Trabajo de **F**in de **G**rado

ALGORITMOS PARA LA RESOLUCIÓN DE PROBLEMAS EN REDES

Alumno: Jorge Barelles Menes

DNI:45804337-K

Grado: Matemática Computacional

Tutores: Jorge Galindo Pastor y Alejandro Miralles Montolío

Supervisor: José Martínez Sotoca

Agradecimientos

El esfuerzo realizado por Jorge y Álex ha hecho realidad este trabajo. Por ello, quiero dar las gracias a ambos. También quiero agradecer a José todo el tiempo que ha depositado en mí, ya que sin él la estancia hubiese sido mucho más difícil.

ÍNDICE

1. Introducción

En este trabajo estudiaremos qué es la investigación operativa en redes y qué mecanismos podemos utilizar para resolver algunos de los problemas que se estudian en esta disciplina. Empezaremos definiendo los conceptos necesarios para poder entender el significado de red y el funcionamiento de los algoritmos más utilizados para resolver los problemas. A continuación, explicaremos cómo podemos utilizar el algoritmo de Ford-Fulkerson para solucionar el problema del flujo máximo. Posteriormente, pasaremos a estudiar el problema del árbol generador de coste mínimo y cómo podemos resolverlo mediante el algoritmo de Prim. El tercer, y último, problema de la investigación de operaciones en redes que estudiaremos, serà el del camino más corto. Una vez planteado, mostraremos, mediante el algoritmo de Dijkstra, su resolución. Para finalizar, expondremos tres problemas reales que solucionaremos paso a paso mediante los algoritmos estudiados.

2. La investigación operativa en redes

La investigación operativa es una moderna disciplina científica que se caracteriza por la aplicación de teoría, métodos y técnicas especiales, para buscar la solución de problemas de administración, organización y control que se producen en los diversos sistemas que existen en la naturaleza y los creados por el ser humano.

Nos tenemos que remontar hasta poco después de que estallara la segunda guerra mundial para ver los primeros indicios de esta disciplina, cuando la Bawdsey Research Station, bajo la dirección de A. P. Rowe, participó en el diseño del radar. Poco después este avance sirvió para analizar todas las fases de las operaciones nocturnas y con ello favorecer a la toma de decisiones.

Las principales fases que emplea esta disciplina son:

1. Examen de la situación real y recolección de la información.
2. Formulación del problema, identificación de las variables y elección de la función objetivo, para ser maximizada o minimizada.
3. Construcción del modelo matemático.
4. Resolución del modelo.
5. Análisis y verificación de las soluciones obtenidas.
6. Utilización y documentación del sistema obtenido para su posterior uso.

Gran parte de los problemas de esta disciplina pueden modelizarse y resolverse sobre una red. En nuestro trabajo nos vamos a centrar en los siguientes problemas: el flujo máximo, el árbol generador de coste mínimo y el camino más corto. El mecanismo para llevar a cabo la resolución de los problemas será el mismo para los tres: análisis, modelado, selección del algoritmo y ejecución del algoritmo.

3. Conceptos básicos

En este apartado, vamos a describir los elementos que forman una red. Con el propósito de que los teoremas y los algoritmos que explicaremos en los siguientes apartados se entiendan correctamente.

- Nodo: Es una representación de un elemento del problema.

- Arista: Es un par de nodos, de de la forma (n_1, n_2), que representa un enlace entre ellos. Existen dos tipos de arista:

    - Arista dirigida: Es un par **ordenado** de nodos con un sentido definido.

    - Arista no dirigida: Es un par **no ordenado** de nodos sin un sentido definido.

- Red: Es un par (N, E) en un conjunto X, donde N es un conjunto no vacío de puntos de X y E es un subconjunto de aristas, NxN, en el que no hay elementos repetidos ni de la forma (n_1, n_1). A una red se le pueden asociar funciones capaces de dar valores específicos a cada una de las aristas.

- Red no dirigida: Es una red cuyo subconjunto E está formado por aristas no dirigidas.

- Red dirigida: Es una red cuyo subconjunto de aristas E está formado por aristas dirigidas.

