Facultad de Ingeniería
Universidad de Buenos Aires

Teoría de algoritmos

Trabajo Práctico N2

Autores:	*Padrón:*
Addin Kevin	94280
Cabrera Jorge	93310
Gatti Nicolas	93570
Orlando Juan Manuel	93152

18 de Noviembre, 2016

Contents

1 Programación dinámica

1.1 El problema de la mochila

1.1.1 Análisis de complejidad

El cálculo de la complejidad de este algoritmo es sencillo ya que lo que se realiza es iterar una matriz bidimensional, realizando en cada iteración una cantidad finita de operaciones $O(1)$. Luego para el obtener el listado de items a incluir en la mochila debemos realizar una iteración adicional $O(n)$.

Por lo tanto, la complejidad del algoritmo es

$$O(nW)$$

siendo n la cantidad de elementos en la mochila y W el peso máximo soportado por la mochila.

1.1.2 Análisis de tiempos

TODO :(

1.1.3 Implementación del algoritmo

TODO :(

1.2 El problema del viajante de comercio

1.2.1 Análisis de complejidad

En este algoritmo tenemos que analizar los distintos conjuntos de vértices posibles. Para cada conjunto de vértices posible, tenemos que determinar el óptimo de cada vértice para recorrer ese conjunto. Para determinar ese óptimo, tenemos que realizar un mínimo entre el resto de los vértices del conjunto, haciendo que la complejidad del algoritmo en tiempo sea:

$$O(n^2 2^n)$$

siendo n la cantidad de vértices.

En cuanto el espacio, hay que tener en cuenta que tenemos n vértices, y por cada vértice necesitamos guardar el resultado óptimo para los distintos conjuntos de vértices posibles. Por lo tanto, la complejidad en espacio del algoritmo es:

$$O(n 2^n)$$

siendo n la cantidad de vértices.

1.2.2 Análisis de tiempos

Para la medición de tiempos, correremos el archivo de prueba "p01.tsp"(renombrado como tsp1) y 7 archivos adicionales que contienen desde 15 vértices hasta 21 vértices inclusive, extraídos del archivo "att48_d.txt". La idea es poder ver como varía el tiempo con tan solo aumentar la entrada en una unidad.

prueba	**clocks**
tsp1	593626
n15	597700
n16	1480802
n17	3629355
n18	8816650
n19	21099357
n20	49919981
n21	117923147

Table 1: Tabla de tiempos

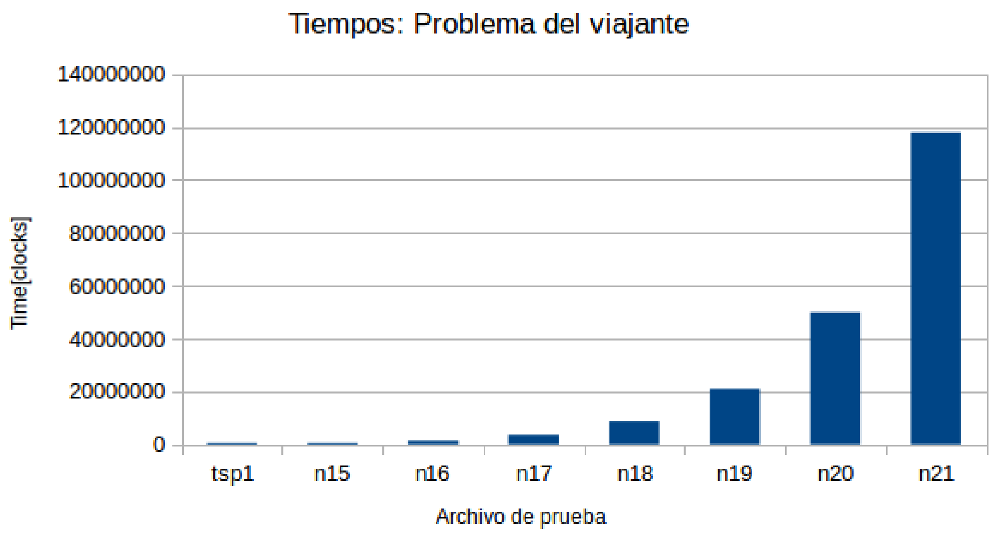

Figure 1: Gráfico de tiempos

Con el gráfico podemos observar que la ejecución del archivo de prueba "p01.tsp"(renombrado como tsp1) es casi instantánea.

También podemos apreciar que a medida que aumenta el número de nodos en tan solo una unidad, aumenta enormemente el tiempo insumido a más del doble.

En cuanto al archivo de prueba "fri26.tsp" de 26 nodos, no pudimos realizar la medición de tiempo debido a que el espacio requerido para la ejecución del algoritmo excedía nuestra memoria RAM disponible.

Si hacemos un poco de cuentas, mencionamos que la complejidad en espacio del algoritmo es $O(n2^n)$. Si $n = 26$, tenemos que $26 * 2^{26} = 1744830464$. Supongamos que requerimos 8B para almacenar el costo asociado a dicho vértice y conjunto, tendríamos: $26 * 2^{26} * 8B \approx 14GB$.

Nuestra implementación trato de optimizar el espacio lo más posible, utilizando para ello un entero para representar al conjunto de vértices, pero no pudo ser posible ejecutarlo.

