

# Algoritmos y Estructuras de Datos III

## TP3

19 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
2.6. Experimentacion . . . . .	6
2.7. Código . . . . .	7
<b>3. Heurística Constructiva Golosa</b>	<b>9</b>
3.1. Algoritmo . . . . .	9
3.2. Complejidad . . . . .	9
3.3. Efectividad de la heurística . . . . .	9
3.4. Experimentacion . . . . .	10
3.5. Código . . . . .	11
<b>4. Heurística de Búsqueda Local</b>	<b>13</b>
4.1. Algoritmo . . . . .	13
4.2. Complejidad . . . . .	13
4.3. Experimentación . . . . .	14
<b>5. Metaheurística GRASP</b>	<b>15</b>
5.1. Algoritmo . . . . .	15
5.2. Complejidad . . . . .	15
5.3. Experimentación . . . . .	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  $G$  está en  $D$  o bien tiene al menos un vecino que está 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 sí 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 óptima al problema (cubrir el tablero con la menor cantidad de caballos posibles) 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 está 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 está 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 público. 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 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 mas rapido posible, en primer lugar buscamos una forma rapida de verificar si el conjunto actual al llamar la funcion recursiva es un cubrimiento. Esto lo resolvimos simplemente haciendo que la funcion 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.

Ademas, antes de comenzar la búsqueda agregamos a 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.

Una estrategia natural también fue evitar los nodos adyacentes a los que ya agrego el algoritmo al potencial conjunto solución sean considerados. De esta forma mantenemos la independencia del conjunto y evitamos tener que agregar innecesariamente muchos nodos. Ademas, evitamos tener que programar una función que verifique la independencia del conjunto.

### 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. Sin embargo, como base pensemos que si. Moon & Moser probaron que un grafo de  $n$  vértices tiene a lo sumo  $3^{n/3}$  conjuntos independientes y dominantes<sup>2</sup>. Por lo tanto, una cota no muy buena para la actualización de soluciones locales es  $\mathcal{O}(n \times 3^{n/3})$ .

Por otro lado, recorremos cada vértice y sus aristas adyacentes una vez por iteracion. Sin embargo hay muchas combinaciones en las que podemos hacer esto. En total, en el peor de los casos sin la poda del minimo, la cantidad de nodos recorridos esta acotada por  $2^n$ .

Por lo tanto, el orden del algoritmo es de  $\mathcal{O}(n \times 3^{n/3})$ .

---

<sup>2</sup>Moon, J. W.; Moser, Leo (1965), 'On cliques in graphs', Israel Journal of Mathematics 3 (1): 23–28, doi:10.1007/BF02760024

## 2.6. Experimentacion

## 2.7. Código

---

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

```

54     cout << nodesUsedInSolution;
55     for (int i = 0; i < n; ++i) {
56         if (localSolution[i] == true) cout << " " << i + 1;
57     }
58     cout << endl;
59
60     return 0;
61 }
62
63 void backtracking(int current, int& n, int coveredNodes, int usedNodes, Node graph[],
64     bool localSolution[], int& nodesUsedInSolution) {
65     if (graph[current].reachable == true) return backtracking(current + 1, n,
66         coveredNodes, usedNodes, graph, localSolution, nodesUsedInSolution);
67     if (current == n) return; // no nodes left to add.
68     if (usedNodes + 1 == nodesUsedInSolution) return; // cant beat current solution
69
70     int pushed = 0;
71     forward_list<int> added; // save changes to graph to then restore
72     graph[current].added = true;
73     for (auto it = graph[current].adj.begin(); it != graph[current].adj.end(); ++it)
74     {
75         int adjNode = *it;
76         if (graph[adjNode].reachable == false) { // node reaches these new vertices
77             graph[adjNode].reachable = true;
78             added.push_front(adjNode);
79             ++pushed;
80         }
81     }
82     int tempCoveredNodes = coveredNodes + pushed + 1;
83     if (tempCoveredNodes == n) { // coverage found
84         for (int i = 0; i < n; ++i) {
85             localSolution[i] = graph[i].added;
86         }
87         nodesUsedInSolution = ++usedNodes;
88     } else {
89         backtracking(current + 1, n, tempCoveredNodes, usedNodes + 1, graph,
90             localSolution, nodesUsedInSolution); // adding current element to coverage
91     }
92     // restore graph state
93     graph[current].added = false;
94     for (auto it = added.begin(); it != added.end(); ++it) {
95         graph[*it].reachable = false;
96     }
97
98     backtracking(current + 1, n, coveredNodes, usedNodes, graph, localSolution,
99         nodesUsedInSolution); // skip current node
100 }

