

FACULTAD DE INGENIERÍA
UNIVERSIDAD DE BUENOS AIRES

TEORÍA DE ALGORITMOS

Trabajo Práctico N2

Autores:

Addin Kevin

Cabrera Jorge

Gatti Nicolas

Orlando Juan Manuel

Padrón:

94280

93310

93570

93152

18 de Noviembre, 2016



Contents

1	Programación dinámica	2
1.1	El problema de la mochila	2
1.1.1	Análisis de complejidad	2
1.1.2	Análisis de tiempos	2
1.1.3	Implementación del algoritmo	5
1.2	El problema del viajante de comercio	10
1.2.1	Análisis de complejidad	10
1.2.2	Análisis de tiempos	10
1.2.3	Implementación del algoritmo	12
2	Flujo de redes	20
2.1	Modelado del problema	20
2.2	Análisis de complejidad	23
2.3	Análisis de tiempos	24
2.4	Implementación del algoritmo	26

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

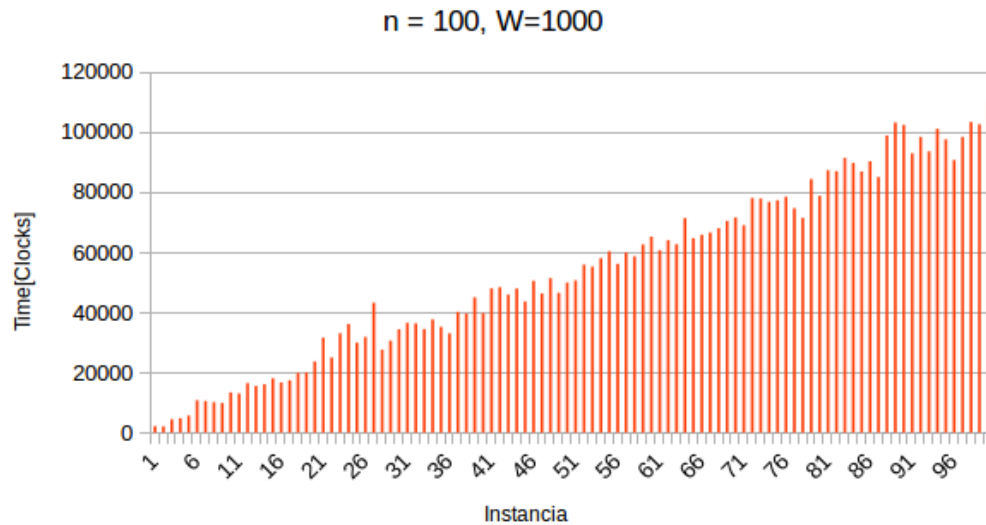


Figure 1: Gráfico de tiempos de las instancias de $n=100$, $W=1000$

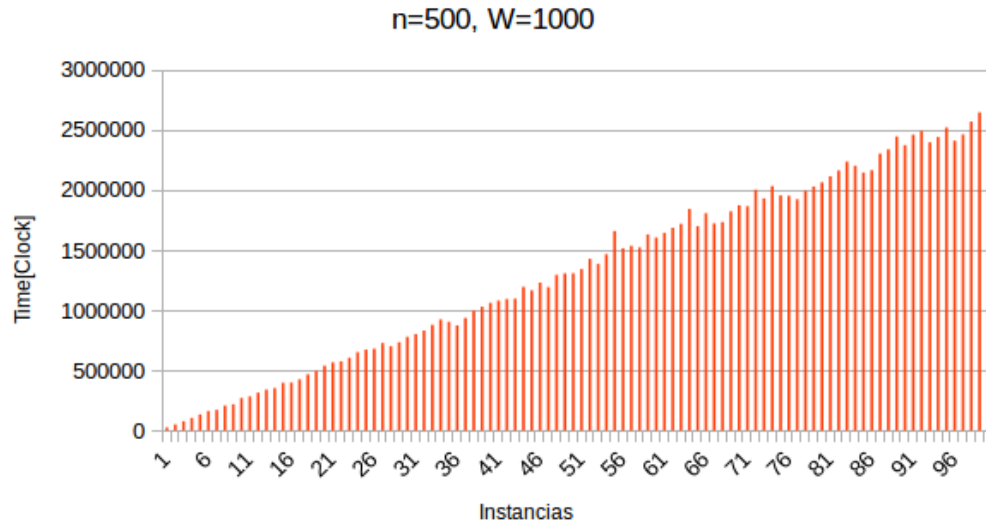


Figure 2: Gráfico de tiempos de las instancias de $n=500$, $W=1000$

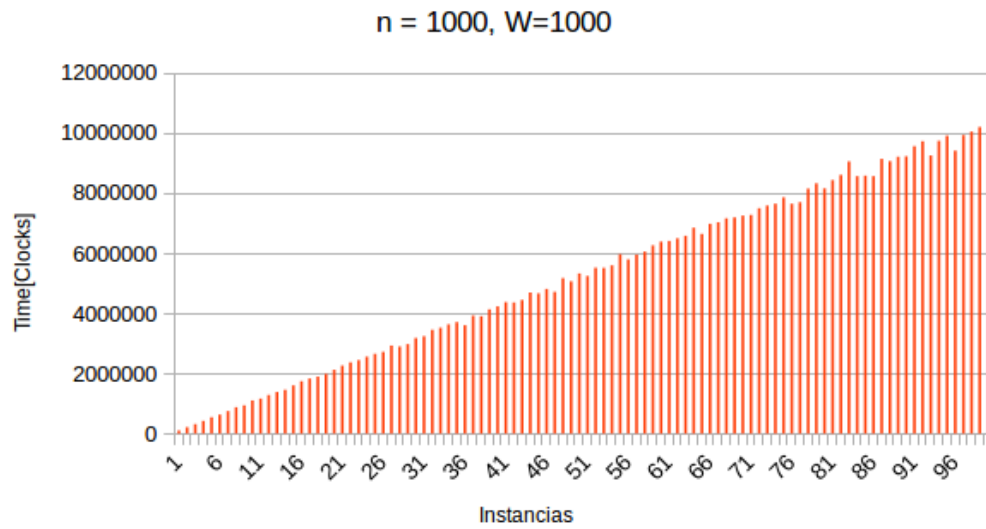


Figure 3: Gráfico de tiempos de las instancias de $n=1000$, $W=1000$

Se puede observar que los archivos de prueba están preparados de forma tal que las primeras instancias no incluyan a ningún ítem y que las ultimas instancias incluyan a todos los items. Es por esto que va aumentando el tiempo incurrido debido a que se hacen algunas operaciones adicionales.

n-w	Segundos
100-1000	0,05347443
500-1000	1,33120953
1000-1000	5,24668911

Table 1: Tiempos promedio

Es esta ultima tabla calculamos los tiempos promedios para las instancias de los archivos probados. Lo extraño que se puede observar es que a pesar de que duplicamos uno de los parámetros, no se duplicó el tiempo, sino que aumentó más del doble.