1.2.3 Implementación del algoritmo

```
#include "TSPRecursive.h"

/
  vector<vector<CostInt>> matrix:
      Matriz de costos no negativos, se asume que es una matriz cuadrada N N, N = size(matrix)

  VertexInt initialVertex:
      Vertice inicial para la óejecucin del algoritmo

 /
TSPRecursive::TSPRecursive(vector<vector<CostInt>>  matrix, VertexInt initialVertex) {
    this->matrix = matrix;
    this->vertexCount = this->matrix->size();

    this->memory = new vector<map<SetInt, CostInt>>();

    this->path = new vector<map<SetInt, VertexInt>>();

    for (unsigned int i = 0; i < this->vertexCount; i++) {
        this->memory->push_back(map<SetInt,CostInt>());
        this->path->push_back(map<SetInt,VertexInt>());
    }

    this->initialVertex = initialVertex;
}

TSPRecursive::~TSPRecursive() {
    delete this->memory;
    delete this->path;
}


/
  Devuelve el vector de vertices que contiene el recorrido a seguir.
 /
vector<VertexInt>  TSPRecursive::createPathList(SetInt setNumber, VertexInt initialVertex) {
    vector<VertexInt> pathList = new vector<VertexInt>();
    pathList->push_back(initialVertex);
```

```
    VertexInt actualVertex = initialVertex;

    for (unsigned int i = 0 ; i < this->vertexCount - 1 ; i++) {
        VertexInt nextVertex = this->path->at(actualVertex).at(setNumber);
        pathList->push_back(nextVertex);
        setNumber = this->getSetNumberWithOutVertex(setNumber, nextVertex);
        actualVertex = nextVertex;
    }

    pathList->push_back(initialVertex);
    return pathList;
}

/
   Apaga el n-esimo bit, siendo n = vertexNumber, sobre el setNumber.
   óPrecondicin: el n-esimo bit áest encendido.
  /
SetInt TSPRecursive::getSetNumberWithOutVertex(SetInt setNumber, VertexInt vertexNumber) {
    return setNumber - (1<< vertexNumber);
}

/
   Lleva a cabo la óejecucin del algoritmo.
   Se obtiene un par pair<CostInt, vector<VertexInt>>
   que contiene el costo de recorrer todas las ciudades
   (suma de los pesos de las aristas recorridas)
   y un vector que contiene los vertices en orden para
   realizar el recorrido y alcanzar dicho costo.
  /
pair<CostInt, vector<VertexInt> > TSPRecursive::run() {
    SetInt setNumber = (1 << this->vertexCount) - 1;
    CostInt cost = this->_run(this->initialVertex, this->getSetNumberWithOutVertex(setNumber, initialVertex));
    vector<VertexInt> pathList = this->createPathList(this->getSetNumberWithOutVertex(setNumber, initialVertex), initialVertex);

    return pair<CostInt, vector<VertexInt>>(cost, pathList);
}

```

```
/
  óFuncin auxiliar que calcula la óversin recursiva
  del problema del viajante utilizando óprogramacin
  dinamica.
 /
CostInt TSPRecursive::_run(VertexInt vertex, SetInt setNumber) {

    if (setNumber == 0) {
        // Conjunto vacio. Retorno el costo de ir hacia el vertice inicial.
        return this->matrix->at(vertex).at(this->initialVertex);
    } else if (this->memory->at(vertex).count(setNumber)) {
        // Ya fue calculado
        return this->memory->at(vertex).at(setNumber);
    }

    CostInt minCost = (CostInt)(-1);
    VertexInt vertexMin;

    SetInt copySetNumber = setNumber;

    for (VertexInt u = 0; u < this->vertexCount; u++) {
        unsigned int n = copySetNumber & 1;
        if (n == 1) {
            // Proceso el vertex u que pertenece al set.
            CostInt result = this->matrix->at(vertex).at(u) +
                             this->_run(u, this->getSetNumberWithOutVertex(setNumber, u));

            if (result < minCost) {
                minCost = result;
                vertexMin = u;
            }
        }
        copySetNumber = copySetNumber >> 1; // setNumber /= 2;
    }

    this->memory->at(vertex)[setNumber] = minCost;
    this->path->at(vertex)[setNumber] = vertexMin;

    return minCost;

```

```
}
```

Listing 1: TSPRecursive.cpp

```
#ifndef TRABAJOPRACTICO2_TSPRECURSIVE_H
#define TRABAJOPRACTICO2_TSPRECURSIVE_H

#include "Types.h"
#include <vector>
#include <utility>
#include <map>

using namespace std;

class TSPRecursive {
private:
    vector<vector<CostInt>> matrix;
    VertexInt vertexCount;
    vector<map<SetInt, CostInt>> memory;
    vector<map<SetInt, VertexInt>> path;
    VertexInt initialVertex;

    SetInt getSetNumberWithOutVertex(SetInt setNumber, VertexInt vertexNumber);

    vector<VertexInt> createPathList(SetInt setNumber, VertexInt initialVertex);

    CostInt _run(VertexInt vertex, SetInt setNumber);

public:
    /
       vector<vector<CostInt>> matrix:
            Matriz de costos no negativos, se asume que es una matriz cuadrada N N, N = size(matrix)

       VertexInt initialVertex:
            Vertice inicial para la óejecucin del algoritmo

     /
    TSPRecursive(vector<vector<CostInt>> matrix, VertexInt initialVertex);
    ~TSPRecursive();

```

```
    /
      Lleva a cabo la óejecucin del algoritmo.
      Se obtiene un par pair<CostInt, vector<VertexInt>>
      que contiene el costo de recorrer todas las ciudades
      (suma de los pesos de las aristas recorridas)
      y un vector que contiene los vertices en orden para
      realizar el recorrido y alcanzar dicho costo.
     /
    pair<CostInt, vector<VertexInt>> run();

};


#endif //TRABAJOPRACTICO2_TSPRECURSIVE_H
```

