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

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 $\mathcal{O}(n \times 2^n)$.

2.6. Codigo

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

```
55      backtracking(0, n, initialNodes, initialNodes, graph, localSolution,
            nodesUsedInSolution);
56
57      // display solution
58      cout << nodesUsedInSolution;
59      for (int i = 0; i < n; ++i) {
60          if (localSolution[i] == true) cout << " " << i + 1;
61      }
62      cout << endl;
63
64      return 0;
65  }
66
67  void backtracking(int current, int& n, int coveredNodes, int usedNodes, Node graph[],
        bool localSolution[], int& nodesUsedInSolution) {
68
69      if (current == n) return; // no nodes left to add.
70      if (graph[current].reachable == true) return backtracking(current + 1, n,
            coveredNodes, usedNodes, graph, localSolution, nodesUsedInSolution);
71      if (usedNodes + 1 == nodesUsedInSolution) return; // cant beat current solution
72
73      int pushed = 0;
74      forward_list<int> added; // save changes to graph to then restore
75      graph[current].added = true;
76
77      for (auto it = graph[current].adj.begin(); it != graph[current].adj.end(); ++it)
            {
78          int adjNode = *it;
79          if (graph[adjNode].reachable == false) { // node reaches these new vertices
80              graph[adjNode].reachable = true;
81              added.push_front(adjNode);
82              ++pushed;
83          }
84      }
85
86      int tempCoveredNodes = coveredNodes + pushed + 1;
87      if (tempCoveredNodes == n) { // coverage found
88          for (int i = 0; i < n; ++i) {
89              localSolution[i] = graph[i].added;
90          }
91          nodesUsedInSolution = ++usedNodes;
92      } else {
93          backtracking(current + 1, n, tempCoveredNodes, usedNodes + 1, graph,
                localSolution, nodesUsedInSolution); // adding current element to coverage
94      }
95
96      // restore graph state
97      graph[current].added = false;
98      for (auto it = added.begin(); it != added.end(); ++it) {
99          graph[*it].reachable = false;
100     }
101
102     backtracking(current + 1, n, coveredNodes, usedNodes, graph, localSolution,
            nodesUsedInSolution); // skip current node
103
104 }
```

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

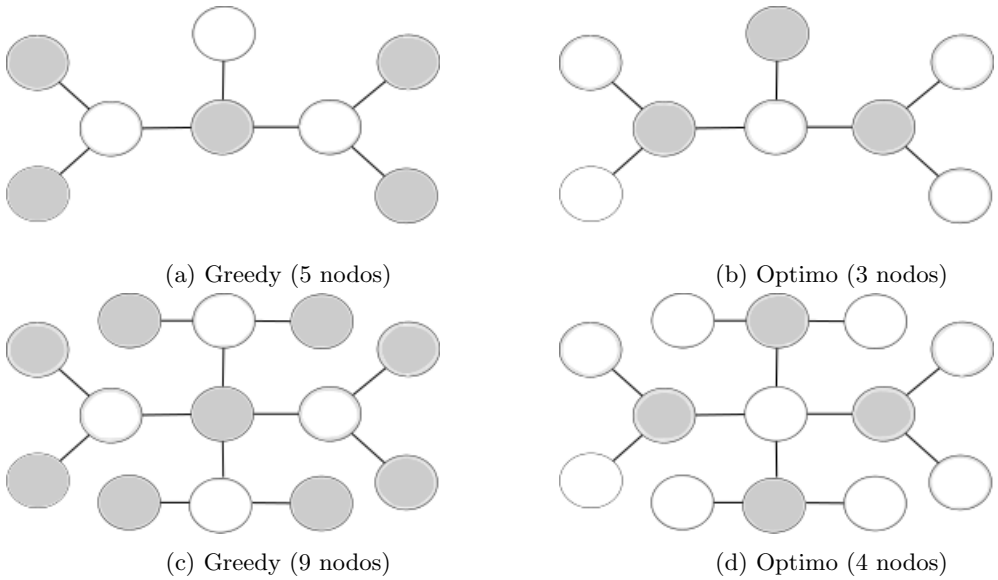
El primer algoritmo resuelve el problema en $\mathcal{O}(n \times log(n))$ simplemente ignorando la actualización de los scores, desencolando de un heap n veces. Sin embargo, este criterio es a simple vista inferior que el de actualización de scores. Aquí hay un tradeoff entre hacer la mejor elección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

3.3. Efectividad de la heurística

Nuestra heuristica 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 heuristica 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.