1.1.3 Implementación del algoritmo

```
1 #ifndef MOCHILA_H_
2 #define MOCHILA_H_
3 #include "../Headers/Parser.h"
4 #include "../Headers/Stage.h"
5 #include "../Headers/Node.h"
6 using namespace std;
7
8 class Mochila {
9 private:
10     int tPesos;
11     void initMochila(int n, int w);
12     Stage stage;
13     int n;
14     int w;
15
16 public:
17     Mochila(Stage stage);
18     ~Mochila();
19
20     // Devuelve el valor optimo obtenido.
21     int computarMochila();
22
23     // Devuelve un vector de los elementos que van a la mochila.
24     vector<int> getItemsInBag();
25
26
27 };
28
29
30 #endif / MOCHILA_H_ /
```

Listing 1: Mochila.h

```
1 #ifndef NODE_H_
2 #define NODE_H_
3
4 using namespace std;
5
6 class Node {
7
8 public:
```

```

9   int valor;
10  int weight;
11  bool isInBag;
12
13  Node();
14  ~Node();
15 };
16
17 #endif / NODE_H_ /

```

Listing 2: Node.h

```

1  #ifndef STAGE_H_
2  #define STAGE_H_
3
4  #include <vector>
5  #include <iostream>
6  #include <fstream>
7  #include <stdlib.h> / atoi /
8  #include "../Headers/Node.h"
9
10 using namespace std;
11
12 class Stage {
13 public:
14     string name;
15     int qtyNode;
16     int capacityBag;
17     int optimo;
18
19     vector<Node> nodes;
20
21     void loadNode(string line);
22     bool isFull();
23     Node getNode(int);
24
25     Stage();
26     ~Stage();
27 };
28
29
30 #endif / STAGE_H_ /

```

Listing 3: Stage.h

```

1 #include "../Headers/Mochila.h"
2 using namespace std;
3
4 Mochila::Mochila(Stage stage){
5     this->stage = stage;
6     this->n = this->stage.qtyNode;
7     this->w = this->stage.capacityBag;
8     this->initMochila(n,w);
9 }
10
11 void Mochila::initMochila(int n, int w) {
12     this->tPesos = new int [n+1];
13     for(int i = 0; i < n+1; ++i) {
14         tPesos[i] = new int[w + 1];
15     }
16
17     for (int i = 0; i <= n; i++) {
18         for(int j = 0; j <= w; j++) {
19             tPesos[i][j] = 0;
20         }
21     }
22 }
23
24
25 int Mochila::computarMochila(){
26     for(int i=1; i<=stage.qtyNode; i++){
27         for(int w=0;w<=stage.capacityBag;w++){
28             int weight = stage.getNode(i-1).weight;
29             int valor = stage.getNode(i-1).valor;
30             if(w < weight){
31                 tPesos[i][w] = tPesos[i-1][w];
32             }
33             else{
34                 if(tPesos[i-1][w] > (valor + tPesos[i-1][w-weight])){
35                     tPesos[i][w] = tPesos[i-1][w];
36                 }
37                 else{
38                     tPesos[i][w] = valor + tPesos[i-1][w-weight];
39                 }
40             }
41         }
42     }
43
44     return tPesos[n][w];

```



```

45 }
46
47 Mochila::~Mochila(){
48     for(int i = 0; i < this->n+1; ++i) {
49         delete [] tPesos[i];
50     }
51     delete [] tPesos;
52 }
53
54 vector<int> Mochila::getItemsInBag() {
55     vector<int> itemsInBag;
56
57     int weight = this->w;
58
59     for (int i = this->n; i >= 1; i--) {
60         if (this->tPesos[i][weight] != this->tPesos[i-1][weight]) ←
61             {
62                 // Se incluyo el item ya que mejoro la ósolucin
63                 itemsInBag.push_back(i);
64                 weight = weight - stage.getNode(i-1).weight;
65             }
66
67     }
68     return itemsInBag;
69 }

```

Listing 4: Mochila.cpp

```

1 #include <sstream>
2 #include "../Headers/Stage.h"
3
4
5 Stage::Stage() {
6 }
7
8 Stage::~Stage() {
9 }
10
11 void Stage::loadNode(string line){
12     Node newNode;
13
14     stringstream ss;
15     ss.str( line);
16     string item;

```

```

17     getline(ss, item, ',');
18     getline(ss, item, ',');
19     newNode.valor = stoi(item);
20     getline(ss, item, ',');
21     newNode.weight = stoi(item);
22     getline(ss, item, ',');
23     newNode.isInBag = (bool)stoi(item);
24
25     this->nodes.push_back(newNode);
26 }
27
28 bool Stage::isFull(){
29     return (this->nodes.size() == this->qtyNode) && (this->qtyNode>=1);
30 }
31
32 Node Stage::getNode(int i){
33     return this->nodes[i];
34 }

