Asignación de vuelos a pilotos

Algoritmia - FIB - UPC

Q1 OTOÑO

Henry Qiu, Jaime Arroyo, Josué Alcántara

January 9, 2018

Contents

0.1 El problema 2
0.2 Algunas consideraciones 2
0.3 Formalización 3
0.4 Red de flujo entrada a MaxFlow 3
0.5 Circulación bajo demanda y capacidades mínima 4
0.6 Interpretación 6
0.7 Implementación 6
0.7.1 Estructura del proyecto 6
0.7.2 Análisis costes temporales 12
0.8 Tiempos de ejecución 13
0.8.1 Diagrama estructural del proyecto 14
0.8.2 Línea de ejecución del programa 15

0.1 El problema

Se nos plantea el problema de asignación óptima de pilotos a vuelos de en una compañía aérea en un día. En otras palabras, tenemos que encontrar el número de pilotos mínimo para servir todos los viajes de la compañía.

Para cada viaje tenemos su destino, origen, tiempo de inicio y tiempo de llegada. Por facilidad, los tiempos serán expresados en minutos. Como nos restringimos a un día, los tiempos serán enteros en el rango $0..(24 * 60)$.

Cada viaje será representado por 4 enteros de la siguiente forma: $(o_i, d_i, h_{1_i}, h_{2_i})$ donde o_i es el origen, d_i es el destino, h_{1_i} es la hora de inicio y h_{2_i} es la hora de llegada.

0.2 Algunas consideraciones

- Cada vuelo tiene exactamente un piloto asignado, que es el que lo pilota.
- Un piloto solo puede pilotar aquellos viajes que le han sido asignados.
- Por simplicidad, solo tendremos en cuenta que los viajes asignados a un piloto no se pueden solapar en el tiempo.
- Dos viajes son alcanzables si des del destino del primero puedo llegar al origen del segundo cumpliendo unas condiciones previas. En este proyecto tendremos dos versiones que nos definen si dos vuelos son alcanzables o no. Propondremos implementaciones para ambas versiones.
- Decimos que k pilotos son suficientes para realizar todos los vuelos si se puede asignar un piloto a cada vuelo de manera factible, cumpliendo las consideraciones anteriores.

0.3 Formalización

Formalicemos un poco mas el problema. Tenemos n trayectos $T = T1, T2, ..., Tn$. Cada trayecto es una tupla $Ti = (o_i, d_i, h_{1_i}, h_{2_i})$ donde o_i es el origen, d_i es el destino, h_{1_i} es la hora de inicio y h_{2_i} es la hora de llegada.

Por ejemplo, con 3 ciudades (Barcelona (BCN), París (CDG), Londres (LGW)) podríamos tener el siguiente conjunto de 6 trayectos,
$T1 = (BCN, CDG, 0, 100), T2 = (CDG, BCN, 450, 550), T3 = (CDG, LGW, 150, 250)$,
$T4 = (LGW, CDG, 600, 700), T5 = (LGX, CDG, 300, 400), T6 = (CDG, BCN, 750, 850)$.

La entrada del ejemplo anterior sería la siguiente:

0 1 0 100
1 0 450 550
1 2 150 250
2 1 600 700
2 1 300 400
1 0 750 850

0.4 Red de flujo entrada a MaxFlow

Se nos propone solucionar el problema siguiendo el diseño planteado en el libro *Algorithm Design - Kleinberg and Tardos*, explicado a continuación.

Un algoritmo eficiente para resolver el problema de asignación óptima de pilotos a vuelos está basada en redes de flujo. Se trata de diseñar la siguiente red:

Conjunto de nodos

- Para cada vuelo i (u_i, v_i) añadimos dos nodos enlazados con una arista de capacidad máxima y mínima de 1. De esta forma obligamos a que el vuelo i sea atendido.
- Tendremos un nodo fuente s y el sumidero t.

Conjunto de aristas

- Para cada vuelo i, hay una arista (u_i, v_i) de capacidad máxima y mínima 1. (Obligamos a que el vuelo i sea asignado).
- Para cada vuelo i y j, si j es alcanzable desde i entonces añadimos una arista (v_i, u_j) de capacidad máxima 1 y mínima 0. (El mismo piloto puede realizar los viajes i y j).
- Para cada vuelo i hay una arista (s, u_i) de capacidad máxima 1 y mínima 0.
- Para cada vuelo i hay una arista (v_i, t) de capacidad máxima 1 y mínima 0.
- Tenemos una arista (s, t) de capacidad máxima k y mínima 0. (Si tenemos pilotos extra no los necesitamos).

Finalmente, el nodo s tendrá una demanda de -k, mientras que el nodo t la tendrá de k. El resto de nodos tendrá una demanda de 0.

Se puede notar que tendremos un grafo resultante con demandas y capacidades mínimas. Por tanto, hemos reducido nuestro problema a un problema de flujo bajo circulación de demandas y capacidades mínimas. En las siguiente sección vamos a ver como podemos reducir este problema a un problema de flujo, que prodemos usar como entrada a *MaxFlow*.