Listing 2: TSPRecursive.h

```
#ifndef TRABAJOPRACTICO2_TSPTEST_H
#define TRABAJOPRACTICO2_TSPTEST_H

#include "Types.h"
#include "ParserTSPFile.h"
#include "TSPRecursive.h"
#include <vector>
#include <string>
#include <iostream>

using namespace std;

class TSPTest {
public:
    CostInt calculateCost(vector<vector<CostInt>> matrixCost, ↩
        vector<VertexInt> list) {
        CostInt cost = 0;
        for (unsigned int i = 0; i < list->size() - 1 ; i++) {
            VertexInt x = list->at(i);
            VertexInt y = list->at(i+1);
            cost += matrixCost->at(x).at(y);
        }

        return cost;
    }

    bool checkPathList(vector<VertexInt> expectedList, vector<↩
```

```
VertexInt> actualList) {
        if (expectedList->size() != actualList->size()) {
            return false;
        }

        // Puede que los recorridos éestn invertidos, pero valen igual en el caso de que la matriz sea simetrica.
        bool toReturn = true;
        for (unsigned int i = 0; i < expectedList->size(); i++) {
            if (expectedList->at(i) != actualList->at(i)) {
                toReturn = false;
                break;
            }
        }

        if (toReturn) return true;

        toReturn = true;
        unsigned int size = expectedList->size();
        for (unsigned int i = 0; i < expectedList->size(); i++) {
            if (expectedList->at(i) != actualList->at(size-i-1)) {
                toReturn = false;
                break;
            }
        }

        return toReturn;
    }

    void printRecorrido(string msg, vector<VertexInt> pathList) {
        std::cout << msg << std::endl;
        for (unsigned int i = 0; i < pathList->size(); i++) {
            VertexInt vertexInt = pathList->at(i);
            std::cout << int(vertexInt) << "\t";
        }
        std::cout << std::endl;
    }

    void runExample(string matrixFileName, string solutionFileName) {
```

```cpp
        ParserTSPFile parserTSPFile(matrixFileName, solutionFileName);
        vector<vector<CostInt>> matrix = parserTSPFile.getMatrix();
        vector<VertexInt> expectedList = parserTSPFile.getSolutionList();
        CostInt expectedCost = calculateCost(matrix, expectedList);

        TSPRecursive  tspRecursive = new TSPRecursive(matrix, 0);

        pair<CostInt, vector<VertexInt>> par = tspRecursive->run();

        if (expectedCost != par.first) {
            delete tspRecursive;
            delete matrix;
            delete expectedList;
            delete par.second;
            throw string("Error al coincidir los costos en " + matrixFileName + ". Se esperaba ") \
                + to_string(expectedCost) + string(" y se obtuvo: ") + to_string(par.first);
        }

        if (!checkPathList(expectedList, par.second)) {
            printRecorrido("Esperado", expectedList);
            printRecorrido("Obtenido", par.second);

            delete tspRecursive;
            delete matrix;
            delete expectedList;
            delete par.second;
            throw string("No hubo coincidencia en los recorridos.");
        }

        std::cout << "Ejecucion de " << matrixFileName << " exitosa." << std::endl;

        delete tspRecursive;
        delete matrix;
        delete expectedList;
        delete par.second;
```

```
    }

};


#endif //TRABAJOPRACTICO2_TSPTEST_H
```

Listing 3: TSPTest.h

```

#ifndef TRABAJOPRACTICO2_TYPES_H
#define TRABAJOPRACTICO2_TYPES_H

#include <stdint.h>

// La finalidad de estos tipos fue probar distintos tipos de ↩
   órepresentacin para ver si se lograba
// ejecutar el archivo de prueba de 26 nodos. Por ese motivo el↩
    VertexInt áest en 8bits(256valores posibles).
typedef uint8_t VertexInt;
typedef uint32_t CostInt;
typedef uint32_t SetInt;

#endif //TRABAJOPRACTICO2_TYPES_H
```

Listing 4: Types.h

2 Flujo de redes

2.1 Modelado del problema

Para el modelado del problema, supongamos que tenemos 4 proyectos, con ganancias de 10, 5, 4 y 3, y supongamos que tenemos 3 áreas con costos de 15, 8 y 4. Para realizar un proyecto dado requerimos contratar ciertas áreas, por ejemplo:

Proyecto	Áreas requeridas
1	1, 2
2	2,3
3	3
4	3

Table 2: Tabla de requisitos

Resolveremos este problema utilizando flujo de redes, basándonos en la versión del libro “Algorithm Design” [1].

Para ello lo que haremos será crear un grafo que contengan como vértices los proyectos, las áreas y dos vértices adicionales, s y t.

Conectaremos el vértice s con cada proyecto y conectaremos cada área con el vértice t. Luego, conectaremos cada proyecto con las áreas requeridas para su realización.

En cuanto a las capacidad de las aristas, las aristas que conecten el vértice s con los proyectos tendrán como capacidad la ganancia de realizar dicho proyecto. Para las aristas que conecten las áreas con el vértice t tendrán como capacidad el costo de contratar dicha área. Por ultimo, las aristas que conectan los proyectos con las áreas tendrán capacidad ”Infinita“, de modo que no constituyan una limitación para el algoritmo. Esta capacidad infinita basta que sea igual a $C + 1$, siendo $C = \sum g_i$, donde g_i es la ganancia del proyecto i.