- Camino: Es una sucesión de al menos dos nodos, tal que para cada uno de sus nodos existe una arista que contiene a dicho nodo y al nodo sucesor. Un camino es de la forma:

$$P = \{n_{i_0},\ n_{i_1},\ n_{i_2},\ \dots,\ n_{i_{k-1}},\ n_{i_k}\}$$

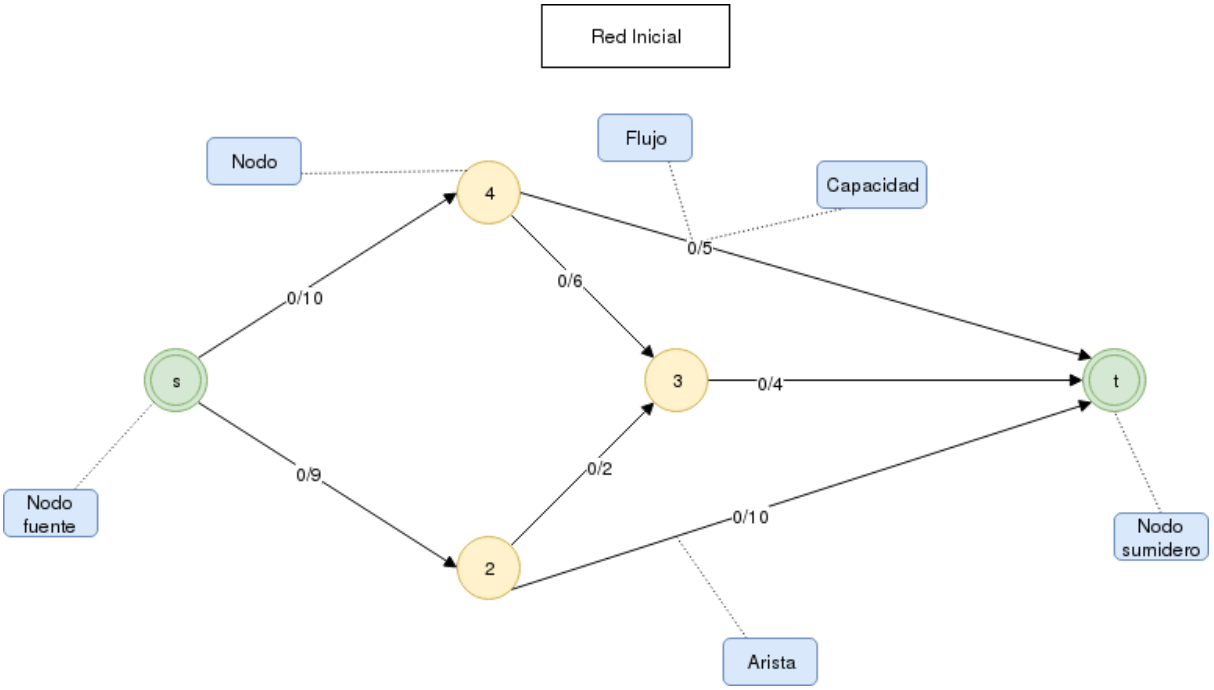
Construiremos distintos modelos matemáticos y distintas redes dependiendo del tipo de problema que formulemos.

4. Problema del flujo máximo

El problema del flujo máximo consiste en determinar la máxima cantidad de flujo que puede ser enviada a lo largo de una red dirigida, con una capacidad por unidad de tiempo asociada a cada una de ellas. Partiendo desde un nodo origen hasta un nodo destino. A continuación, vamos a ver las definiciones necesarias para entender cómo modelar el problema y aplicar el algoritmo de Ford-Fulkerson para resolverlo:

- Red con capacidad: Es un conjunto (N, E, c), donde $c : E \to \mathbb{R}$ es la función capacidad encargada de asociar, actualizar y mostrar la capacidad asociada a cada arista.

Nuestra red empieza en un nodo único, llamado nodo fuente (s), que no puede pertenecer al segundo término de una arista, para toda arista perteneciente a E. El nodo único que indica el final de la red se denomina nodo sumidero (t) y no puede pertenecer al primer término de una arista, para toda arista perteneciente a E.



Nota: En este apartado siempre que hagamos referencia a una red nos estaremos refiriendo a una red con capacidad.

