

# Algoritmos y Estructuras de Datos III

## TP3

25 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

<b>1. Introducción</b>	<b>3</b>
1.1. Definiciones . . . . .	3
1.2. Introducción . . . . .	3
1.3. Maximalidad y dominancia . . . . .	3
1.4. Modelado . . . . .	4
1.4.1. Planificador Urbano . . . . .	4
<b>2. Algoritmo Exacto</b>	<b>5</b>
2.1. Algoritmo . . . . .	5
2.2. Podas y estrategias . . . . .	5
2.3. Complejidad . . . . .	5
2.4. Complejidad Espacial . . . . .	5
2.5. Complejidad Temporal . . . . .	5
<b>3. Heurística Constructiva Golosa</b>	<b>6</b>
3.1. Algoritmo . . . . .	6
3.2. Complejidad . . . . .	6
3.3. Efectividad de la heurística . . . . .	6
<b>4. Heurística de Búsqueda Local</b>	<b>7</b>
4.1. Algoritmo . . . . .	7
4.2. Vecindades . . . . .	7
4.3. Complejidad . . . . .	7
<b>5. Metaheurística GRASP</b>	<b>8</b>
5.1. Algoritmo . . . . .	8
5.1.1. Random Greedy Heuristic . . . . .	8
5.1.2. Criterio de terminación . . . . .	8
<b>6. Código</b>	<b>9</b>
6.1. data_structures.h . . . . .	9
6.2. backtracking.cpp . . . . .	9
6.3. greedy.cpp . . . . .	11
6.4. local.cpp . . . . .	13
6.5. grasp.cpp . . . . .	16

# 1. Introducción

## 1.1. Definiciones

Antes de enunciar el problema a resolver en este trabajo práctico, 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<sup>1</sup>. 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.

---

<sup>1</sup>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 todos 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 además 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 más 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 logra evitando los nodos adyacentes a los que ya agregamos al 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 número es igual al número 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.

Además, 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 está considerando un número 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 automáticamente, 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 está acotada por  $2^n$ . Esto significa que el algoritmo pertenece a  $\mathcal{O}(n \times 2^n)$ .

### 3. Heurística Constructiva Golosa

#### 3.1. Algoritmo

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.

Al principio decidimos implementar esta heurística utilizando un heap, ordenando los nodos por su grado. Sin embargo, aunque este método es rápido, encontramos un método de selección mejor. 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 al cubrimiento. 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  $\mathcal{O}(n \times \log(n))$  simplemente ignorando la actualización de los scores, desentolando 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 dinamico).