0.5 Circulación bajo demanda y capacidades mínima

Dado un grafo $G = (V, E)$ con $c(e)$, $c(e) >= l(e) >= 0$ para cada arista $e \in E$ y $d(v)v \in V$, definimos la circulación como una función $f : E \to R^+$ tal que:

1. *Criterio de capacidad*: Para toda $e \in E$,

$$l(e) \leq f(e) \leq c(e)$$

2. *Criterio de conservación* Para todo $v \in V$

En este tipo de problemas lo que queremos encontrar es si existe una circulación factible en G que satishaga las dos condiciones anteriores. Para encontrar una circulación factible en G tenemos que hacer las siguientes transformaciones para eliminar las demandas y las capacidades mínimas de G, obteniendo un nuevo grafo G':

1. *Eliminar capacidades mínimas:* para cada $e = (u, v) \in E$,

   a) Actualizar las capacidades

$$c' = c(e) - l(e)$$

   b) Actualizar demandas

$$d'(u) = d(e) + l(e)$$
$$d'(v) = d(e) - l(e)$$

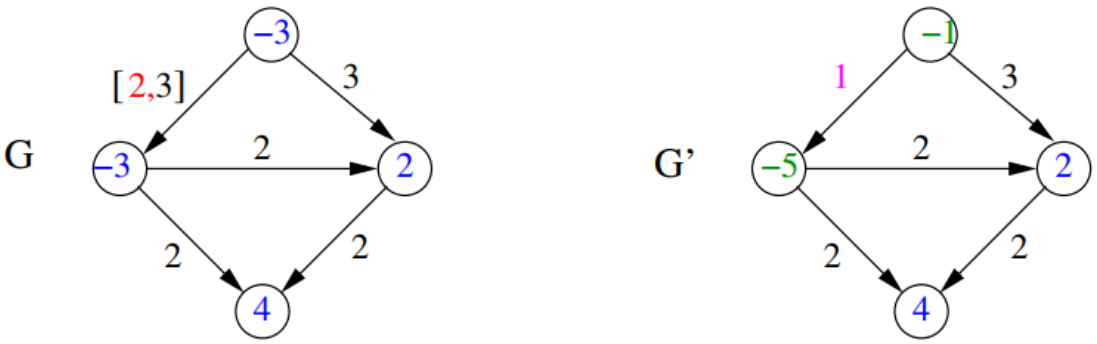

Figure 1: Ejemplo eliminación de capacidades mínimas

2. *Eliminar las demandas:*

   a) Añadir una nueva fuente s y un nuevo drenador t.

   b) Para cada $v \in V$ tal que $(d(v) < 0)$ añadir la arista (s, v) con capacidad $-d(v)$.

   c) Para cada $v \in V$ tal que $(d(v) > 0)$ añadir la arista (v, t) con capacidad $d(v)$.

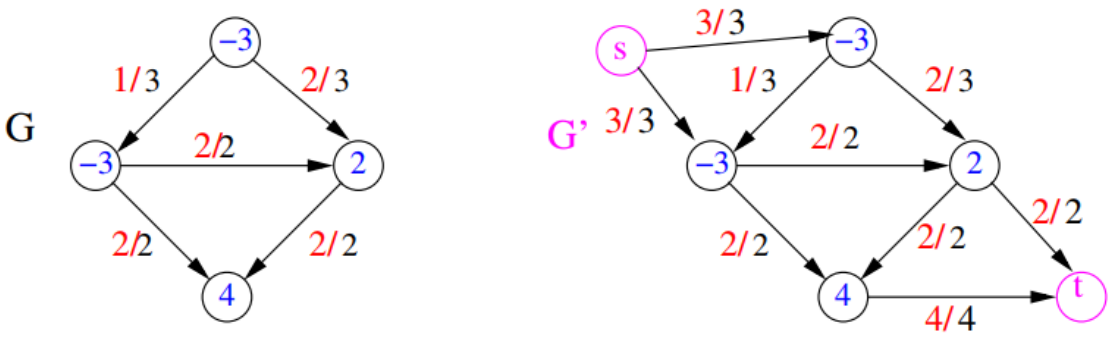

Figure 2: Ejemplo eliminación de demandas

Por tanto, juntanda los dos pasos anteriores, la red de flujo resultante de entrada a MaxFlow sería la siguiente:

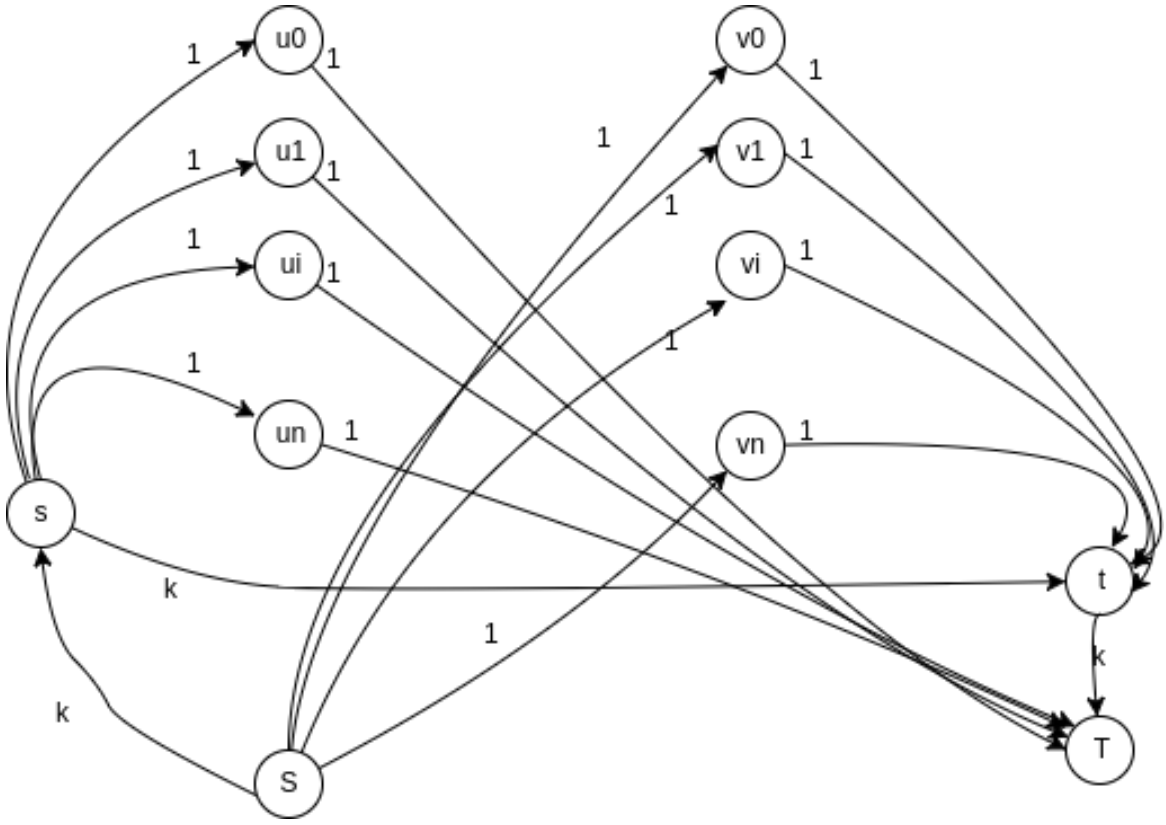

Figure 3: Red de flujo resultante

0.6 Interpretación

Hasta ahora hemos visto como generar la red de flujo a partir de la lista de vuelos. En este apartado vamos a ver como se interpretamos el resultado obtenido al ejectura el MaxFlow.

Algunas consideraciones previas:

- k es el número de pilotos máximo que podemos utilizar, en otras palabras, el flujo máximo que puede tener la red.
- f es el flujo que pasa por la arista (s, t). Es importante este valor porque por esta arista pasan los pilotos que hemos descartado, es decir, los que no utilizamos

A partir de las considereciones anteriores, se puede decucir que el número de pilotos mínimos que pueden realizar todos los vuelos de la lista dada es:

$$min_{pilots} = k - f$$

0.7 Implementación

En las siguiente secciones vamos a explicar la implementación realizada a partir de las explicaciones dadas en las secciones anteriores.

0.7.1 Estructura del proyecto

Por comodidad y para tener un código más usable y adaptable a cambios, el proyecto tiene una estructura modular basada en clases. A continuación se explica cada clase o módulo con sus métodos y atributos juntamente con su funcionalidad específica.

Módulo airShe.cpp

Este módulo tiene el hilo de programa principal. La explicación de su funcionamiento es la siguiente:

Listing 1: airSche usage

```
Usage: running a scheduling airline problem solver.
The correct usage is:

                ./airSche vx max_flow_algorithm ?all
 where
* vx is the reacheable version to run, x must be 1 or 2,
 depending on the reachable version.
* max_flow_algorithm must be:
* edmonds: run using Edmonds Karp, using BFS.
    * ford: run using Ford Fulkerson, using DFS.
* all: is not mandatory, you have to set this param
if you want to print all pilots rutes.
```

Como podemos ver, se le puede pasar hasta un máximo de tres parámetros, donde los dos primeros son obligatorios y el último opcional, dependiendo de si queremos ver la lista de vuelos de cada piloto.

Módulo load [load.hh - load.cpp]

Clase madre de las clases LoadV1 y LoadV2. Tiene definido el struct Trip que nos permite representar lo que es medio vuelo. Para tener todo el vuelo necesitamos dos instancias de este struct, una instancia marcada como origen y con el origen marcado a falso.