El grafo nos quedaría de la siguiente manera:

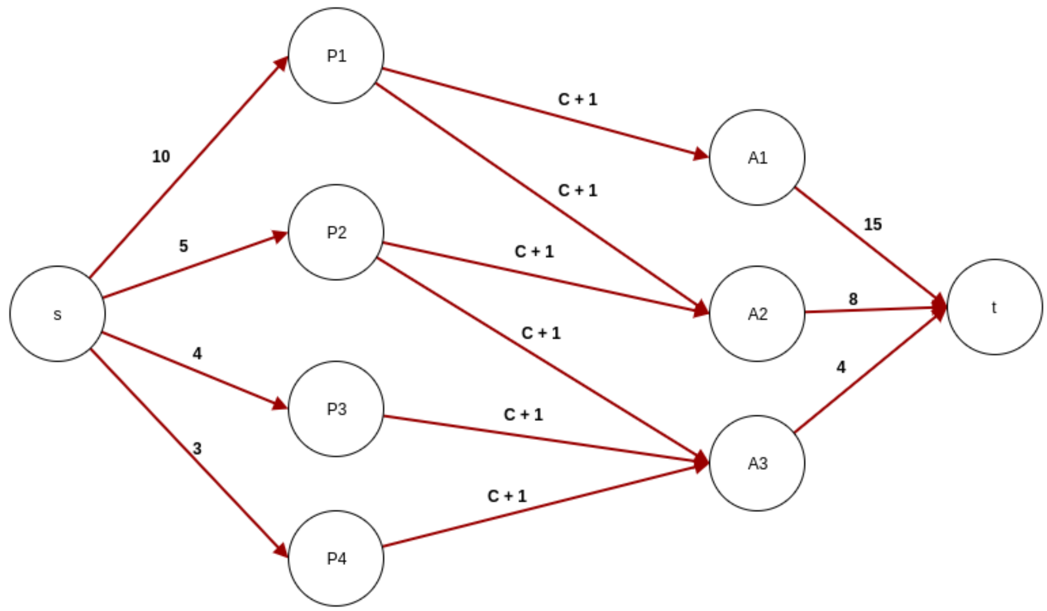

Figure 2: Grafo de ejemplo

Para obtener qué proyectos se realizarán y qué áreas se contratatarán se debe calcular el corte mínimo de este grafo, esto es equivalente a calcular el flujo máximo.

Sea A un conjunto de vértices tal que si existe un proyecto dentro del mismo, también están las áreas requeridas para dicho proyecto. Definimos $A' = A \cup \{s\}$, $B' = (V - A) \cup \{t\}$, donde V es el conjunto de vértices del grafo. Sea $c(A', B')$ la capacidad del corte (A', B'), se puede demostrar que

$$c(A', B') = C - \left(\sum g_i - \sum c_i\right)$$

donde g_i es la ganancia del proyecto i y c_i es el costo para contratar el área i. Tenemos tres tipos de aristas: Las aristas que conectan s con los proyectos, las aristas que conectan las áreas con t y las aristas que conectan los proyectos con las áreas. Estas ultimas no contribuyen a dicho corte debido al supuesto inicial sobre A, esto es, si está un proyecto también están las áreas requeridas para el mismo.

Las aristas que conectan s con los proyectos que no están en A aportan al corte

$$\sum_{project\ i \notin A} g_i$$

Las aristas que conectan las áreas que están en A con t aportan al corte

$$\sum_{area\ i \in A} c_i$$

Usando la definición de C

$$\sum_{project\ i \notin A} g_i = C - \sum_{project\ i \in A} g_i$$

Por lo tanto, la capacidad del corte $c(A', B')$ es

$$c(A', B') = \left(C - \sum_{project\ i \in A} g_i \right) + \sum_{area\ i \in A} c_i$$

$$c(A', B') = C - \left(\sum_{project\ i \in A} g_i - \sum_{area\ i \in A} c_i \right)$$

Entonces, si (A', B') es un corte con capacidad a lo sumo C, entonces el conjunto $A = A' - \{s\}$ es válido, ya que no existirá ninguna arista del estilo $(proy, area)$ que contribuya al corte.

Al maximizar el flujo, obtenemos un corte mínimo. Esto quiere decir que maximizamos $\sum_{project\ i \in A} g_i - \sum_{area\ i \in A} c_i$, que es lo que buscamos.

Volviendo al grafo de ejemplo, al aplicar el algoritmo de Ford Fulkerson, obtenemos un corte c(S,T), tal que

$$S = \{s, Proyecto3, Proyecto4, Area3\}$$

$$T = \{t, Proyecto1, Proyecto2, Area1, Area2\}$$

La conclusión que obtenemos de este resultado es que:

- Los proyectos a realizar son: Proyecto 3, Proyecto 4
- Las áreas a contratar son: Área 3
- Los proyectos a descartar: Proyecto 1, Proyecto 2

- Las áreas a no contratar son: Área1, Área 2

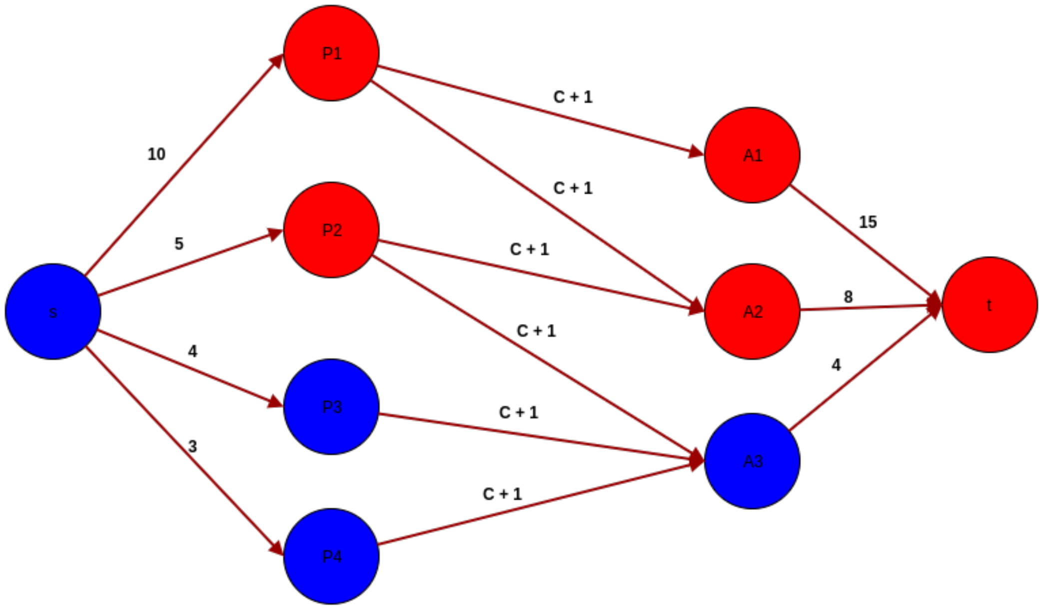

Figure 3: Resultado aplicado al grafo. En azul el conjunto S, en rojo el conjunto T.