#### 3.3. Efectividad de la heurística

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

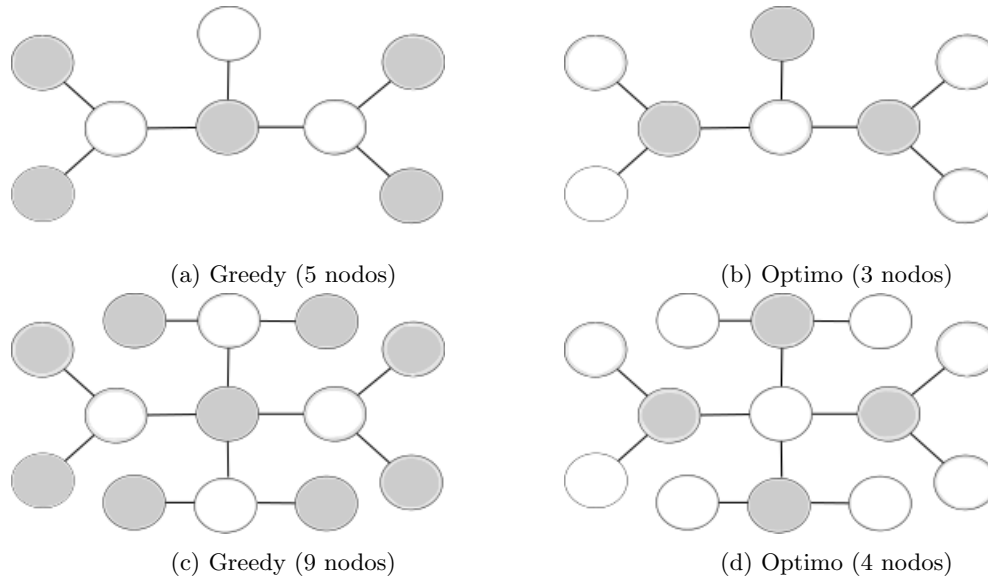


Figura 1: Ejemplos de nuestra heurística comparado con el óptimo.

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 óptimo tiene  $c \times n$  nodos.

## 4. Heurística de Búsqueda Local

### 4.1. Algoritmo

Antes de explicar nuestro algoritmo, comencemos 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 óptimo local. El algoritmo se puede ver con el siguiente pseudocódigo:

```
procedure LOCALSEARCH( $G$ )  
   $s \leftarrow \text{getInitialSolution}(G)$   
  bool localSolution  $\leftarrow$  true  
  while localSolution do  
     $\text{localSolution} \leftarrow \text{false}$   
    for all  $\hat{s} \in N(s)$  do  
      if  $|\hat{s}| < |s|$  then  
         $s \leftarrow \hat{s}$   
         $\text{localSolution} \leftarrow \text{true}$   
      break
```

En primer lugar hay que pensar que algoritmo utilizar en la función  $\text{getInitialSolution}(G)$ . Para esto, utilizamos la heurística constructiva golosa con score del paso anterior. Sin embargo también podemos utilizar la que se basa en un heap o se puede modificar backtracking para que tome la primera solución que encuentra.

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

En una iteración, nuestro algoritmo a lo sumo toma cada nodo del cubrimiento. Por lo tanto, en el peor caso una iteración tiene orden  $\mathcal{O}(n \times m^2)$ . Esto se debe a que se deben checkear todos los nodos adyacentes del que saque, y luego se debe verificar si ese nodo adicional tiene algún nodo adyacente perteneciente al cubrimiento.

**NOTA: Puedo reemplazar  $m$  por el máximo grado en el grafo.**

**Falta análisis de complejidad para el segundo criterio de vecindad.**

## 5. Metaheurística GRASP

### 5.1. Algoritmo

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

```
procedure GRASP( $G, k$ )  
   $G \leftarrow \text{bestSolution}$   
  while  $\neg \text{terminationCondition}(G, k, \text{bestSolution})$  do  
     $s \leftarrow \text{randomGreedyHeuristic}(G)$   
     $s \leftarrow \text{localSearch}(G, s)$   
    if  $|s| < |\text{bestSolution}|$  then  
       $\text{bestSolution} \leftarrow s$ 
```

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 Restrita 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: **TODO (programarlo también, es un toque)**

#### 5.1.2. Criterio de terminación

1. No se encontró ninguna mejora en las ultimas  $j$  iteraciones.
2. Se alcanzo un limite prefijado de  $j$  iteraciones.

**2x2x2, tenes 4 experimentaciones posibles combinando esto, x2 con las dos localidades definidas (hay que testear, mejorar y corregir eso)**



## 6. Codigo

### 6.1. data\_structures.h

---

```
1  #ifndef DATA_STRUCTURES_H
2  #define DATA_STRUCTURES_H
3
4  #include <forward_list>
5
6  using namespace std;
7
8  struct Node {
9      int degree;
10     int score;
11     bool added;
12     bool reachable;
13     forward_list<int> adj;
14
15     Node() {
16         degree = 0;
17         score = 0;
18         added = false;
19         reachable = false;
20     }
21 };
22
23 struct _Pair {
24     int score;
25     int id;
26
27     _Pair(int _score, int _id) {
28         score = _score;
29         id = _id;
30     }
31
32     bool operator <(const _Pair& x) {
33         return this->score < x.score;
34     }
35 };
36
37 #endif
```