- Flujo: Es una función $f : E \to \mathbb{R},$ que muestra el valor que representa la cantidad de capacidad ocupada asociada a una arista. Esta función siempre tiene que cumplir las siguientes condiciones:

    - El flujo resultante nunca debe ser mayor que la capacidad, es decir, $f(arista) \leq c(arista)$.

    - El flujo que entre en un nodo debe de ser igual al que salga del mismo nodo, es decir, $\sum_{arista\ que\ llega\ a\ n} f(arista) = \sum_{arista\ que\ sale\ de\ n} f(arista)$

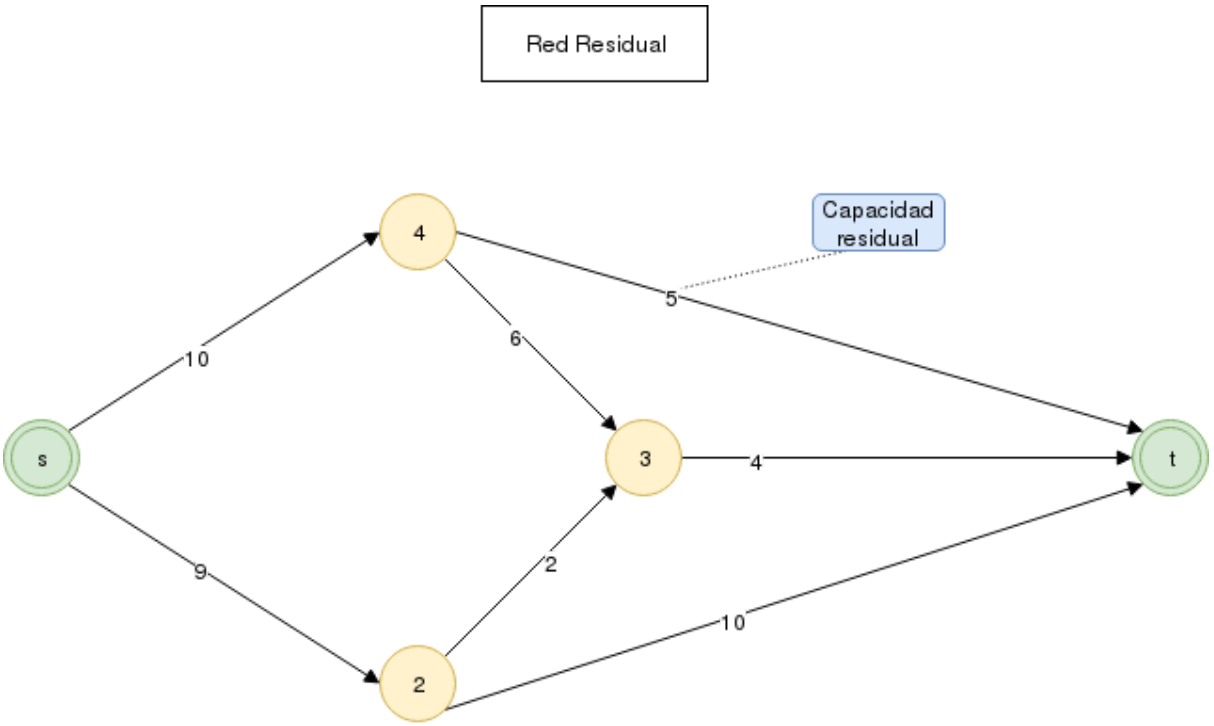
- Valor del flujo: Es la suma del flujo o de los flujos que salen del nodo fuente: $val(f) = \sum_{arista\ que\ sale\ de\ s} f(arista)$.

- Arista residual: Sea $arista \in E$ una arista dirigida, entonces su $arista_{residual}$ será un par ordenado cuyo primer término corresponderá al segundo término del par $arista$ y su segundo término al primero. Llamaremos E_f a la unión de todas las $aristas \in E$ y de sus $aristas_{residuales}$, es decir,

    $E_f = \{(a,b) / a,b \in N\ ;\ (b,a) \in E\}$.