Se puede observar que no hay ninguna arista de capacidad $C + 1$ que contribuya a la capacidad del corte. Esto significa que no existe ningún proyecto i en el conjunto S tal que alguna de sus áreas requeridas esté en el conjunto T, lo cual es correcto ya que en ese caso no se cumplirían las restricciones necesarias para la realización del proyecto.

2.2 Análisis de complejidad

Para realizar el análisis vamos a suponer lo siguiente. Sea G el grafo y sea m la cantidad de aristas y n la cantidad de vértices. Diremos que cada vértice tiene al menos una arista incidente, por lo tanto $m >= \frac{n}{2}$, de modo tal que $O(m + n) = O(m)$, con el fin de simplificar las cuentas.

Lo primero que podemos observar es que necesitamos calcular un camino desde el nodo s hasta el nodo t. Utilizando BFS, logramos esto con una complejidad de $O(m)$.

Luego, aplicamos el método "augment", que tiene una complejidad de $O(m)$ debido a que se calcula el cuello de botella del camino obtenido en el BFS, teniendo que iterar por todas las aristas, y luego se itera nuevamente cada arista para mejorar el flujo donde cada iteración realiza operaciones finitas en tiempo $O(1)$.

Lo que faltaría determinar es la cantidad de iteraciones que realiza este algoritmo. Dentro del método "augment", el método "bottleneck" calcula el cuello de botella y en función de ese resultado se mejora el flujo del camino obtenido. En el peor de los casos, tendríamos que en cada iteración a realizar obtengamos un cuello de botella de 1, provocando que tengamos que iterar hasta C veces como máximo para maximizar el flujo, siendo $C = \sum_{e\ out\ of\ s} c_e$, para que de esta forma no quede camino de s hacia t en el grafo residual y pueda terminar el algoritmo.

En conclusión, la complejidad del algoritmo es:

$$O(C * m)$$

Sea p el numero de proyectos, a el número de áreas y r la cantidad total de requisitos por parte de todos los proyectos. Estos números determinan la cantidad de aristas que tendrá nuestro grafo sobre el cual aplicaremos el algoritmo de Ford-Fulkerson.

Dicha relación la podemos calcular teniendo en cuenta que para crear el grafo agregamos p aristas que unen al vértice s con cada proyecto, agregamos a aristas que unen cada área con el vértice t, y luego tenemos una arista por cada requisito entre un proyecto y un área, agregándose r aristas más. Por lo tanto tenemos que:

$$m = p + a + r$$

Por lo tanto, la complejidad del algoritmo utilizando estos parámetros es:

$$O(C * (p + a + r))$$

2.3 Análisis de tiempos

Para poder analizar los tiempos de ejecución, supondremos como casos una entrada donde el número de proyectos y el número de áreas esté fijado a 1000. Las ganancias y los costos los dejaremos fijo a una unidad. Variaremos la cantidad de restricciones r para ver como se comporta el algoritmo. $C = \sum g_i$ no variará, por lo que solo tendrá influencia la cantidad de restricciones.

r	clocks
1000	94334
2000	142673
4000	301215
8000	266351
16000	348620
32000	723408
64000	1507858
128000	3052011
256000	5779835
512000	11684001

Table 3: Tabla de tiempos

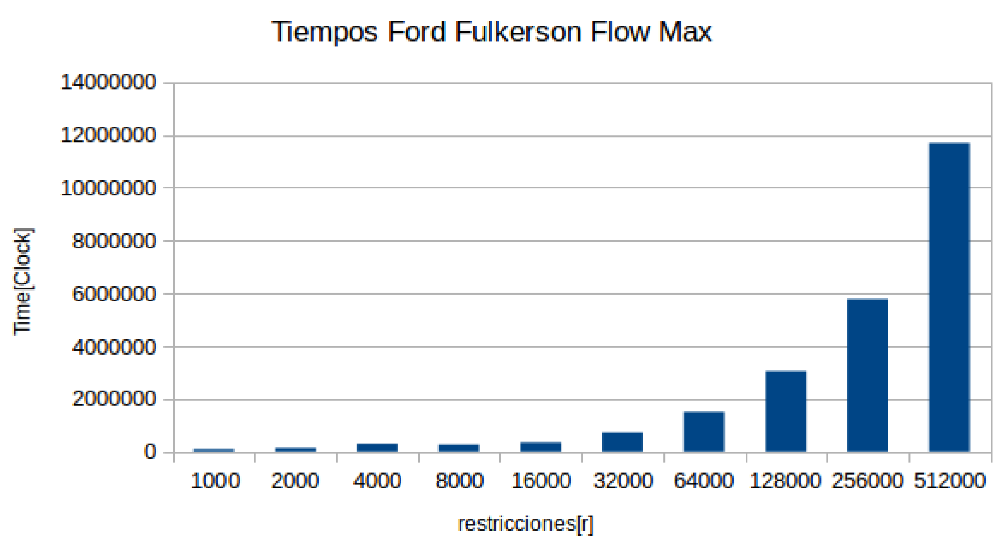

Figure 4: Gráfico de tiempos

Por medio del gráfico, se puede verificar el comportamiento lineal del algoritmo. Esto es, cuando duplicamos la cantidad de restricciones, se duplica el tiempo insumido para la ejecución del algoritmo.