---

## 6.2. backtracking.cpp

---

```
1 #include <iostream>
2 #include <forward_list>
3 #include "../data_structures.h"
4
5 using namespace std;
6
7 void backtracking(int current, int& n, int coveredNodes, int usedNodes, Node graph[],
8     bool localSolution[], int& nodesUsedInSolution);
9
10 int main() {
11     int n, m; // n: vertices, m: edges
12     cin >> n >> m;
13
14     Node graph[n]; // graph container
15     bool localSolution[n];
16
17     int u, v;
18     for (int i = 1; i <= m; ++i) { // (u,v) edges
19         cin >> u >> v;
20         u--; // nodes are counted from 0 in array.
21         v--;
22         graph[u].adj.push_front(v);
23         graph[v].adj.push_front(u);
24
25         graph[u].degree++;
26         graph[v].degree++;
27     }
28
29     int initialNodes = 0;
30     for (int i = 0; i < n; ++i) { // add d(v)=0 nodes to cover.
31         if (graph[i].degree == 0) {
32             graph[i].added = true;
33             graph[i].reachable = true;
34             localSolution[i] = true;
35             initialNodes++;
36         }
37     }
38
39     int nodesUsedInSolution = n; // worst case scenario is n, that way I avoid
40     // setting all the array as true.
41
42     backtracking(0, n, initialNodes, initialNodes, graph, localSolution,
43         nodesUsedInSolution);
44
45     // display solution
46     cout << nodesUsedInSolution;
47     for (int i = 0; i < n; ++i) {
48         if (localSolution[i] == true) cout << " " << i + 1;
49     }
50     cout << endl;
51
52     return 0;
53 }
54
55 void backtracking(int current, int& n, int coveredNodes, int usedNodes, Node graph[],
56     bool localSolution[], int& nodesUsedInSolution) {
```

```

54
55     if (current == n) return; // no nodes left to add.
56     if (graph[current].reachable == true) return backtracking(current + 1, n,
57         coveredNodes, usedNodes, graph, localSolution, nodesUsedInSolution);
58     if (usedNodes + 1 == nodesUsedInSolution) return; // cant beat current solution
59
60     int pushed = 0;
61     forward_list<int> added; // save changes to graph to then restore
62     graph[current].added = true;
63     for (auto it = graph[current].adj.begin(); it != graph[current].adj.end(); ++it)
64     {
65         int adjNode = *it;
66         if (graph[adjNode].reachable == false) { // node reaches these new vertices
67             graph[adjNode].reachable = true;
68             added.push_front(adjNode);
69             ++pushed;
70         }
71     }
72     int tempCoveredNodes = coveredNodes + pushed + 1;
73     if (tempCoveredNodes == n) { // coverage found
74         for (int i = 0; i < n; ++i) {
75             localSolution[i] = graph[i].added;
76         }
77         nodesUsedInSolution = ++usedNodes;
78     } else {
79         backtracking(current + 1, n, tempCoveredNodes, usedNodes + 1, graph,
80             localSolution, nodesUsedInSolution); // adding current element to coverage
81     }
82     // restore graph state
83     graph[current].added = false;
84     for (auto it = added.begin(); it != added.end(); ++it) {
85         graph[*it].reachable = false;
86     }
87
88     backtracking(current + 1, n, coveredNodes, usedNodes, graph, localSolution,
89         nodesUsedInSolution); // skip current node
90 }