Listing 2: Trip struct

```
struct Trip{
            int airport;
            int time;
            bool origen;

            Trip(){}
            Trip(int airport, int time, bool origen){
                this->airport = airport;
                this->origen = origen;
                this->time = time;
            }
        };
```

Esta clase tiene definido los siguientes métodos:

- **void usage()**: explica la forma de lanzar el programa en caso de fallo.
- **pair< vector<vector<int> >,vector<vector<int> > > load()**: define la función de lectura de los vuelos, diferente para cada versión. Devuelve una tupla en el que:
  - La primera posición corresponde con una matriz n*n (n = número de vuelos * 2 + 4) que representa el grafo G.
  - La segunda posición corresponde a una matriz que representa la lista de adyacencias de cada nodo.
- **bool reachable(const Trip &t1, const Trip &t2)**: para dos vuelos dados, t1 y t2, tal que t1 es origen, nos dice si podemos alcanzar el origen de t2 desdel destino de t1 teniendo en cuenta la diferencia de tiempo de 15 minutos entre la llegada de t1 y la salida de t2. Esta función es útil para generar las aristas en el grafo final de entrada a MaxFlow.

Módulo Loadv1 [load_v1.hh - load_v1.cpp]

Como hemos comentado anteriormente, este módulo hereda de la clase Load. En este módulo viene implementada la creación del grafo a partir de la definición de la versión 1 del problema. Tiene los siguientes métodos:

- **pair< vector<vector<int> >,vector<vector<int> > > load()**: define la función de carga para la versión 1.
- **void load_recheable_v1(vector<vector<int> > &E, vector<vector<int> > &C, const vector<Trip> &trips)**: extensión de la función load, nos crea la tupla resultante para el load() utilizando la definición de la versión 1, a partir de una lista de vuelos.

Cabe explicar que para crear las aristas el orden es importante. En nuestra implementación usamos un vector de vectores para guardar para cada nodo, una lista de nodos a la que puede acceder haciendo pushback, esto hace que dependiendo de qué nodos guardemos primero, en el algoritmo de Edmonds Karp al encontrar el "shortest augmenting path" el resultado dé cosas diferentes.

Para empezar, el arco que no nos importa en qué momento crearlo es el que va de t a T, ya que desde t solo sale un nodo único a T.

Posteriormente, es importante crear primero los arcos que van desde cada nodo destino a cada nodo origen alcanzable según la versión que sea. Esto es debido a que si creamos primero los arcos de cada destino al nodo t, el algoritmo cuando busque el "shortest augmenting path" siempre cogerá como resultado el camino desde S al nodo destino al nodo t al nodo T, y de esta manera nunca nos dará que un mismo piloto pueda hacer más de un viaje. También cabe decir que de los orígenes alcanzables el orden también es importante, hay que tener los orígenes alcanzables en orden creciente respecto al tiempo de inicio.

Una vez hecho lo más importante, podemos crear las asociaciones restantes en una misma iteración , donde el nodo s se une a cada origen, cada nodo origen se une a T, cada nodo destino se une a t, el nodo S a cada nodo destino. Y finalmente, el nodo S a s, y el nodo s a t.

Módulo Loadv2 [load_v2.hh - load_v2.cpp]

Como hemos comentado anteriormente, este módulo hereda de la clase Load. En este módulo viene implementada la creación del grafo a partir de la definición de la versión 2 del problema. Tiene los siguientes métodos:

- **pair< vector<vector<int»,vector<vector<int» > load()**: define la función de carga para la versión 1.
- **void load_recheable_v2(vector<vector<int> > &E, vector<vector<int> > &C, const vector<Trip> &trips)**: extensión de la función load, nos crea la tupla resultante para el load() utilizando la definición de la versión 2, a partir de una lista de vuelos.
- **vector<int> BFSReach(vector<vector<int> > &E, int i)**: partiendo del grafo de vuelos con todas las aristas reachables según la V1 ya añadidas, utilizamos la búsqueda en anchura partiendo desde cada nodo "destino" y guardamos en el vector result todos los nodos "origen" que visitamos durante la ejecución del BFS. Estos són los nodos reachables desde el destino sobre el que estamos haciendo la búsqueda y todos cumplen el tiempo mínimo de descanso entre vuelo y vuelo ya que todas las aristas por las que pasamos en el algoritmo lo cumplen.

La diferencia entre las dos versiones es que esta versión dos permite más libertad en la asignación de aristas "alcanzables" desde un nodo destino a un nodo origen. Esto quiere decir que el número de aristas desde un destino a un origen en la versión 2 es mayor o igual que el número de aristas desde un destino a un origen en la versión 1. Esto permite que sea más posible que un mismo piloto pueda realizar más vuelos ya que hay más aristas alcanzables desde un destino a un origen, como consecuencia, el número mínimo de pilotos que se necesita para cubrir los vuelos será igual o menor en la versión 2 respecto a la versión 1. Como penalización a esto, és un poco más costoso crear y trabajar con el grafo de la versión 2 que la versión 1 ya que puede tener muchas más aristas.

Módulo Results [results.hh - results.cpp]

Tal como se puede interpretar del nombre del módulo, este módulo sirve para mostrar los resultados. Tiene un único método, definido de la siguiente forma:
void print_paths(const pair<int,vector<vector<int> > > &pilots, const vector<vector<int> >&E, const vector<vector<int> >&C, bool all)

Descripción de los parámetros El método admite los siguiente parámetros.