2.4 Implementación del algoritmo

```cpp

#include "Edge.h"

Edge::Edge(int source, int dest, int capacity) {
   this->source = source;
   this->dest = dest;
    this->capacity = capacity;
}

int Edge::getSource(){
   return this->source;
}

int Edge::getDest(){
   return this->dest;
}

int Edge::getCapacity() {
    return this->capacity;
}

Edge::~Edge() {
}

void Edge::setCapacity(int capacity) {
    this->capacity = capacity;
}
```

Listing 5: Edge.cpp

```cpp
#include "EdgeInfo.h"

EdgeInfo::EdgeInfo(Edge  edge, int flow, int capacity) {
    this->edge = edge;
    this->flow = flow;
    this->capacity = capacity;
}

int EdgeInfo::getCapacity() {
    return this->capacity;
}
```

```cpp

int EdgeInfo::getFlow() {
    return this->flow;
}

int EdgeInfo::getResidualCapacity() {
    return this->capacity - this->flow;
}

void EdgeInfo::setFlow(int newFlow) {
    this->flow = newFlow;
}
```

Listing 6: EdgeInfo.cpp

```cpp
#include "Flow.h"

Flow::Flow(Digraph  digraph) {
    this->digraph = digraph;
    this->init();
}

void Flow::init() {
    // f(e) = 0 para toda arista del grafo.
    for (int v = 0; v < this->digraph->getVertices(); v++) {
        std::list<Edge > list = this->digraph->getAdjList(v);
        for (std::list<Edge >::iterator it = list->begin(); it != list->end(); ++it) {
            Edge  edge =  it;
            int flow = 0;
            EdgeInfo  edgeInfo = new EdgeInfo(edge, flow, edge->getCapacity());
            this->edgeMap[edge] = edgeInfo;
        }
    }
}

EdgeInfo  Flow::getEdgeInfo(Edge  edge) {
    return this->edgeMap[edge];
}

Flow::~Flow() {
    for (map<Edge ,EdgeInfo >::iterator it = this->edgeMap.begin(); it != this->edgeMap.end(); ++it) {
```

```cpp
        delete it->second;
    }
}
```

Listing 7: Flow.cpp

```cpp
#ifndef TRABAJOPRACTICO2_FORDFULKENSON_H
#define TRABAJOPRACTICO2_FORDFULKENSON_H

#include <vector>
#include "Edge.h"
#include "Digraph.h"
#include "Flow.h"
#include "ParserNetworkFlow.h"

using namespace std;

class FordFulkerson {
private:
    ParserNetworkFlow  parser;
    Flow  flow;
    vector<int> setS;
    vector<int> setT;

    int bottleneck(vector<Edge > pathInResidualGraph);

    void augment(vector<Edge > pathInResidualGraph);

    bool isEdgeForward(Edge  edge);

    Edge  getEdgeInG(Edge  edgeInGResidual);

    vector<Edge  > getPathST();

    void calculateMinCut();

public:

    FordFulkerson(ParserNetworkFlow  parser);
    ~FordFulkerson();

    void maxFlow();

    vector<int> getSetS();
```

```
    vector<int> getSetT();

    vector<int> getProjects();
    vector<int> getAreas();

};


#endif //TRABAJOPRACTICO2_FORDFULKENSON_H
```

Listing 8: FordFulkerson.h

```
#include "FordFulkerson.h"
#include "ParserNetworkFlow.h"
#include "PathBFS.h"

FordFulkerson::FordFulkerson(ParserNetworkFlow  parser) {
    this->parser = parser;
    this->flow = new Flow(parser->graph);

}

FordFulkerson::~FordFulkerson() {
    delete flow;
}

/
   Calcula el cuello de botella del camino pasado como argumento
  /
int FordFulkerson::bottleneck(vector<Edge > pathInResidualGraph) {
    int min = pathInResidualGraph.at(0)->getCapacity();

    for (unsigned int i = 1; i < pathInResidualGraph.size(); i++) {
        int residualCapacity = pathInResidualGraph.at(i)->getCapacity();
        if (residualCapacity < min) {
            min = residualCapacity;
        }
    }
    return min;
}
```

```


Edge  FordFulkerson::getEdgeInG(Edge  edgeInGResidual) {
    return this->parser->mapping.mapEdgeResidualGToEdgeG[edgeInGResidual];
}


bool FordFulkerson::isEdgeForward(Edge  edgeInGResidual) {
    return this->parser->mapping.isEdgeResidualForward[edgeInGResidual];
}

void FordFulkerson::augment(vector<Edge  > pathInResidualGraph){
    int b = this->bottleneck(pathInResidualGraph);
    for (vector<Edge >::iterator it = pathInResidualGraph.begin(); it != pathInResidualGraph.end(); ++it) {
        Edge  edgeInGResidual =  it;
        Edge  edgeInG = this->getEdgeInG(edgeInGResidual);
        int newFlow;
        if (this->isEdgeForward(edgeInGResidual)) {
            newFlow = this->flow->getEdgeInfo(edgeInG)->getFlow() + b;
        } else {
            newFlow = this->flow->getEdgeInfo(edgeInG)->getFlow() - b;
        }

        this->flow->getEdgeInfo(edgeInG)->setFlow(newFlow);
        pair<Edge ,Edge > edgesInResidualG = this->parser->mapping.mapEdgeGToResidualG[edgeInG];
        Edge  edgeForward = edgesInResidualG.first;
        Edge  edgeBackward = edgesInResidualG.second;
        edgeForward->setCapacity(edgeInG->getCapacity() - newFlow);
        edgeBackward->setCapacity(newFlow);
    }
}

vector<Edge  > FordFulkerson::getPathST() {
    int sVertex = 0;
    int tVertex = this->parser->areasCount + this->parser->projectsCount + 1;
    vector<Edge > path;
```

```
    PathBFS pathBFS(this->parser->residualGraph, sVertex, tVertex);
    if (pathBFS.visited(tVertex)) {
        std::list<Edge > pathList = pathBFS.pathTo(tVertex);
        for (std::list<Edge >::iterator it = pathList.begin(); it != pathList.end(); ++it) {
            Edge edge = it;
            path.push_back(edge);
        }
    }

    return path;
}

void FordFulkerson::maxFlow() {
    while (true) {
        vector<Edge > path = getPathST();
        if (path.empty()) break;
        this->augment(path);
    }

    this->calculateMinCut();
}

void FordFulkerson::calculateMinCut() {
    int sVertex = 0;
    int tVertex = this->parser->areasCount + this->parser->projectsCount + 1;

    PathBFS pathBFS(this->parser->residualGraph, sVertex, tVertex);
    for (int i = 1; i < this->parser->residualGraph->getVertices() - 1; i++) {
        if (pathBFS.visited(i)) {
            this->setS.push_back(i);
        } else {
            this->setT.push_back(i);
        }
    }
}

vector<int> FordFulkerson::getSetT() {
    return this->setT;
}