```

---

### 6.3. greedy.cpp

---

```
1  #include <iostream>
2  #include <algorithm>
3  #include "../data_structures.h"
4
5  using namespace std;
6
7  int greedyHeapConstructiveRandomized(Node graph[], int n, int k) {
8
9      if (k == 0) {
10         return 0;
11     }
12
13     vector<_Pair> currentPicks;
14     vector<_Pair> heap;
15     int nodesUsed = 0;
16
17     for (int i = 0; i < n; i++) {
18         if (graph[i].degree == 1) {
19             graph[i].added = true;
20             nodesUsed++;
21         } else {
22             graph[i].added = false;
23             graph[i].reachable = false;
24             heap.push_back(_Pair(graph[i].score, i));
25         }
26     }
27     make_heap(heap.begin(), heap.end());
28
29     int i = 0;
30     while (i < k && i < (int) heap.size()) {
31         _Pair p = heap.front();
32         currentPicks.push_back(p);
33         pop_heap(heap.begin(), heap.end());
34         heap.pop_back();
35         i++;
36     }
37
38     while (currentPicks.size() > 0) {
39         int id = rand() % currentPicks.size();
40         // cout << "picked id " << id << endl;
41         _Pair p = currentPicks.at(id);
42         currentPicks.erase(currentPicks.begin() + id);
43
44         // cout << "node id " << p.id << endl;
45
46         if (heap.size() > 0) {
47             // cout << heap.size() << endl;
48             _Pair p2 = heap.front();
49             currentPicks.push_back(p2);
50             pop_heap(heap.begin(), heap.end());
51             heap.pop_back();
52         }
53
54         if (graph[p.id].reachable == true) continue;
55
56         graph[p.id].added = true;
57         nodesUsed++;
```

```

58
59         for (auto it = graph[p.id].adj.begin(); it != graph[p.id].adj.end(); ++it) {
60             int adjNode = *it;
61             graph[adjNode].reachable = true;
62         }
63     }
64 }
65
66     return nodesUsed;
67 }
68
69 int greedyHeapConstructive(Node graph[], int n) {
70
71     vector<_Pair> heap;
72     int nodesUsed = 0;
73
74     for (int i = 0; i < n; i++) {
75         if (graph[i].added == false)
76             heap.push_back(_Pair(graph[i].score, i));
77     }
78     make_heap(heap.begin(), heap.end());
79
80     for (int i = 0; i < n; i++) {
81         _Pair p = heap.front();
82         pop_heap(heap.begin(), heap.end());
83         heap.pop_back();
84
85         if (graph[p.id].reachable == true) continue;
86
87         graph[p.id].added = true;
88         nodesUsed++;
89
90         for (auto it = graph[p.id].adj.begin(); it != graph[p.id].adj.end(); ++it) {
91             int adjNode = *it;
92             graph[adjNode].reachable = true;
93         }
94     }
95
96     return nodesUsed;
97 }
98
99 /**
100 * This function can be improved by:
101 * 1. Using some sort of 'dynamic heap'.
102 * 2. Not iterating degree 0 nodes.
103 * 3. Using a list instead of an array, not to iterate
104 *    through nodes that are not necessary.
105 */
106 int greedyConstructive(Node graph[], int n) {
107
108     int nodesUsedInSolution = 0;
109
110     for (int i = 0; i < n; ++i) {
111
112         int greatest = 0;
113         int score = 0;
114         bool flag = false;
115
116         // search for max score.

```

```

117     for (int j = 0; j < n; ++j) {
118         if (graph[j].reachable == true) continue;
119         if (graph[j].score >= score) { // can be improved here!
120             greatest = j;
121             score     = graph[j].score;
122             flag = true;
123         }
124     }
125
126     if (!flag) break; // no more nodes to search.
127
128     graph[greatest].added = true;
129     graph[greatest].reachable = true;
130
131     // update adjacent nodes of reachable nodes' scores.
132     for (auto it = graph[greatest].adj.begin(); it != graph[greatest].adj.end();
133         ++it) {
134         int adjNode = *it;
135         graph[adjNode].reachable = true;
136         for (auto it2 = graph[adjNode].adj.begin(); it2 != graph[adjNode].adj.end();
137             ++it2) {
138             graph[*it2].score--;
139         }
140     }
141     nodesUsedInSolution++;
142 }
143 return nodesUsedInSolution;
144 }