- pilots: una tupla donde:
  - El primer valor corresponde al número de pilotos mínimos para realizar la lista de vuelos de la entrada. Sea f el máximo flujo que puede ir por la red G y sea k el flujo que pasa por la arista (s,t) [por esta arista pasa el flujo [pilotos] que no se necesita para completar todos los vuelos], entonces el valor de este parámetro es: f - k.
  - El segundo valor corresponde a la red resultante de aplicar MaxFlow.
- E: representa la lista de adyacencias de cada noco.
- C: representa la red de flujo únicamente con sus capacidades.
- all: es un booleano que nos indica si deseamos imprimir la lista de vuelos que realiza cada piloto.

Funcionamiento A continuación se explica el funcionamiento de este método:
El algoritmo consiste hacer k iteraciones y en cada iteración enseñar por pantalla todos los los viajes que realizará el mismo piloto, donde k representa el número de pilotos mínimos que asignaremos para realizar los viajes.

Para comprobar cuáles son los vuelos que han hecho cada piloto recorremos empezando desde el primer piloto, que siempre realizará el primer viaje. Gracias a la matriz E y la red resultante guardada en pilots, en el nodo destino de cada viaje consultamos cuál es la arista que tiene flujo 1 (con la comprobación while (pilots.second[act][t] != 1)), en caso de que la arista con flujo 1 sea la que va hacia t, sabemos que allí se acaba el trayecto de ese piloto, en caso contrario, tenemos que mirar cuáles de las aristas del que van hacia algún destino es el que tiene flujo 1, ese será el nodo origen del siguiente vuelo que cogerá, y así iteramos hasta que se acaben los vuelos que ese piloto tiene asignado.

Una vez acabado el recorrido de un piloto, para saber cuál es el primer viaje asignado al siguiente piloto se recorre el vector done, que guarda si un viaje está hecho por algún piloto, hasta encontrar el primer viaje no hecho.

Explicación de las variables Cada variable creada en el método tiene la siguiente funcionalidad:

- *done*: vector de mida número de viajes totales que guarda en cada posición si ese viaje está hecho por algún piloto.
- *act*: variable auxiliar que indica el número de nodo actual en la que estoy tratando.
- *viajeaux*: variable auxiliar que indica el número del primer viaje que recorrerá cada piloto.

Módulo FordFulkerson [ford_fulkerson.hh - ford_fulkerson.cpp]

En este módulo está implementada la solución propuesta para resolver MaxFlow utilizando el algoritmo Ford Fulkerson.

El algoritmo Ford Fulkerson se centra en ir aumentando el flujo de la red de flujo mientras exista un camino aumentador de s a t. Un camino aumentador de s a t es un camino P por el cual s puede mandar alguna unidad de flujo hasta t sin superar la condición de capacidad máxima de capacidad de las arista que forman P.

Ford Fulkerson sigue el siguiente pseudocódigo:

Listing 3: Ford Fulkerson pseudocódigo

```
for each edge (u,v) set f(u,v)=0 and f(v,u)=0
while there is an augmenting path p from s to t {
        cf = minimal c(u,v)-f(u,v) from all edges in p
        for all edges (u,v) in p {
                f(u,v)=f(u,v)+cf
                f(v,u)=f(v,u)-cf
        }
}
```

La gracia de Ford Fulkerson es que para encontrar el camino aumentador existen varias técnicas. Alguna de estas técnicas se basan en utilizar un DFS o BFS. En el caso de utilizar un BFS, el algoritmo se etiqueta como Edmonds Karp. En la siguiente sección se explica esta versión.

Dado a que utilizar un BFS corresponde a implementar Edmonds Karp, en esta solución propuesto hemos utilizado un DFS para encontrar un camino aumentador de s a t. En este caso, una vez encontramos el camino aumentador P, decimos que la flujo máximo que se puede transmitir por P es 1, esto nos facilita la implementación del DFS.

En definitiva, en este módulo podemos encontrar lo siguiente:

Atributos Esta es la lista de atributos.

- vector<vector<int> >C: representa las capacidades de la red de flujo.
- vector<vector<int> >E: representa la lista de adyacencias de cada nodo.
- vector<vector<int> >F: representa la red de flujo con todos los flujos de las aristas inicializados a 0.
- int n: representa el número de nodos de la red de flujos.
- vector<bool> visited: vector de booleanos de tamaño n, útil durante la ejecución del DFS. Cada posición i nos marca si hemos visitado el nodo i durante el DFS.
- int k: indica el valor máximo de pilotos que podemos utilizar.

Métodos Esto son los métodos implementados:

- **FordFulkerson(const vector<vector<int> > &C, const vector<vector<int> > &E)**: método para crear una instancia de la clase. El nombre de los parámetros corresponde al nombre de los atributos.
- **pair<int, vector<vector<int> > > max_flow()**: ejecuta el algoritmo de MaxFlow sobre la red dada. Utilizamos su valor retornado como entrada al método results.print_paht. Devuelve la tupla que necesita la clase Results para mostrar el resultado.
- **bool is_path(int _s, int _t)**: función que implementa el DFS. Nos indica si en la red de flujos existe un camino P de _s a _t. En caso de que P exista, aumentaremos el flujo en 1 para cada arista e =(u,v) y actualizaremos la red residual con el valor del flujo añadido.

