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

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. Sin embargo, asumir que si sirve para acotar la complejidad. Moon & Moser probaron que un grafo de n vértices tiene a lo sumo $3^{n/3}$ conjuntos independientes y dominantes [2]. 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 iteración. Aunque por construcción forzamos la independencia de los vertices, 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 $\mathcal{O}(n \times 3^{n/3})$.

[2] 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. Codigo

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

using namespace std;

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

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

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

    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 = 1; i <= m; ++i) { // add d(v)=0 nodes to cover.
        if (graph[i].degree == 0) {
            graph[i].added = true;
            graph[i].reachable = true;
            initialNodes++;
        }
    }

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

    // display solution
    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[],
        bool localSolution[], int& nodesUsedInSolution) {
64
65      if (graph[current].reachable == true) return backtracking(current + 1, n,
            coveredNodes, usedNodes, graph, localSolution, nodesUsedInSolution);
66      if (current == n) return; // no nodes left to add.
67      if (usedNodes + 1 == nodesUsedInSolution) return; // cant beat current solution
68
69      int pushed = 0;
70      forward_list<int> added; // save changes to graph to then restore
71      graph[current].added = true;
72
73      for (auto it = graph[current].adj.begin(); it != graph[current].adj.end(); ++it)
            {
74          int adjNode = *it;
75          if (graph[adjNode].reachable == false) { // node reaches these new vertices
76              graph[adjNode].reachable = true;
77              added.push_front(adjNode);
78              ++pushed;
79          }
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,
                localSolution, nodesUsedInSolution); // adding current element to coverage
90      }
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,
            nodesUsedInSolution); // skip current node
99
100 }
```

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. Decidimos utilizar como criterio 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 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.

3.2. Complejidad

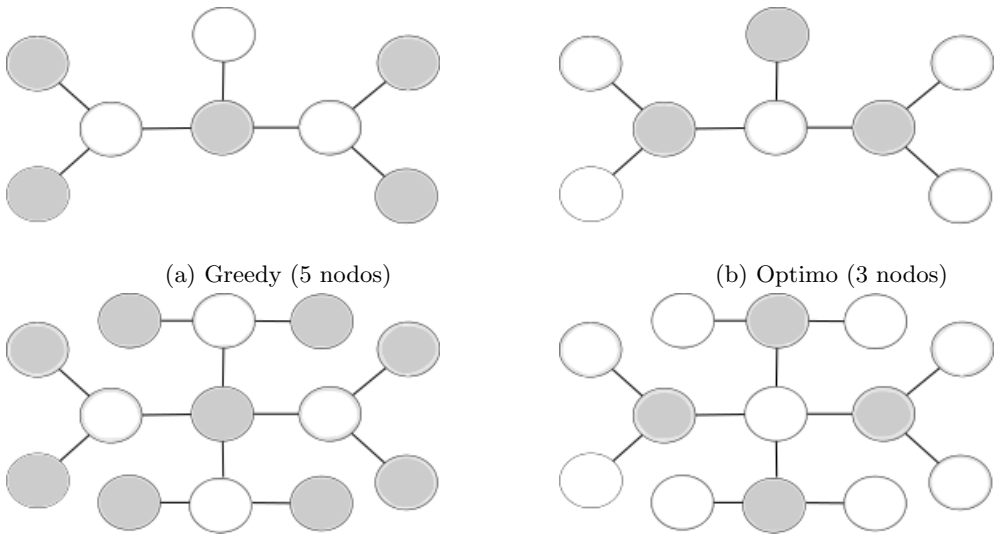
El algoritmo recorre cada nodo del arreglo n veces. A su vez, buscar los adyacentes de los adyacentes se hace m veces. Luego actualizamos en total el score de m nodos. Por lo tanto, el algoritmo tiene orden $\mathcal{O}(n^2 + 2 \times m)$.

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.

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 la mejor elección y la complejidad temporal del algoritmo.

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. Experimentacion

3.5. Codigo

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

using namespace std;

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

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

};

int main() {

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

    Node graph[n]; // graph container
    int nodesUsedInSolution = 0;

    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].score++;
        graph[v].score++;
    }

    /** This script can be improved by:
     *    1. Using some sort of 'dynamic heap'.
     *    2. Not iterating degree 0 nodes.
     *    3. Using a list instead of an array, not to iterate
     *       through nodes that are not necessary.
     */

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

        int greatest = 0;
        unsigned 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;
```

```
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 adyacent nodes of reachable nodes scores.
69          for (auto it = graph[greatest].adj.begin(); it != graph[greatest].adj.end();
               ++it) {
70              int adjNode = *it;
71              graph[adjNode].reachable = true;
72              for (auto it2 = graph[adjNode].adj.begin(); it2 != graph[adjNode].adj.end
                   (); ++it2) {
73                  graph[*it2].score--;
74              }
75          }
76
77          nodesUsedInSolution++;
78      }
79
80      // display solution
81      cout << nodesUsedInSolution;
82      for (int i = 0; i < n; ++i) {
83          if (graph[i].added == true) cout << " " << i + 1;
84      }
85      cout << endl;
86
87      return 0;
88  }
```

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 funcion $getInitialSolution(G)$. Para esto, utilizamos la heurística constructiva golosa 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 $N(S)$. Para ello, seguiremos el siguiente procedimiento:

1. Tomamos un nodo del conjunto solución actual y lo quitamos.
2. Tomamos todos los nodos adyacentes a ese nodo, y los agregamos al conjunto solución.
3. Para mantener la independencia, quitamos todos los nodos adyacentes a los nuevos nodos.
4. A medida que recorremos los nodos, vamos marcando si los hemos manipulado durante el procedimiento.

4.2. Complejidad

4.3. Experimentación

5. Metaheurística GRASP

5.1. Algoritmo

5.2. Complejidad

5.3. Experimentación