```

---



### 3. Heurística Constructiva Golosa

#### 3.1. Algoritmo

Para poder armar un algoritmo para el problema del CIMD, en primer lugar hay que buscar un buen criterio para seleccionar que nodos pertenecerán al cubrimiento, dado los nodos que ya han sido agregados. Decidimos utilizar como criterio el numero de nodos adyacentes efectivos a los que cada nodo puede acceder. Definimos a un nodo adyacente efectivo (score) 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 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 maximo score. Luego, al identificarlo, actualizaremos los scores de los nodos adyacentes a los adyacentes del mismo.

#### 3.2. Complejidad

El algoritmo recorre cada nodo del arreglo  $n$  veces. A su vez, actualizar los scores al identificar un máximo se hace  $m$  veces. Por lo tanto, el algoritmo tiene orden  $\mathcal{O}(n^2 + m)$ .

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 iteracion 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.

Podríamos resolver el problema en  $\mathcal{O}(n \times \log(n))$  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 el mejor pick y la complejidad temporal del algoritmo.

#### 3.3. Efectividad de la heurística

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

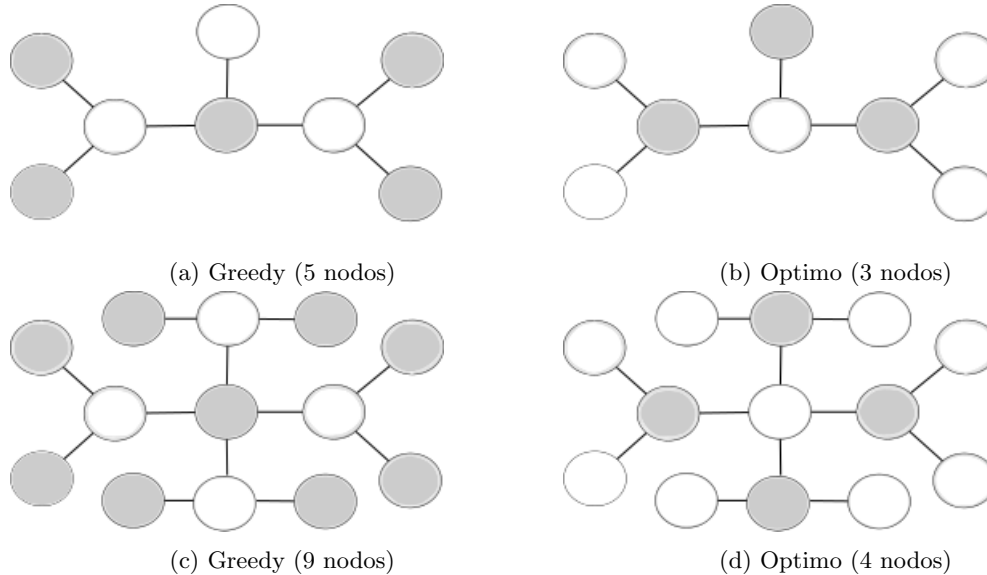


Figura 1: Ejemplos de nuestra heurística comparado con el óptimo.

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.