```

---

## 6.4. local.cpp

---

```
1 #include <iostream>
2 #include <forward_list>
3 #include <algorithm>
4 #include <stdlib.h>
5 #include "local.h"
6
7 using namespace std;
8
9 int localSearch(Node graph[], int n, int nodesUsedInSolution) {
10
11     int currentNodes = nodesUsedInSolution;
12
13     for (int i = 0; i < n; ++i) {
14         if (graph[i].added == true || graph[i].degree == 1) continue; // search for a
15                                 // node not in S.
16         currentNodes++;
17
18         bool reachable;
19
20         for (auto it = graph[i].adj.begin(); it != graph[i].adj.end(); ++it) {
21             if (graph[*it].added == false) continue; // we already know its reachable
22             .
23             reachable = false; // flag that indicates if all removed nodes are
24                                 // reachable.
25
26             int adjNode = *it;
27             currentNodes--;
28
29             for (auto it2 = graph[adjNode].adj.begin(); it2 != graph[adjNode].adj.end
30                 (); ++it2) {
31                 if (adjNode == *it2) continue;
32                 if (graph[*it2].added == true) { // if the adj to the adj is added,
33                     can remove safely.
34                     reachable = true;
35                     break;
36                 }
37             }
38             if (reachable == false) break;
39         }
40
41         if (reachable == true && currentNodes < nodesUsedInSolution) { // build graph
42                                 // once we know we can improve it.
43             graph[i].added = true;
44             for (auto it = graph[i].adj.begin(); it != graph[i].adj.end(); ++it) {
45                 graph[*it].added = false;
46             }
47             nodesUsedInSolution = currentNodes;
48             i = 0; // s <- s'
49         } else {
50             currentNodes = nodesUsedInSolution;
51         }
52     }
53
54     return nodesUsedInSolution;
55 }
```

```

52
53 int localSearch2(Node graph[], int n, int nodesUsedInSolution) {
54
55     int currentNodes = nodesUsedInSolution;
56
57     for (int i = 0; i < n; ++i) {
58
59         // find index of two nodes not in S.
60         if (graph[i].added == true || graph[i].degree == 1) continue;
61
62         int j;
63         for (j = i + 1; j < n; j++) { // search for a second node
64             if (graph[j].added == true || graph[j].degree == 1) continue;
65
66             if (j == n) break; // no pair found
67
68             // cout << "i: " << i << " j: " << j << endl;
69
70             // check if S with these 2 nodes and without adj nodes is a 'better'
              cover.
71             currentNodes = currentNodes + 2;
72             bool reachable;
73
74             for (auto it = graph[i].adj.begin(); it != graph[i].adj.end(); ++it) {
75
76                 reachable = false; // flag that indicates if all removed nodes are
                  reachable.
77
78                 int adjNode = *it;
79
80                 if (graph[*it].added == true) currentNodes--;
81
82                 for (auto it2 = graph[adjNode].adj.begin(); it2 != graph[adjNode].adj
                  .end(); ++it2) {
83                     if (adjNode == *it2) continue;
84                     if ((graph[*it2].added == true || *it2 == j) && !belongsTo(graph[
                        j].adj, *it2)) { // if the adj to the adj is added, can remove
                          safely.
85                         reachable = true;
86                         break;
87                     }
88                 }
89                 if (reachable == false) break;
90             }
91
92             if (reachable == true) {
93
94                 for (auto it = graph[j].adj.begin(); it != graph[j].adj.end(); ++it)
                    {
95
96                     reachable = false; // flag that indicates if all removed nodes
                        are reachable.
97
98                     int adjNode = *it;
99
100                    if (graph[*it].added == true && !belongsTo(graph[i].adj, *it))
                        currentNodes--;
101