```

```
vector<int> FordFulkerson::getSetS() {
    return this->setS;
}

vector<int> FordFulkerson::getProjects() {
    vector<int> projectsToDo;
    for (std::vector<int>::iterator it = this->setS.begin(); it↩
        != this->setS.end(); ++it ) {
        VertexInfo vertexInfo = this->parser->vertexMap[(it)];
        if (vertexInfo.isProject) {
            projectsToDo.push_back(vertexInfo.number);
        }
    }
    return projectsToDo;
}

vector<int> FordFulkerson::getAreas() {
    vector<int> areasToPay;
    for (std::vector<int>::iterator it = this->setS.begin(); it↩
        != this->setS.end(); ++it ) {
        VertexInfo vertexInfo = this->parser->vertexMap[(it)];
        if (!vertexInfo.isProject) {
            areasToPay.push_back(vertexInfo.number);
        }
    }
    return areasToPay;
}
```

Listing 9: FordFulkerson.cpp

```
#ifndef TRABAJOPRACTICO2_MAPPING_H
#define TRABAJOPRACTICO2_MAPPING_H

#include <map>
#include "Edge.h"

using namespace std;

// El mapeo se realiza para obtener mejores tiempos, y no estar↩
     creando el grafo residual en cada óiteracin del algoritmo
// de Ford-Fulkerson.
class Mapping {
public:
    map<Edge, pair<Edge,Edge>> mapEdgeGToResidualG;
```

```
    map<Edge , Edge > mapEdgeResidualGToEdgeG;
    map<Edge , bool> isEdgeResidualForward;

    Mapping() {

    }
};


#endif //TRABAJOPRACTICO2_MAPPING_H
```