### 3.4. Experimentacion

### 3.5. Codigo

---

```
1 #include <iostream>
2 #include <forward_list>
3 #include <algorithm>
4
5 using namespace std;
6
7 struct Node {
8     forward_list<int> adj;
9     unsigned int score;
10    bool added;
11    bool reachable;
12
13    Node() {
14        score = 0;
15        added = false;
16        reachable = false;
17    }
18
19 };
20
21 int main() {
22
23     int n, m; // n: vertices , m: edges
24     cin >> n >> m;
25
26     Node graph[n]; // graph container
27     int nodesUsedInSolution = 0;
28
29     int u, v;
30     for (int i = 1; i <= m; ++i) { // (u,v) edges
31         cin >> u >> v;
32         u--; // nodes are counted from 0 in array.
33         v--;
34         graph[u].adj.push_front(v);
35         graph[v].adj.push_front(u);
36         graph[u].score++;
37         graph[v].score++;
38     }
39
40     /* This script can be improved by:
41     *     1. Using some sort of 'dynamic heap'.
42     *     2. Not iterating degree 0 nodes.
43     *     3. Using a list instead of an array, not to iterate
44     *         through nodes that are not necessary.
45     */
46
47     for (int i = 0; i < n; ++i) {
48
49         int greatest = 0;
50         unsigned int score = 0;
51         bool flag = false;
52
53         // search for max score.
54         for (int j = 0; j < n; ++j) {
55             if (graph[j].reachable == true) continue;
56             if (graph[j].score >= score) { // can be improved here!
```

```

57         greatest = j;
58         score     = graph[j].score;
59         flag = true;
60     }
61 }
62
63 if (!flag) break; // no more nodes to search.
64
65 graph[greatest].added = true;
66 graph[greatest].reachable = true;
67
68 // update adjacent nodes of reachable nodes scores.
69 for (auto it = graph[greatest].adj.begin(); it != graph[greatest].adj.end();
70      ++it) {
71     int adjNode = *it;
72     graph[adjNode].reachable = true;
73     for (auto it2 = graph[adjNode].adj.begin(); it2 != graph[adjNode].adj.end()
74          (); ++it2) {
75         graph[*it2].score--;
76     }
77 }
78
79 nodesUsedInSolution++;
80
81 // display solution
82 cout << nodesUsedInSolution;
83 for (int i = 0; i < n; ++i) {
84     if (graph[i].added == true) cout << " " << i + 1;
85 }
86 cout << endl;
87
88 return 0;
89 }

```

---

## 4. Heurística de Búsqueda Local

### 4.1. Algoritmo

El algoritmo propuesto sigue los siguientes pasos:

1. Utilizando la heurística constructiva golosa, armamos una potencial solución factible  $s \in S$ . También es posible tomar la primera solución encontrada por backtracking.
2. A continuación, construimos un posible vecino  $s' \in N(s)$ . Para ello, primero tomamos un nodo del cubrimiento, y lo sacamos.
3. Tomamos todos los nodos adyacentes a ese nodo, y los agregamos al conjunto solución.
4. Para mantener la independencia, quitamos todos los nodos adyacentes a los nuevos nodos.
5. A medida que recorremos los nodos, vamos marcando si los hemos manipulado durante el procedimiento.
6. Si la nueva solución  $s'$  es mejor, intercambiamos la solución anterior  $s$  por  $s'$  y repetimos el paso 2. Caso contrario, finaliza el algoritmo.

Notar que por cada componente conexa tenemos un solo vecino. La idea es ir probando con todas las componentes conexas. Si modificamos una vez un nodo, después cuando hagamos el proceso de tomar un nodo, lo tomamos de la lista de elementos que NO modificamos. Esto se ve claro en los ejemplos de la constructiva greedy.

### 4.2. Complejidad

### 4.3. Experimentación

## 5. Metaheurística GRASP

### 5.1. Algoritmo

### 5.2. Complejidad

### 5.3. Experimentación