Módulo Edmondskarp [edmonds_karp.hh - edmonds_karp.cpp]

En este módulo está implementada la solución propuesta para resolver MaxFlow utilizando el algoritmo Edmonds Karp. Como hemos comentado anteriormente, Edmonds Karp es una versión de Ford Fulkerson con la principal diferencia que Edmonds Karp utiliza un BFS para encontrar el camino aumentador. Se basa en el siguiente pseudocódigo:

Listing 4: Edmonds Karp pseudocódigo

```
algoritmo EdmondsKarp
    entrada:
        C[1..n, 1..n] (matrices de capacidad)
        E[1..n, 1..?] (lista de vecinos)
        s             (origen)
        t             (hundimiento)
    salida:
        f             (valor del flujo maximal)
        F             (Una matriZ que da un flujo valido con el
                      maximo valor)
    f := 0 (el flujo inicialmente es cero)
    F := array(1..n, 1..n) (Capacidad del residuo de u a v es
                        C[u,v] - F[u,v])
    forever
        m, P := BFS(C, E, s, t, F)
        if m = 0
            break
        f := f + m
        (Busqueda usando backtracking y estrictura del flujo)
        v := t
        while v != s
            u := P[v]
            F[u,v] := F[u,v] + m
            F[v,u] := F[v,u] - m
            v := u
    return (f, F)

algoritmo BFS
    entrada:
        C, E, s, t, F
    salida:
        M[t]          (Capacidad del camino encontrado)
        P             (Tabla de padres)
    P := array(1..n)
```

```
    for u in 1..n
        P[u] := -1
    P[s] := -2 (asegurarse de que el 'origen' no sea reemplazado)
    M := array(1..n) (Capacidad del camino encontrado hasta el nodo)
    M[s] := INF
    Q := queue()
    Q.push(s)
    while Q.size() > 0
        u := Q.pop()
        for v in E[u]
            (Si hay capacidad disponible, y v no fue usado antes
            en la busqueda)
            if C[u,v] - F[u,v] > 0 and P[v] = -1
                P[v] := u
                M[v] := min(M[u], C[u,v] - F[u,v])
                if v != t
                    Q.push(v)
                else
                    return M[t], P
    return 0, P
[EXTRAIDO DE WIKIPEDIA]
```

En este caso, se puede notar que en cada iteración el flujo aumenta el mínimo posible en vez de ir uno a uno, como en el caso del Ford Fulkerson implementado. Esto nos permite que para casos en que la diferencia entre los pilotos mínimos que se necesitan y el número de vuelos es muy grande, la ejecución de Edmonds Karp sea bastante más eficiente que en el caso de Ford Fulkerson.

El método creador para este tipo de instancias recibe los mismos parámetros que el método creador de Ford Fulkerson.

0.7.2 Análisis costes temporales

Ambos métodos comparten la lectura de los datos y la creación del grafo, también como el printado de los resultados por pantalla. Tenemos que tener en cuenta que la creación del grafo en la versión 1 es $O(n * n * k)$, donde n es el número de vuelos y k es el número de orígenes alcanzables que tiene el nodo, ya que para cada destino miramos los orígenes de los n-1 vuelos restantes para comprobar si son alcanzables, y además para insertar el nuevo origen recorremos los orígenes que ya están añadidos para tenerlos ordenados. Para la versión 2 aún es más costosa, el coste es $O(n * n * k * (|V| + |E|))$, donde V es el número de nodos, y m el número de arcos, ya que ahora para cada nodo destino en vez de añadir solamente los orígenes directamente alcanzables, añadimos los todos orígenes alcanzables de estos utilizando un BFS.

A continuación nos centraremos en el análisi del coste de la implementación de los métodos.

Como hemos explicado anteriormente, para el método de Edmonds Karp el algoritmo que usamos para hacer la búsqueda del "augmenting path" es un BFS, que tiene coste $O(|V| + |E|^2)$, donde V es el número de nodos y E el número de arcos. Es debido a que el BFS tiene un coste temporal de $O(|V| + |E|)$ y necesitamos como máximo aumentar el camino $O(|E|)$ veces para saturar la red.

Para el método Ford Fulkerson el algoritmo que usamos para hacer la búsqueda del "augmenting path" es un DFS, que tiene coste $O(b^m)$, donde b es el factor de ramificación (número promedio de ramificaciones por nodo) y m es la máxima profundidad del espacio de estados.

0.8 Tiempos de ejecución

En esta sección se muestran los tiempos medios de ejecución para cada tipo de instancia. El script *generate_time_table.sh* es el encargado de calcular los tiempos medios de cada instancia y guardarlos en el archivo *times_table*1.*txt* o *times_table*2.*txt*, uno para cada versión.

Las siguientes figuras muestran gráficamente estos tiempos.