Listing 10: Mapping.h

```
#include "ParserNetworkFlow.h"
#include "Digraph.h"
#include <fstream>
#include <vector>
#include <sstream>

ParserNetworkFlow::ParserNetworkFlow(string fileName) {
    this->fileName = fileName;
    this->parse();
    this->createGraphs();
}

vector<int> ParserNetworkFlow::getAreaCosts(ifstream& file, int areasCount) {
    vector<int> costs;
    int i = 0;
    string line;
    while (getline(file,line)) {
        costs.push_back(stoi(line));
        i++;
        if (i == areasCount) break;
    }

    return costs;
}

int ParserNetworkFlow::calculateTotalCapacity(vector<int> v) {
    int total = 0;
    for (int i = 0; i < v.size(); i++) {
        total += v.at(i);
    }
```

```

    return total;
}

pair<int,vector<int>> ParserNetworkFlow::splitLine(string line, char delim) {
    vector<int > dependencies;
    stringstream ss;
    ss.str(line);
    string item;

    getline(ss, item, delim);
    int profit = stoi(item);

    while (getline(ss, item, delim)) {
        if (!item.empty()) {
            dependencies.push_back(stoi(item));
        }
    }

    return pair<int,vector<int>>(profit, dependencies);
}

void ParserNetworkFlow::getProjects(ifstream& file, int count) {
    int i = 0;
    string line;
    while (getline(file,line)) {
        pair<int,vector<int>> split = splitLine(line,    );
        this->profits.push_back(split.first);
        this->projectDependencies.push_back(split.second);

        i++;
        if (i == count) break;
    }
}


void ParserNetworkFlow::parse() {
    ifstream file = ifstream(this->fileName.c_str());
    if (!file.is_open()) {
        throw string("Error al abrir el archivo de flujo ") + this->fileName;
    }
    string line;
```

```

    getline(file, line);
    this->areasCount = stoi(line);

    getline(file,line);
    this->projectsCount = stoi(line);

    this->areaCosts = getAreaCosts(file, this->areasCount);

    getProjects(file,this->projectsCount);
    this->infinityCapacity = calculateTotalCapacity(this->profits) + 1;

    file.close();
}


void ParserNetworkFlow::createGraphs() {
    this->graph = new Digraph(this->projectsCount + this->areasCount + 2);
    this->residualGraph = new Digraph(this->projectsCount + this->areasCount + 2);

    int sVertex = 0;
    int tVertex = this->projectsCount + this->areasCount + 1;

    for (int i = 0; i < this->projectsCount; i++) {
        // Agrego aristas s -> projects
        int projectVertex = i + 1;
        Edge  edgeG = new Edge(sVertex, projectVertex, this->profits.at(i));
        Edge  edgeForward = new Edge(sVertex, projectVertex, this->profits.at(i));
        Edge  edgeBackward = new Edge(projectVertex, sVertex, 0);
        this->mapping.isEdgeResidualForward[edgeForward] = true;
        this->mapping.isEdgeResidualForward[edgeBackward] = false;
        this->mapping.mapEdgeResidualGToEdgeG[edgeForward] = edgeG;
        this->mapping.mapEdgeResidualGToEdgeG[edgeBackward] = edgeG;

        this->mapping.mapEdgeGToResidualG[edgeG] = pair<Edge ,
```

```
            Edge >(edgeForward, edgeBackward);
        VertexInfo vertexInfo("Projecto:" + to_string(i+1), i↩
            +1, true);
        this->vertexMap[projectVertex] = vertexInfo;

        this->graph->addEdge(edgeG);
        this->residualGraph->addEdge(edgeForward);
        this->residualGraph->addEdge(edgeBackward);
    }

    for (int i = 0; i < this->areasCount; i++) {
        // Agrego aristas areas-> t
        int areaVertex = this->projectsCount+i+1;
        Edge  edgeG = new Edge(areaVertex, tVertex, this->↩
            areaCosts.at(i));
        Edge  edgeForward = new Edge(areaVertex, tVertex, this↩
            ->areaCosts.at(i));
        Edge  edgeBackward = new Edge(tVertex, areaVertex, 0);
        this->mapping.isEdgeResidualForward[edgeForward] = true↩
            ;
        this->mapping.isEdgeResidualForward[edgeBackward] = ↩
            false;
        this->mapping.mapEdgeResidualGToEdgeG[edgeForward] = ↩
            edgeG;
        this->mapping.mapEdgeResidualGToEdgeG[edgeBackward] = ↩
            edgeG;

        this->mapping.mapEdgeGToResidualG[edgeG] = pair<Edge ,↩
            Edge >(edgeForward, edgeBackward);
        VertexInfo vertexInfo("Area:" + to_string(i+1), i+1, ↩
            false);
        this->vertexMap[areaVertex] = vertexInfo;

        this->graph->addEdge(edgeG);
        this->residualGraph->addEdge(edgeForward);
        this->residualGraph->addEdge(edgeBackward);
    }

    for (int i = 0; i < this->projectsCount; i++) {
        // Agregado de dependencias
        int projectVertex = i + 1;
        vector<int> dependencies = this->projectDependencies.at↩
            (i);
        for (vector<int >::iterator it = dependencies.begin(); ↩
            it != dependencies.end(); ++it) {
```

```
            int areaVertex = this->projectsCount + ( it); // Se asume que el archivo indica las areas a partir de 1.

            Edge  edgeG = new Edge(projectVertex, areaVertex, this->infinityCapacity);
            Edge  edgeForward = new Edge(projectVertex, areaVertex, this->infinityCapacity);
            Edge  edgeBackward = new Edge(areaVertex, projectVertex, 0);
            this->mapping.isEdgeResidualForward[edgeForward] = true;
            this->mapping.isEdgeResidualForward[edgeBackward] = false;
            this->mapping.mapEdgeResidualGToEdgeG[edgeForward] = edgeG;
            this->mapping.mapEdgeResidualGToEdgeG[edgeBackward] = edgeG;

            this->mapping.mapEdgeGToResidualG[edgeG] = pair<Edge ,Edge >(edgeForward, edgeBackward);

            this->graph->addEdge(edgeG);
            this->residualGraph->addEdge(edgeForward);
            this->residualGraph->addEdge(edgeBackward);
        }
    }

}
```

Listing 11: ParserNetworkFlow.cpp

```

#include "PathBFS.h"

#include <queue>

using namespace std;

PathBFS::PathBFS(Digraph  g, int source, int dest ) : Path(g, source, dest ){

  for(int i=0;i<g->getVertices();i++) {
    marked[i] = false;
```

```
      distance[i] = Path::NO_PATH;         //Inicializo distancias con distancia infinito
   }

   queue<int> queue;

   marked[this->source] = true;
   distance[this->source] = 0;

   queue.push(this->source);  //arranco desde el source

   //Mientras haya un vertice pendiente y no se haya encontrado el destino
   while(!queue.empty() && !marked[this->dest]){
      int v = queue.front();
      queue.pop();
      list<Edge > adjList = g->getAdjList(v);

      //Barro la lista de adyacencia
      for (std::list<Edge >::iterator it=adjList->begin(); it != adjList->end(); ++it){
         // Agregado adicional para no procesar las aristas invalidas del grafo residual.
         if ((it)->getCapacity() <= 0) continue;

          int vert = (it)->getDest();

          if(!marked[vert]){
            distance[vert] = distance[v] + 1; // Acumulo la distancia desde el origen hasta el vertice
            marked[vert]= true;
            queue.push(vert);
            edgeTo[vert] = it;  //guardo el camino por el que fui
          }
      }
   }

}

PathBFS::~PathBFS() {

}
```

Listing 12: PathBFS.cpp

```cpp
#ifndef TRABAJOPRACTICO2_VERTEXINFO_H
#define TRABAJOPRACTICO2_VERTEXINFO_H


class VertexInfo {
public:
    VertexInfo() {
    }

    VertexInfo(string description, int number, bool isProject) {
        this->description = description;
        this->isProject = isProject;
        this->number = number;
    }

    string description;
    bool isProject = false;
    int number = 0;
};


#endif //TRABAJOPRACTICO2_VERTEXINFO_H
```

Listing 13: VertexInfo.h

Referencias

[1] Algorithm Design, Jon Kleinberg, Eva Tardos, 7.11 Project Selection