```



```

102         for (auto it2 = graph[adjNode].adj.begin(); it2 != graph[adjNode
103             ].adj.end(); ++it2) {
104             if (adjNode == *it2) continue;
105             if ((graph[*it2].added == true || *it2 == i) && !belongsTo(
106                 graph[i].adj, *it2)) { // if the adj to the adj is added,
107                 // can remove safely.
108                 reachable = true;
109                 break;
110             }
111         }
112         if (reachable == false) break;
113     }
114     if (reachable == true && currentNodes < nodesUsedInSolution) { // build
115         // graph once we know we can improve it.
116         // cout << "currentNodes: " << currentNodes << " nodesUsedInSolution:
117         // " << nodesUsedInSolution << endl;
118         graph[i].added = true;
119         graph[j].added = true;
120         for (auto it = graph[i].adj.begin(); it != graph[i].adj.end(); ++it)
121             {
122                 graph[*it].added = false;
123             }
124         for (auto it = graph[j].adj.begin(); it != graph[j].adj.end(); ++it)
125             {
126                 graph[*it].added = false;
127             }
128         nodesUsedInSolution = currentNodes;
129         i = 0; // s <- s'
130     } else {
131         currentNodes = nodesUsedInSolution;
132     }
133 }
134 return nodesUsedInSolution;
135 }
136
137 bool belongsTo(forward_list<int> adj, int x) {
138     for (auto it = adj.begin(); it != adj.end(); ++it) {
139         if (*it == x) return true;
140     }
141     return false;
142 }

```

---

## 6.5. grasp.cpp

---

```
1 #include <iostream>
2 #include <forward_list>
3 #include <algorithm>
4 #include <stdlib.h>
5 #include "../greedy/greedy.h"
6 #include "../local/local.h"
7
8 using namespace std;
9
10 void displaySolution(Node graph[], int n, int nodesUsedInSolution);
11 int graspMIDS(Node graph[], int n, int j, int k, bool localSolution[]);
12
13 int main() {
14
15     int n, m; // n: vertices, m: edges
16     cin >> n >> m;
17
18     Node graph[n]; // graph container
19     bool localSolution[n];
20
21     int u, v;
22     for (int i = 1; i <= m; ++i) { // (u,v) edges
23         cin >> u >> v;
24         u--; // nodes are counted from 0 in array.
25         v--;
26         graph[u].adj.push_front(v);
27         graph[v].adj.push_front(u);
28
29         graph[u].degree++;
30         graph[v].degree++;
31     }
32
33     int initialNodes = 0;
34     for (int i = 0; i < n; ++i) { // add d(v)=0 nodes to cover.
35         if (graph[i].degree == 0) {
36             graph[i].added = true;
37             graph[i].reachable = true;
38             localSolution[i] = true;
39             initialNodes++;
40         }
41     }
42
43     int nodesUsedInSolution = graspMIDS(graph, n, 3, 3, localSolution);
44
45     // display solution
46     cout << nodesUsedInSolution;
47     for (int i = 0; i < n; ++i) {
48         if (localSolution[i] == true) cout << " " << i + 1;
49     }
50     cout << endl;
51
52     return 0;
53 }
54
55 void displaySolution(Node graph[], int n, int nodesUsedInSolution) {
56     cout << nodesUsedInSolution;
```

```

58     for (int i = 0; i < n; ++i) {
59         if (graph[i].added == true) cout << " " << i + 1;
60     }
61     cout << endl;
62 }
63
64 /* GRASP Heuristic
65  * Minimum Independent Dominating Set
66  * @param j Amount of attempts to improve solution.
67  * @param k Parameter used for greedy heuristic.
68  */
69 int graspMIDS(Node graph[], int n, int j, int k, bool localSolution[]) {
70     int currentBest = n + 1;
71     while (j > 0) {
72         int nodesUsed = greedyHeapConstructiveRandomized(graph, n, k);
73         nodesUsed = localSearch(graph, n, nodesUsed);
74
75         if (nodesUsed < currentBest) { // save local solution
76             for (int i = 0; i < n; ++i) {
77                 localSolution[i] = graph[i].added;
78             }
79             currentBest = nodesUsed;
80         }
81
82         j--;
83     }
84     return currentBest;
85 }

```

---