```

Listing 5: Stage.cpp

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 2: Tabla de tiempos

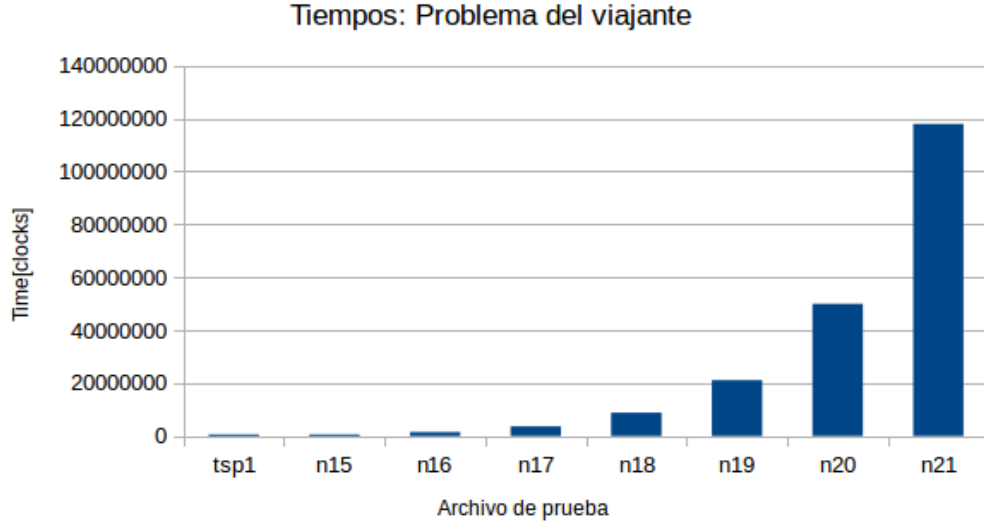


Figure 4: Gráfico de tiempos

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

```
1 #include "TSPRecursive.h"
2
3 /
4 vector<vector<CostInt>> matrix:
5     Matriz de costos no negativos, se asume que es una ↵
6     matriz cuadrada N N, N = size(matrix)
7
8 VertexInt initialVertex:
9     Vertice inicial para la óejecucin del algoritmo
10
11 /
12 TSPRecursive::TSPRecursive(vector<vector<CostInt>> matrix, ↵
13     VertexInt initialVertex) {
14     this->matrix = matrix;
15     this->vertexCount = this->matrix->size();
16
17     this->memory = new vector<map<SetInt, CostInt>>();
18
19     this->path = new vector<map<SetInt, VertexInt>>();
20
21     for (unsigned int i = 0; i < this->vertexCount; i++) {
22         this->memory->push_back(map<SetInt, CostInt>());
23         this->path->push_back(map<SetInt, VertexInt>());
24     }
25
26     this->initialVertex = initialVertex;
27 }
28
29 TSPRecursive::~TSPRecursive() {
30     delete this->memory;
31     delete this->path;
32 }
33
34 /
35 Devuelve el vector de vertices que contiene el recorrido a ↵
36 seguir.
37
38 /
39 vector<VertexInt> TSPRecursive::createPathList(SetInt ↵
40     setNumber, VertexInt initialVertex) {
41     vector<VertexInt> pathList = new vector<VertexInt>();
42     pathList->push_back(initialVertex);
```

```

39     VertexInt actualVertex = initialVertex;
40
41     for (unsigned int i = 0 ; i < this->vertexCount - 1 ; i++) {
42         VertexInt nextVertex = this->path->at(actualVertex).at(
43             setNumber);
44         pathList->push_back(nextVertex);
45         setNumber = this->getSetNumberWithOutVertex(setNumber,
46             nextVertex);
47         actualVertex = nextVertex;
48     }
49     pathList->push_back(initialVertex);
50     return pathList;
51 }
52 /
53 Apaga el n-esimo bit , siendo n = vertexNumber , sobre el
54 setNumber.
55 óPrecondicin: el n-esimo bit áest encendido.
56 /
57 SetInt TSPRecursive::getSetNumberWithOutVertex(SetInt setNumber,
58     VertexInt vertexNumber) {
59     return setNumber - (1<< vertexNumber);
60 }
61 /
62 Lleva a cabo la óejecucin del algoritmo.
63 Se obtiene un par pair<CostInt, vector<VertexInt>>
64 que contiene el costo de recorrer todas las ciudades
65 (suma de los pesos de las aristas recorridas)
66 y un vector que contiene los vertices en orden para
67 realizar el recorrido y alcanzar dicho costo.
68 /
69 pair<CostInt, vector<VertexInt> > TSPRecursive::run() {
70     SetInt setNumber = (1 << this->vertexCount) - 1;
71     CostInt cost = this->_run(this->initialVertex, this->
72         getSetNumberWithOutVertex(setNumber, initialVertex));
73     vector<VertexInt> pathList = this->createPathList(this->
74         getSetNumberWithOutVertex(setNumber, initialVertex),
75         initialVertex);
76
77     return pair<CostInt, vector<VertexInt>>(cost, pathList);
78 }
79

```

```

76 /
77  óFuncin auxiliar que calcula la óversin recursiva
78  del problema del viajante utilizando óprogramacin
79  dinamica.
80 /
81 CostInt TSPRecursive::_run(VertexInt vertex, SetInt setNumber) ←
    {
82
83     if (setNumber == 0) {
84         // Conjunto vacio. Retorno el costo de ir hacia el ←
            vertice inicial.
85         return this->matrix->at(vertex).at(this->initialVertex)←
            ;
86     } else if (this->memory->at(vertex).count(setNumber)) {
87         // Ya fue calculado
88         return this->memory->at(vertex).at(setNumber);
89     }
90
91     CostInt minCost = (CostInt)(-1);
92     VertexInt vertexMin;
93
94     SetInt copySetNumber = setNumber;
95
96     for (VertexInt u = 0; u < this->vertexCount; u++) {
97         unsigned int n = copySetNumber & 1;
98         if (n == 1) {
99             // Proceso el vertex u que pertenece al set.
100             CostInt result = this->matrix->at(vertex).at(u) +
101                 this->_run(u, this->←
                    getSetNumberWithOutVertex(←
                        setNumber, u));
102
103             if (result < minCost) {
104                 minCost = result;
105                 vertexMin = u;
106             }
107         }
108         copySetNumber = copySetNumber >> 1; // setNumber /= 2;
109     }
110
111     this->memory->at(vertex)[setNumber] = minCost;
112     this->path->at(vertex)[setNumber] = vertexMin;
113
114     return minCost;
115