3.4. Codigo

```cpp
#include <iostream>
#include <forward_list>
#include <algorithm>

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;
    }

};

void displaySolution(Node graph[], int n, int nodesUsedInSolution);
int greedyConstructive(Node graph[], int n);
int greedyHeapConstructive(Node graph[], int n);

int main() {

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

    Node graph[n]; // graph container

    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++;
```

```
58          graph[u].score++;
59          graph[v].score++;
60      }
61
62      int initialNodes = 0;
63      for (int i = 0; i < n; ++i) { // add d(v)=0 nodes to cover.
64          if (graph[i].degree == 0) {
65              graph[i].added = true;
66              graph[i].reachable = true;
67              initialNodes++;
68          }
69      }
70
71      int nodesUsedInSolution = greedyConstructive(graph, n);
72      // int nodesUsedInSolution = greedyHeapConstructive(graph, n);
73
74      displaySolution(graph, n, nodesUsedInSolution + initialNodes);
75
76      return 0;
77  }
78
79  void displaySolution(Node graph[], int n, int nodesUsedInSolution) {
80      cout << nodesUsedInSolution;
81      for (int i = 0; i < n; ++i) {
82          if (graph[i].added == true) cout << " " << i + 1;
83      }
84      cout << endl;
85  }
86
87  int greedyHeapConstructive(Node graph[], int n) {
88
89      vector<_Pair> heap;
90      int nodesUsed = 0;
91
92      for (int i = 0; i < n; i++) {
93          if (graph[i].added == false)
94              heap.push_back(_Pair(graph[i].score, i));
95      }
96      make_heap(heap.begin(), heap.end());
97
98      for (int i = 0; i < n; i++) {
99          _Pair p = heap.front();
100         pop_heap(heap.begin(), heap.end());
101         heap.pop_back();
102
103         if (graph[p.id].reachable == true) continue;
104
105         graph[p.id].added = true;
106         nodesUsed++;
107
108         for (auto it = graph[p.id].adj.begin(); it != graph[p.id].adj.end(); ++it) {
109             int adjNode = *it;
110             graph[adjNode].reachable = true;
111         }
112     }
113
114     return nodesUsed;
115 }
116
```

```
117 /**
118 * This function can be improved by:
119 * 1. Using some sort of 'dynamic heap'.
120 * 2. Not iterating degree 0 nodes.
121 * 3. Using a list instead of an array, not to iterate
122 *    through nodes that are not necessary.
123 */
124 int greedyConstructive(Node graph[], int n) {
125
126     int nodesUsedInSolution = 0;
127
128     for (int i = 0; i < n; ++i) {
129
130         int greatest = 0;
131         int score = 0;
132         bool flag = false;
133
134         // search for max score.
135         for (int j = 0; j < n; ++j) {
136             if (graph[j].reachable == true) continue;
137             if (graph[j].score >= score) { // can be improved here!
138                 greatest = j;
139                 score    = graph[j].score;
140                 flag = true;
141             }
142         }
143
144         if (!flag) break; // no more nodes to search.
145
146         graph[greatest].added = true;
147         graph[greatest].reachable = true;
148
149         // update adyacent nodes of reachable nodes' scores.
150         for (auto it = graph[greatest].adj.begin(); it != graph[greatest].adj.end();
               ++it) {
151             int adjNode = *it;
152             graph[adjNode].reachable = true;
153             for (auto it2 = graph[adjNode].adj.begin(); it2 != graph[adjNode].adj.end
                   (); ++it2) {
154                 graph[*it2].score--;
155             }
156         }
157
158         nodesUsedInSolution++;
159     }
160
161     return nodesUsedInSolution;
162 }
```

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 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 algun nodo adyacente perteneciente al cubrimiento. **NOTA: Puedo reemplazar m por el máximo grado en el 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)
  while !*terminationCondition*() **do**
    s ← randomGreedyHeuristic(G)
    s ← localSearch(G,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

RCL (Restricted Candidate List)

5.1.2. Criterio de terminación