graph[v].degree++;
    }

    int initialNodes = 0;
    for (int i = 0; i < n; ++i) { // add d(v)=0 nodes to cover.
        if (graph[i].degree == 0) {
            graph[i].added = true;
            graph[i].reachable = true;
            localSolution[i] = true;
            initialNodes++;
        }
    }

    int nodesUsedInSolution = graspMIDSByIterations(graph, n, 3, 3, localSolution);
    // int nodesUsedInSolution = graspMIDSByValue(graph, n, 3, 3, localSolution);

    // display solution
    cout << nodesUsedInSolution;
    for (int i = 0; i < n; ++i) {
        if (localSolution[i] == true) cout << " " << i + 1;
    }
    cout << endl;

    return 0;
}
```

```cpp
void displaySolution(Node graph[], int n, int nodesUsedInSolution) {
    cout << nodesUsedInSolution;
    for (int i = 0; i < n; ++i) {
        if (graph[i].added == true) cout << " " << i + 1;
    }
    cout << endl;
}

/* GRASP Heuristic
 * Minimum Independent Dominating Set
 * Stop criteria: Iterations
 * @param j Amount of attempts to improve solution.
 * @param k Parameter used for greedy heuristic.
 * @return Nodes used in solution set.
 */
int graspMIDSByIterations(Node graph[], int n, int j, int k, bool localSolution[]) {
    int currentBest = n + 1;
    while (j > 0) {
        int nodesUsed = greedyHeapConstructiveRandomized(graph, n, k);
        //int nodesUsed = greedyHeapConstructiveRandomized2(graph, n, k);
        nodesUsed = localSearch(graph, n, nodesUsed);
        // nodesUsed = localSearch2(graph, n, nodesUsed);

        if (nodesUsed < currentBest) { // save local solution
            for (int i = 0; i < n; ++i) {
                localSolution[i] = graph[i].added;
            }
            currentBest = nodesUsed;
        }

        j--;
    }
    return currentBest;
}

/* GRASP Heuristic
 * Minimum Independent Dominating Set
 * Stop criteria: Cycles without improvements
 * @param j Limit to cycles without improvements.
 * @param k Parameter used for greedy heuristic.
 * @return Nodes used in solution set.
 */
int graspMIDSByValue(Node graph[], int n, int j, int k, bool localSolution[]) {
    int currentBest = n + 1;
    int cycles = 0;
    while (cycles < j) {
        int nodesUsed = greedyHeapConstructiveRandomized(graph, n, k);
        //int nodesUsed = greedyHeapConstructiveRandomized2(graph, n, k);
        nodesUsed = localSearch(graph, n, nodesUsed);
        // nodesUsed = localSearch2(graph, n, nodesUsed);

        if (nodesUsed < currentBest) { // save local solution
            for (int i = 0; i < n; ++i) {
                localSolution[i] = graph[i].added;
            }
            currentBest = nodesUsed;
            cycles = 0;
        } else {
            cycles++;
```

```
            }
        }
        return currentBest;
}
```