```

116 }

Listing 6: TSPRecursive.cpp

```
1 #ifndef TRABAJOPRACTICO2_TSPRECURSIVE_H
2 #define TRABAJOPRACTICO2_TSPRECURSIVE_H
3
4 #include "Types.h"
5 #include <vector>
6 #include <utility>
7 #include <map>
8
9 using namespace std;
10
11 class TSPRecursive {
12 private:
13     vector<vector<CostInt>> matrix;
14     VertexInt vertexCount;
15     vector<map<SetInt, CostInt>> memory;
16     vector<map<SetInt, VertexInt>> path;
17     VertexInt initialVertex;
18
19     SetInt getSetNumberWithOutVertex(SetInt setNumber, ↵
        VertexInt vertexNumber);
20
21     vector<VertexInt> createPathList(SetInt setNumber, ↵
        VertexInt initialVertex);
22
23     CostInt _run(VertexInt vertex, SetInt setNumber);
24
25 public:
26     /
27     vector<vector<CostInt>> matrix:
28         Matriz de costos no negativos, se asume que es una ↵
        matriz cuadrada N N, N = size(matrix)
29
30     VertexInt initialVertex:
31         Vertice inicial para la óejecucin del algoritmo
32
33     /
34     TSPRecursive(vector<vector<CostInt>> matrix, VertexInt ↵
        initialVertex);
35     ~TSPRecursive();
36
```



```

37 /
38     Lleva a cabo la ejecución del algoritmo.
39     Se obtiene un par pair<CostInt, vector<VertexInt>>
40     que contiene el costo de recorrer todas las ciudades
41     (suma de los pesos de las aristas recorridas)
42     y un vector que contiene los vertices en orden para
43     realizar el recorrido y alcanzar dicho costo.
44 /
45     pair<CostInt, vector<VertexInt>> run();
46
47 };
48
49
50 #endif //TRABAJOPRACTICO2_TSPRECURSIVE.H

```

Listing 7: TSPRecursive.h

```

1 #ifndef TRABAJOPRACTICO2_TSPTEST.H
2 #define TRABAJOPRACTICO2_TSPTEST.H
3
4 #include "Types.h"
5 #include "ParserTSPFile.h"
6 #include "TSPRecursive.h"
7 #include <vector>
8 #include <string>
9 #include <iostream>
10
11 using namespace std;
12
13 class TSPTest {
14 public:
15     CostInt calculateCost(vector<vector<CostInt>> matrixCost, ↵
16         vector<VertexInt> list) {
17         CostInt cost = 0;
18         for (unsigned int i = 0; i < list->size() - 1 ; i++) {
19             VertexInt x = list->at(i);
20             VertexInt y = list->at(i+1);
21             cost += matrixCost->at(x).at(y);
22         }
23         return cost;
24     }
25
26     bool checkPathList(vector<VertexInt> expectedList, vector<↵

```

```

27     VertexInt > actualList) {
28         if (expectedList->size() != actualList->size()) {
29             return false;
30         }
31         // Puede que los recorridos éestn invertidos , pero ↵
32         // valen igual en el caso de que la matriz sea ↵
33         // simetrica.
34         bool toReturn = true;
35         for (unsigned int i = 0; i < expectedList->size(); i++)↵
36         {
37             if (expectedList->at(i) != actualList->at(i)) {
38                 toReturn = false;
39                 break;
40             }
41         }
42         if (toReturn) return true;
43
44         toReturn = true;
45         unsigned int size = expectedList->size();
46         for (unsigned int i = 0; i < expectedList->size(); i++)↵
47         {
48             if (expectedList->at(i) != actualList->at(size-i-1)↵
49                 ) {
50                 toReturn = false;
51                 break;
52             }
53         }
54         return toReturn;
55     }
56
57     void printRecorrido(string msg, vector<VertexInt> pathList↵
58     ) {
59         std::cout << msg << std::endl;
60         for (unsigned int i = 0; i < pathList->size(); i++) {
61             VertexInt vertexInt = pathList->at(i);
62             std::cout << int(vertexInt) << "\t";
63         }
64         std::cout << std::endl;
65     }
66
67     void runExample(string matrixFileName, string ↵
68     solutionFileName) {

```

```

64     ParserTSPFile parserTSPFile(matrixFileName, ↵
        solutionFileName);
65     vector<vector<CostInt>> matrix = parserTSPFile.↵
        getMatrix();
66     vector<VertexInt> expectedList = parserTSPFile.↵
        getSolutionList();
67     CostInt expectedCost = calculateCost(matrix, ↵
        expectedList);
68
69     TSPRecursive tspRecursive = new TSPRecursive(matrix, ↵
        0);
70
71     pair<CostInt, vector<VertexInt>> par = tspRecursive->↵
        run();
72
73     if (expectedCost != par.first) {
74         delete tspRecursive;
75         delete matrix;
76         delete expectedList;
77         delete par.second;
78         throw string("Error al coincidir los costos en " + ↵
            matrixFileName + ". Se esperaba ") \
79             + to_string(expectedCost) + string(" y se ↵
                obtuvo: ") + to_string(par.first);
80     }
81
82     if (!checkPathList(expectedList, par.second)) {
83         printRecorrido("Esperado", expectedList);
84         printRecorrido("Obtenido", par.second);
85
86         delete tspRecursive;
87         delete matrix;
88         delete expectedList;
89         delete par.second;
90         throw string("No hubo coincidencia en los ↵
            recorridos.");
91     }
92
93     std::cout << "Ejecucion de " << matrixFileName << " ↵
        exitosa." << std::endl;
94
95     delete tspRecursive;
96     delete matrix;
97     delete expectedList;
98     delete par.second;

```

```

99     }
100
101 };
102
103
104 #endif //TRABAJOPRACTICO2_TSPTEST.H

```

Listing 8: TSPTTest.h

```

1
2 #ifndef TRABAJOPRACTICO2_TYPES.H
3 #define TRABAJOPRACTICO2_TYPES.H
4
5 #include <stdint.h>
6
7 // La finalidad de estos tipos fue probar distintos tipos de ↵
   órepresentacin para ver si se lograba
8 // ejecutar el archivo de prueba de 26 nodos. Por ese motivo el ↵
   VertexInt áest en 8bits(256 valores posibles).
9 typedef uint8_t VertexInt;
10 typedef uint32_t CostInt;
11 typedef uint32_t SetInt;
12
13 #endif //TRABAJOPRACTICO2_TYPES.H

```

Listing 9: 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 3: 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:

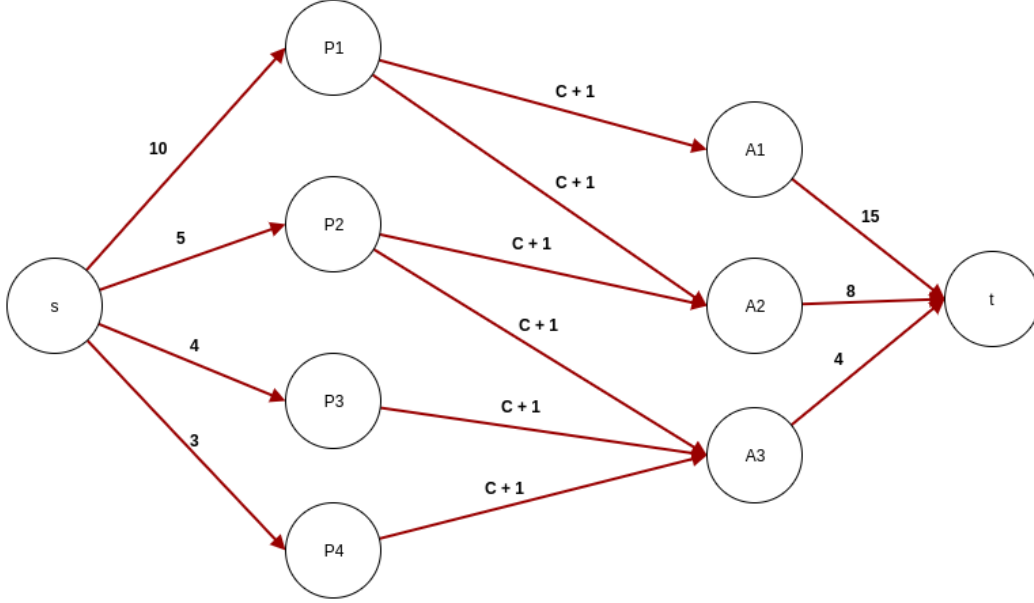


Figure 5: Grafo de ejemplo

Para obtener qué proyectos se realizarán y qué áreas se contratarán 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

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

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

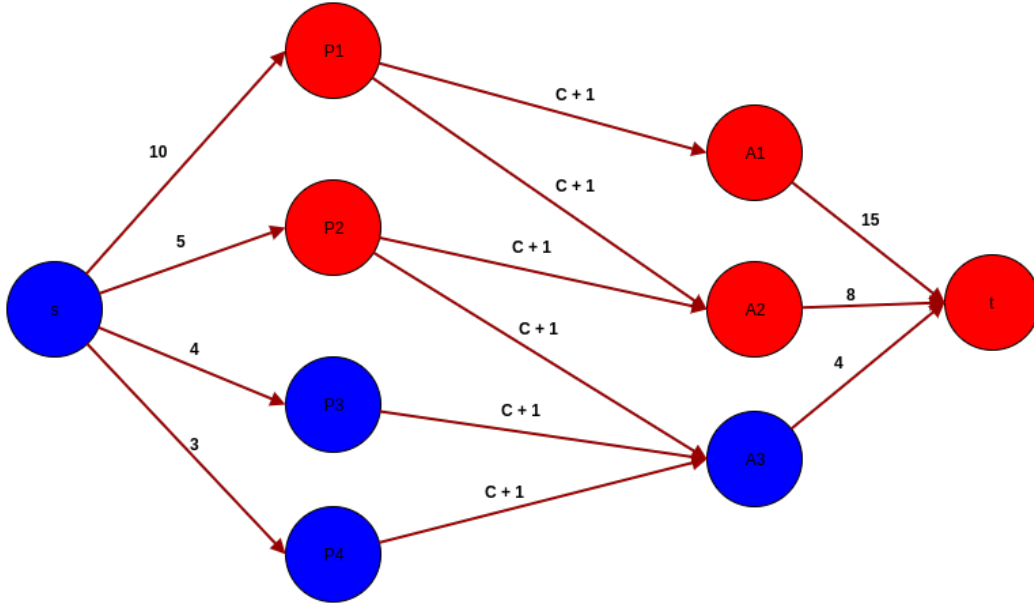


Figure 6: Resultado aplicado al grafo. En azul el conjunto S, en rojo el conjunto T.

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 $m \geq \frac{n}{2}$, 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 $C = \sum_{e \text{ out of } s} c_e$, 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 4: Tabla de tiempos

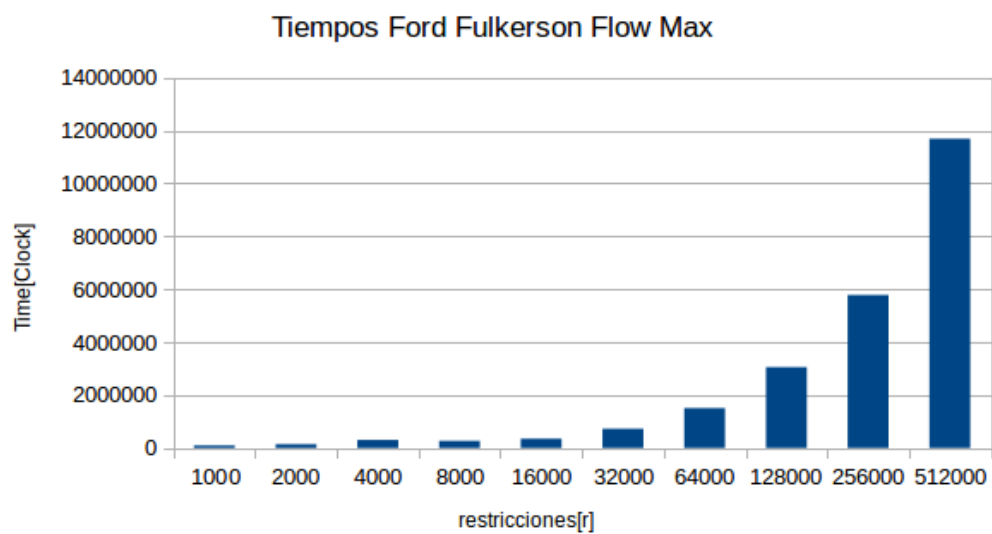


Figure 7: Gráfico de tiempos

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

```
1
2 #include "Edge.h"
3
4 Edge::Edge(int source, int dest, int capacity) {
5     this->source = source;
6     this->dest = dest;
7     this->capacity = capacity;
8 }
9
10 int Edge::getSource() {
11     return this->source;
12 }
13
14 int Edge::getDest() {
15     return this->dest;
16 }
17
18 int Edge::getCapacity() {
19     return this->capacity;
20 }
21
22 Edge::~~Edge() {
23 }
24
25 void Edge::setCapacity(int capacity) {
26     this->capacity = capacity;
27 }
```