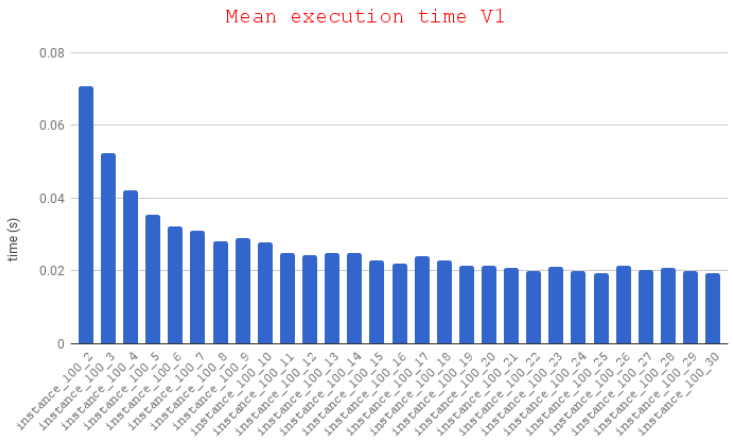

Figure 4: tiempo medio de ejecución v1

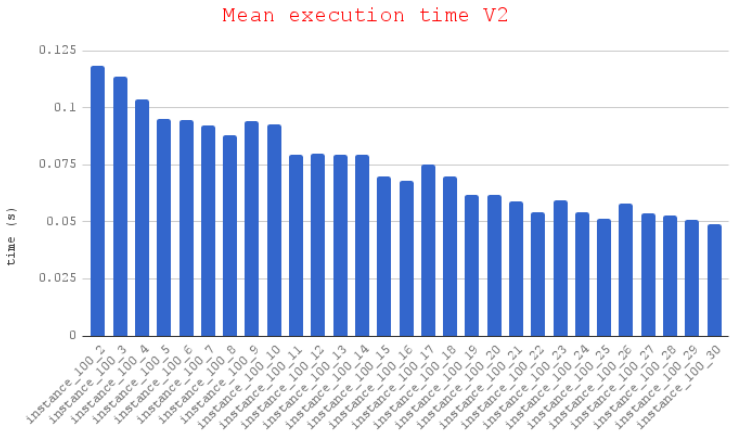

Figure 5: tiempo medio de ejecución v2

Se puede ver que para la versión dos los tiempos de ejecución son prácticamente el doble el la mayoriía de los casos. Esto es debido al tiempo de carga del grafo, es decir, el tiempo que tardamos en crear el grafo. Como explicamos anteriormente, en la versión 2 hacemos más calculos para encontrar los orígenes que son alcanzables desde otros destinos.