Listing 10: Edge.cpp

```
1 #include "EdgeInfo.h"
2
3 EdgeInfo::EdgeInfo(Edge edge, int flow, int capacity) {
4     this->edge = edge;
5     this->flow = flow;
6     this->capacity = capacity;
7 }
8
9 int EdgeInfo::getCapacity() {
10     return this->capacity;
11 }
```

```

12
13 int EdgeInfo::getFlow() {
14     return this->flow;
15 }
16
17 int EdgeInfo::getResidualCapacity() {
18     return this->capacity - this->flow;
19 }
20
21 void EdgeInfo::setFlow(int newFlow) {
22     this->flow = newFlow;
23 }

```

Listing 11: EdgeInfo.cpp

```

1 #include "Flow.h"
2
3 Flow::Flow(Digraph digraph) {
4     this->digraph = digraph;
5     this->init();
6 }
7
8 void Flow::init() {
9     // f(e) = 0 para toda arista del grafo.
10    for (int v = 0; v < this->digraph->getVertices(); v++) {
11        std::list<Edge> list = this->digraph->getAdjList(v);
12        for (std::list<Edge>::iterator it = list->begin(); it <
13            != list->end(); ++it) {
14            Edge edge = *it;
15            int flow = 0;
16            EdgeInfo edgeInfo = new EdgeInfo(edge, flow, edge->
17                getCapacity());
18            this->edgeMap[edge] = edgeInfo;
19        }
20    }
21
22    EdgeInfo Flow::getEdgeInfo(Edge edge) {
23        return this->edgeMap[edge];
24    }
25
26    Flow::~~Flow() {
27        for (map<Edge, EdgeInfo>::iterator it = this->edgeMap->
28            begin(); it != this->edgeMap->end(); ++it) {

```

```

27         delete it->second;
28     }
29 }

```

Listing 12: Flow.cpp

```

1  #ifndef TRABAJOPRACTICO2FORDFULKENSON_H
2  #define TRABAJOPRACTICO2FORDFULKENSON_H
3
4  #include <vector>
5  #include "Edge.h"
6  #include "Digraph.h"
7  #include "Flow.h"
8  #include "ParserNetworkFlow.h"
9
10 using namespace std;
11
12 class FordFulkerson {
13 private:
14     ParserNetworkFlow parser;
15     Flow flow;
16     vector<int> setS;
17     vector<int> setT;
18
19     int bottleneck(vector<Edge > pathInResidualGraph);
20
21     void augment(vector<Edge > pathInResidualGraph);
22
23     bool isEdgeForward(Edge edge);
24
25     Edge getEdgeInG(Edge edgeInGResidual);
26
27     vector<Edge > getPathST();
28
29     void calculateMinCut();
30
31 public:
32
33     FordFulkerson(ParserNetworkFlow parser);
34     ~FordFulkerson();
35
36     void maxFlow();
37
38     vector<int> getSetS();

```

```

39     vector<int> getSetT();
40
41     vector<int> getProjects();
42     vector<int> getAreas();
43
44 };
45
46
47 #endif //TRABAJOPRACTICO2.FORDFULKERSON.H

```

Listing 13: FordFulkerson.h

```

1  #include "FordFulkerson.h"
2  #include "ParserNetworkFlow.h"
3  #include "PathBFS.h"
4
5  FordFulkerson::FordFulkerson(ParserNetworkFlow parser) {
6      this->parser = parser;
7      this->flow = new Flow(parser->graph);
8
9  }
10
11 FordFulkerson::~~FordFulkerson() {
12     delete flow;
13 }
14
15 /
16     Calcula el cuello de botella del camino pasado como ←
17     argumento
18 int FordFulkerson::bottleneck(vector<Edge > ←
19     pathInResidualGraph) {
20     int min = pathInResidualGraph.at(0)->getCapacity();
21     for (unsigned int i = 1; i < pathInResidualGraph.size(); i←
22         ++i) {
23         int residualCapacity = pathInResidualGraph.at(i)->←
24             getCapacity();
25         if (residualCapacity < min) {
26             min = residualCapacity;
27         }
28     }
29     return min;
30 }

```

```

29
30
31 Edge FordFulkerson::getEdgeInG(Edge edgeInGResidual) {
32     return this->parser->mapping.mapEdgeResidualGToEdgeG[←
        edgeInGResidual];
33 }
34
35
36 bool FordFulkerson::isEdgeForward(Edge edgeInGResidual) {
37     return this->parser->mapping.isEdgeResidualForward[←
        edgeInGResidual];
38 }
39
40 void FordFulkerson::augment(vector<Edge > pathInResidualGraph)←
    {
41     int b = this->bottleneck(pathInResidualGraph);
42     for (vector<Edge >::iterator it = pathInResidualGraph.begin←
        (); it != pathInResidualGraph.end(); ++it) {
43         Edge edgeInGResidual = *it;
44         Edge edgeInG = this->getEdgeInG(edgeInGResidual);
45         int newFlow;
46         if (this->isEdgeForward(edgeInGResidual)) {
47             newFlow = this->flow->getEdgeInfo(edgeInG)->getFlow←
                () + b;
48         } else {
49             newFlow = this->flow->getEdgeInfo(edgeInG)->getFlow←
                () - b;
50         }
51
52         this->flow->getEdgeInfo(edgeInG)->setFlow(newFlow);
53         pair<Edge ,Edge > edgesInResidualG = this->parser->←
            mapping.mapEdgeGToResidualG[edgeInG];
54         Edge edgeForward = edgesInResidualG.first;
55         Edge edgeBackward = edgesInResidualG.second;
56         edgeForward->setCapacity(edgeInG->getCapacity() - ←
            newFlow);
57         edgeBackward->setCapacity(newFlow);
58     }
59 }
60
61 vector<Edge > FordFulkerson::getPathST() {
62     int sVertex = 0;
63     int tVertex = this->parser->areasCount + this->parser->←
        projectsCount + 1;
64     vector<Edge > path;