0.8.1 Diagrama estructural del proyecto

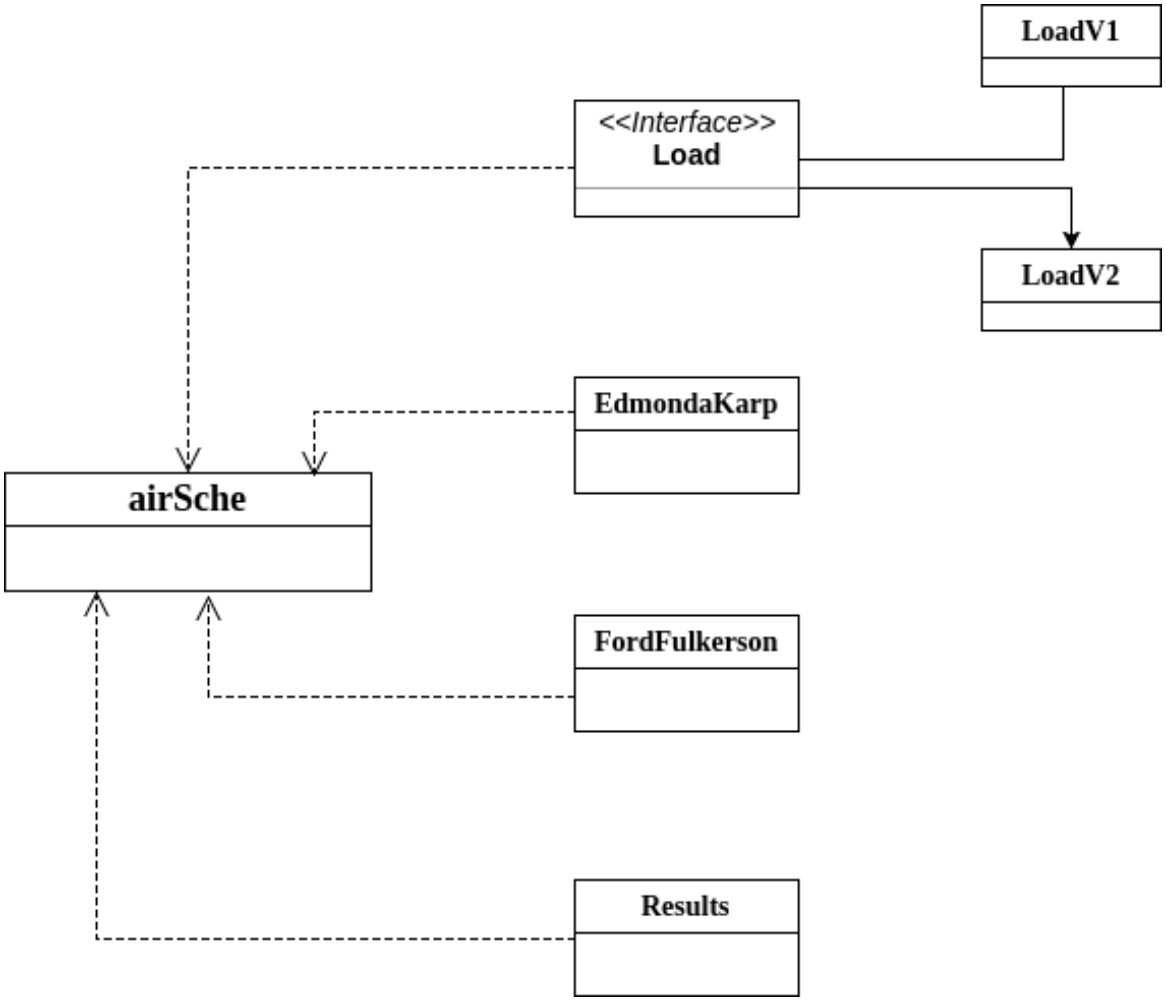

Figure 6: Diagrama estructural del proyecto

0.8.2 Línea de ejecución del programa

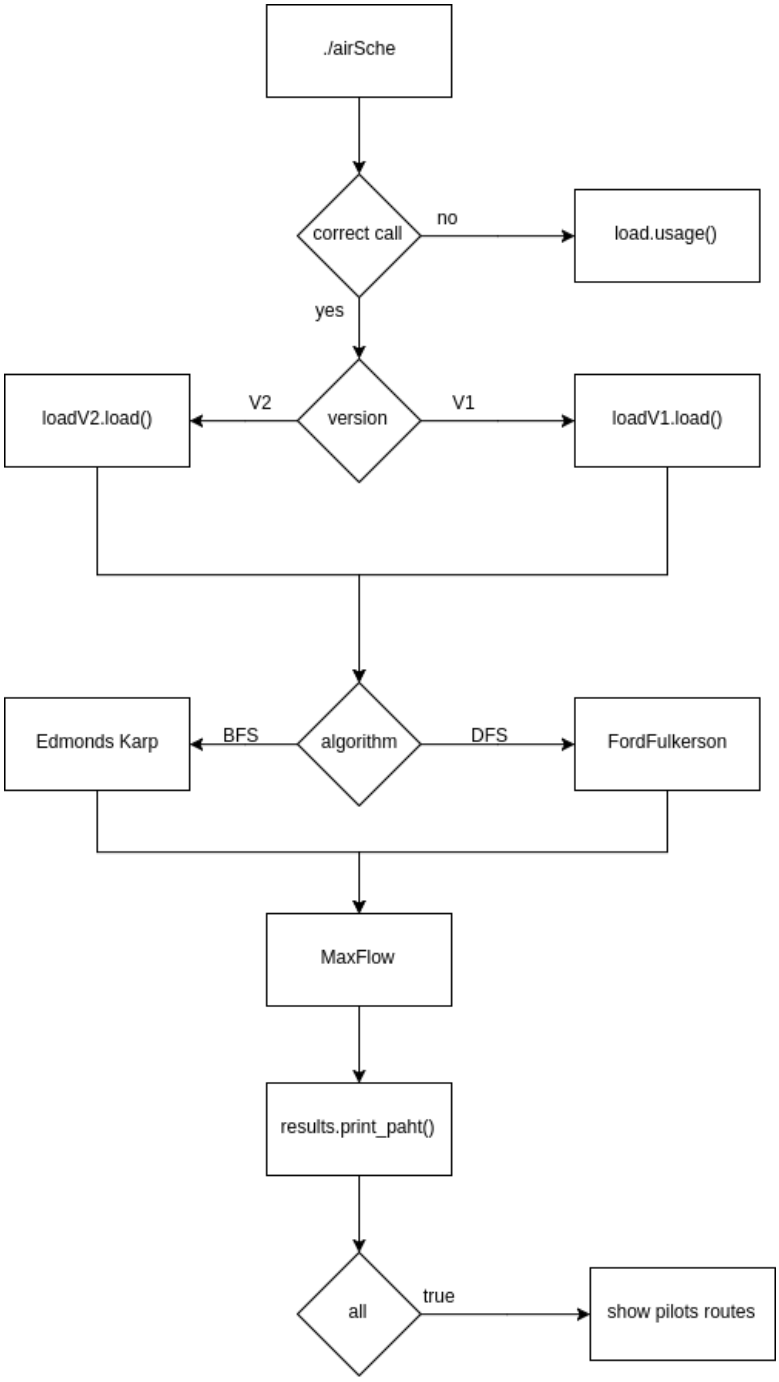

Figure 7: Línea de ejecución del programa

Bibliography

[1] Maxflow in O(mn)

[2] MaxFlow explanation

[3] Kleinberg and Tardos explanation

[4] Edmonds Karp code

[5] Ford Fulkerson wiki

[6] Ford Fulkerson code

[7] Blog thread

[8] Blog thread

[9] Blog thread

[10] Hungarian Algorith

[11] Kleinberg and Tardos, 2005 *Algorithm Design*