```

```

65     PathBFS pathBFS(this->parser->residualGraph, sVertex, ←
        tVertex);
66     if (pathBFS.visited(tVertex)) {
67         std::list<Edge> pathList = pathBFS.pathTo(tVertex);
68         for (std::list<Edge>::iterator it = pathList.begin(); ←
            it != pathList.end(); ++it) {
69             Edge edge = *it;
70             path.push_back(edge);
71         }
72     }
73
74     return path;
75 }
76
77 void FordFulkerson::maxFlow() {
78     while (true) {
79         vector<Edge> path = getPathST();
80         if (path.empty()) break;
81         this->augment(path);
82     }
83
84     this->calculateMinCut();
85 }
86
87 void FordFulkerson::calculateMinCut() {
88     int sVertex = 0;
89     int tVertex = this->parser->areasCount + this->parser->←
        projectsCount + 1;
90
91     PathBFS pathBFS(this->parser->residualGraph, sVertex, ←
        tVertex);
92     for (int i = 1; i < this->parser->residualGraph->←
        getVertices() - 1; i++) {
93         if (pathBFS.visited(i)) {
94             this->setS.push_back(i);
95         } else {
96             this->setT.push_back(i);
97         }
98     }
99 }
100
101 vector<int> FordFulkerson::getSetT() {
102     return this->setT;
103 }
104

```



```

105 vector<int> FordFulkerson::getSetS() {
106     return this->setS;
107 }
108
109 vector<int> FordFulkerson::getProjects() {
110     vector<int> projectsToDo;
111     for (std::vector<int>::iterator it = this->setS.begin(); it↵
        != this->setS.end(); ++it ) {
112         VertexInfo vertexInfo = this->parser->vertexMap[( it)];
113         if (vertexInfo.isProject) {
114             projectsToDo.push_back(vertexInfo.number);
115         }
116     }
117     return projectsToDo;
118 }
119
120 vector<int> FordFulkerson::getAreas() {
121     vector<int> areasToPay;
122     for (std::vector<int>::iterator it = this->setS.begin(); it↵
        != this->setS.end(); ++it ) {
123         VertexInfo vertexInfo = this->parser->vertexMap[( it)];
124         if (!vertexInfo.isProject) {
125             areasToPay.push_back(vertexInfo.number);
126         }
127     }
128     return areasToPay;
129 }

```

Listing 14: FordFulkerson.cpp

```

1 #ifndef TRABAJOPRACTICO2.MAPPING.H
2 #define TRABAJOPRACTICO2.MAPPING.H
3
4 #include <map>
5 #include "Edge.h"
6
7 using namespace std;
8
9 // El mapeo se realiza para obtener mejores tiempos, y no estar↵
    creando el grafo residual en cada óiteracin del algoritmo
10 // de Ford-Fulkerson.
11 class Mapping {
12 public:
13     map<Edge , pair<Edge ,Edge>> mapEdgeGToResidualG;

```

```

14     map<Edge , Edge > mapEdgeResidualGToEdgeG;
15     map<Edge , bool> isEdgeResidualForward;
16
17     Mapping() {
18
19     }
20 };
21
22
23 #endif //TRABAJOPRACTICO2.MAPPING.H

```

Listing 15: Mapping.h

```

1 #include "ParserNetworkFlow.h"
2 #include "Digraph.h"
3 #include <fstream>
4 #include <vector>
5 #include <sstream>
6
7 ParserNetworkFlow::ParserNetworkFlow(string fileName) {
8     this->fileName = fileName;
9     this->parse();
10    this->createGraphs();
11 }
12
13 vector<int> ParserNetworkFlow::getAreaCosts(ifstream& file, int↵
14     areasCount) {
15     vector<int> costs;
16     int i = 0;
17     string line;
18     while (getline(file,line)) {
19         costs.push_back(stoi(line));
20         i++;
21         if (i == areasCount) break;
22     }
23     return costs;
24 }
25
26 int ParserNetworkFlow::calculateTotalCapacity(vector<int> v) {
27     int total = 0;
28     for (int i = 0; i < v.size(); i++) {
29         total += v.at(i);
30     }

```

```

31
32     return total;
33 }
34
35 pair<int, vector<int>> ParserNetworkFlow::splitLine(string line, ↵
    char delim) {
36     vector<int> dependencies;
37     stringstream ss;
38     ss.str(line);
39     string item;
40
41     getline(ss, item, delim);
42     int profit = stoi(item);
43
44     while (getline(ss, item, delim)) {
45         if (!item.empty()) {
46             dependencies.push_back(stoi(item));
47         }
48     }
49
50     return pair<int, vector<int>>(profit, dependencies);
51 }
52
53 void ParserNetworkFlow::getProjects(ifstream& file, int count) ↵
{
54     int i = 0;
55     string line;
56     while (getline(file, line)) {
57         pair<int, vector<int>> split = splitLine(line,    );
58         this->profits.push_back(split.first);
59         this->projectDependencies.push_back(split.second);
60
61         i++;
62         if (i == count) break;
63     }
64 }
65
66
67 void ParserNetworkFlow::parse() {
68     ifstream file = ifstream(this->fileName.c_str());
69     if (!file.is_open()) {
70         throw string("Error al abrir el archivo de flujo ") + ↵
            this->fileName;
71     }
72     string line;

```

```

73
74     getline(file, line);
75     this->areasCount = stoi(line);
76
77     getline(file, line);
78     this->projectsCount = stoi(line);
79
80     this->areaCosts = getAreaCosts(file, this->areasCount);
81
82     getProjects(file, this->projectsCount);
83     this->infinityCapacity = calculateTotalCapacity(this->profits) + 1;
84
85     file.close();
86 }
87
88
89 void ParserNetworkFlow::createGraphs() {
90     this->graph = new Digraph(this->projectsCount + this->areasCount + 2);
91     this->residualGraph = new Digraph(this->projectsCount + this->areasCount + 2);
92
93     int sVertex = 0;
94     int tVertex = this->projectsCount + this->areasCount + 1;
95
96     for (int i = 0; i < this->projectsCount; i++) {
97         // Agrego aristas s -> projects
98         int projectVertex = i + 1;
99         Edge edgeG = new Edge(sVertex, projectVertex, this->profits.at(i));
100         Edge edgeForward = new Edge(sVertex, projectVertex, this->profits.at(i));
101         Edge edgeBackward = new Edge(projectVertex, sVertex, 0);
102         this->mapping.isEdgeResidualForward[edgeForward] = true;
103         this->mapping.isEdgeResidualForward[edgeBackward] = false;
104         this->mapping.mapEdgeResidualGToEdgeG[edgeForward] = edgeG;
105         this->mapping.mapEdgeResidualGToEdgeG[edgeBackward] = edgeG;
106
107         this->mapping.mapEdgeGToResidualG[edgeG] = pair<Edge,

```

```

108         Edge >(edgeForward, edgeBackward);
VertexInfo vertexInfo("Projecto:" + to_string(i+1), i↵
    +1, true);
109     this->vertexMap[projectVertex] = vertexInfo;
110
111     this->graph->addEdge(edgeG);
112     this->residualGraph->addEdge(edgeForward);
113     this->residualGraph->addEdge(edgeBackward);
114 }
115
116 for (int i = 0; i < this->areasCount; i++) {
117     // Agrego aristas areas-> t
118     int areaVertex = this->projectsCount+i+1;
119     Edge edgeG = new Edge(areaVertex, tVertex, this->↵
        areaCosts.at(i));
120     Edge edgeForward = new Edge(areaVertex, tVertex, this↵
        ->areaCosts.at(i));
121     Edge edgeBackward = new Edge(tVertex, areaVertex, 0);
122     this->mapping.isEdgeResidualForward[edgeForward] = true↵
        ;
123     this->mapping.isEdgeResidualForward[edgeBackward] = ↵
        false;
124     this->mapping.mapEdgeResidualGToEdgeG[edgeForward] = ↵
        edgeG;
125     this->mapping.mapEdgeResidualGToEdgeG[edgeBackward] = ↵
        edgeG;
126
127     this->mapping.mapEdgeGToResidualG[edgeG] = pair<Edge ,↵
        Edge >(edgeForward, edgeBackward);
128     VertexInfo vertexInfo("Area:" + to_string(i+1), i+1, ↵
        false);
129     this->vertexMap[areaVertex] = vertexInfo;
130
131     this->graph->addEdge(edgeG);
132     this->residualGraph->addEdge(edgeForward);
133     this->residualGraph->addEdge(edgeBackward);
134 }
135
136 for (int i = 0; i < this->projectsCount; i++) {
137     // Agregado de dependencias
138     int projectVertex = i + 1;
139     vector<int> dependencies = this->projectDependencies.at↵
        (i);
140     for (vector<int>::iterator it = dependencies.begin(); ↵
        it != dependencies.end(); ++it) {

```

```

141         int areaVertex = this->projectsCount + ( it); // Se
           asume que el archivo indica las areas a partir
           de 1.
142
143         Edge  edgeG = new Edge(projectVertex, areaVertex,
           this->infinityCapacity);
144         Edge  edgeForward = new Edge(projectVertex,
           areaVertex, this->infinityCapacity);
145         Edge  edgeBackward = new Edge(areaVertex,
           projectVertex, 0);
146         this->mapping.isEdgeResidualForward[edgeForward] =
           true;
147         this->mapping.isEdgeResidualForward[edgeBackward] =
           false;
148         this->mapping.mapEdgeResidualGToEdgeG[edgeForward] =
           edgeG;
149         this->mapping.mapEdgeResidualGToEdgeG[edgeBackward] =
           edgeG;
150
151         this->mapping.mapEdgeGToResidualG[edgeG] = pair<
           Edge ,Edge >(edgeForward, edgeBackward);
152
153         this->graph->addEdge(edgeG);
154         this->residualGraph->addEdge(edgeForward);
155         this->residualGraph->addEdge(edgeBackward);
156     }
157 }
158
159 }

```

Listing 16: ParserNetworkFlow.cpp

```

1
2 #include "PathBFS.h"
3
4 #include <queue>
5
6 using namespace std;
7
8 PathBFS::PathBFS(Digraph g, int source, int dest ) : Path(g,
           source, dest ){
9
10     for(int i=0;i<g->getVertices();i++) {
11         marked[i] = false;

```

```

12     distance[i] = Path::NO_PATH; //Inicializo ←
        distancias con distancia infinito
13 }
14
15 queue<int> queue;
16
17 marked[this->source] = true;
18 distance[this->source] = 0;
19
20 queue.push(this->source); //arranco desde el source
21
22 //Mientras haya un vertice pendiente y no se haya encontrado←
    el destino
23 while(!queue.empty() && !marked[this->dest]){
24     int v = queue.front();
25     queue.pop();
26     list<Edge > adjList = g->getAdjList(v);
27
28     //Barro la lista de adyacencia
29     for (std::list<Edge >::iterator it=adjList->begin(); it ←
        != adjList->end(); ++it){
30         // Agregado adicional para no procesar las aristas ←
            invalidas del grafo residual.
31         if (( it)->getCapacity() <= 0) continue;
32
33         int vert = ( it)->getDest();
34
35         if(!marked[vert]){
36             distance[vert] = distance[v] + 1; // Acumulo la ←
                distancia desde el origen hasta el vertice
37             marked[vert]= true;
38             queue.push(vert);
39             edgeTo[vert] = it; //guardo el camino por el ←
                que fui
40         }
41     }
42 }
43
44 }
45
46 PathBFS::~~PathBFS() {
47
48 }

```

Listing 17: PathBFS.cpp

```

1 #ifndef TRABAJOPRACTICO2.VERTEXINFO.H
2 #define TRABAJOPRACTICO2.VERTEXINFO.H
3
4
5 class VertexInfo {
6 public:
7     VertexInfo() {
8     }
9
10    VertexInfo(string description, int number, bool isProject) ↵
11    {
12        this->description = description;
13        this->isProject = isProject;
14        this->number = number;
15    }
16
17    string description;
18    bool isProject = false;
19    int number = 0;
20 };
21
22 #endif //TRABAJOPRACTICO2.VERTEXINFO.H

```

Listing 18: VertexInfo.h

Referencias

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