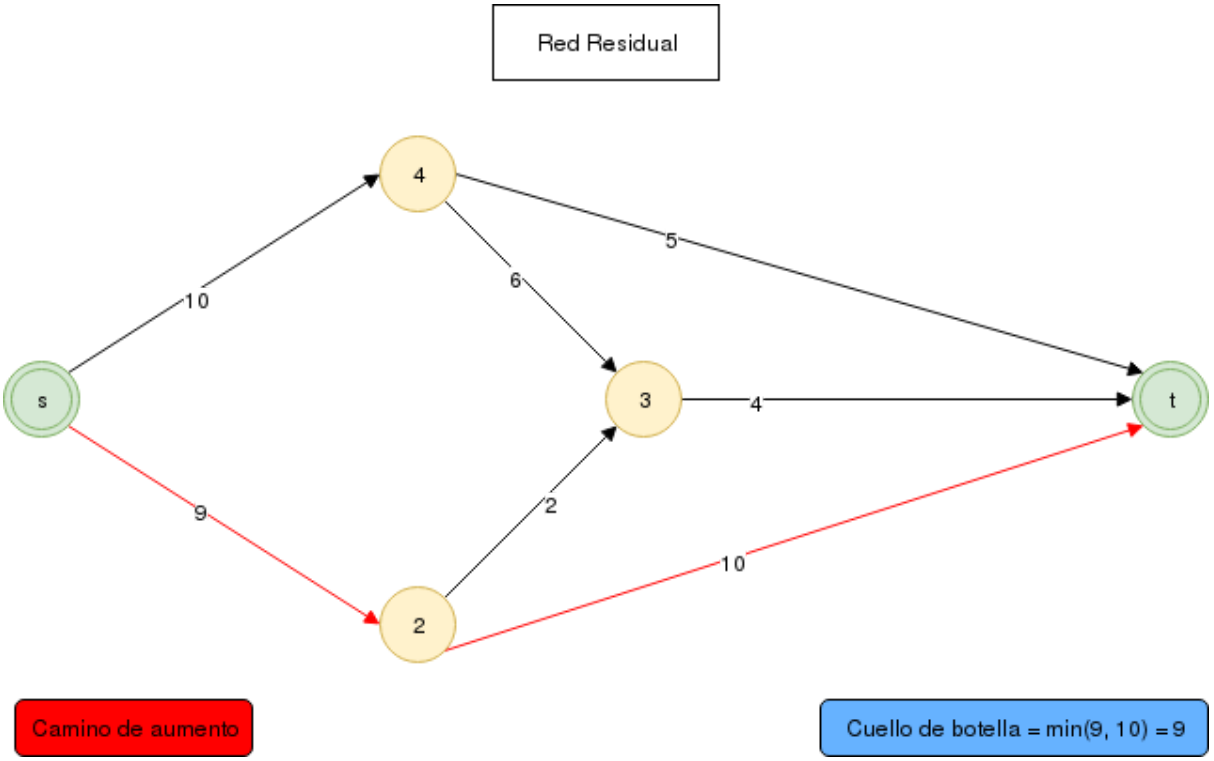
- Capacidad residual: Es el flujo restante para completar la capacidad de la arista: $c_f(arista) = c(arista) - f(arista)$.

- Red residual: Cuando tenemos una red $G = (N, E, c)$ y una función flujo f, tenemos definida $G_f = (N,\ E_f,\ c_f)$. El algoritmo se apoyará en esta red y la modificará con el objetivo de hallar el flujo máximo.



- Camino de aumento: Es un camino sin nodos repetidos que une s con t.

- Cuello de botella: El cuello de botella de un camino de aumento P es el mínimo de las capacidades residuales de las aristas de P.



4.1 Teorema del flujo máximo y corte mínimo

Una vez definidos los términos necesarios, vamos a explicar los conceptos de flujo máximo y corte mínimo así como su relación:

- Flujo máximo: Es el número máximo de unidades que pueden salir del nodo fuente y llegar al sumidero.

$$Flujo\ máximo\ =\ max\{val(f)\ /\ f\ es\ un\ flujo\ en\ G\}.$$

- Corte: Sea K un subconjunto de nodos, de tal forma que $N = K \cup N - K$, entonces diremos que $(K,\ N-K)$ es un corte en G. La capacidad del corte vendrá definida por $c(K)$ que suma las capacidades de las aristas que apuntan desde un nodo que pertenezca a K a otro nodo que pertenezca a $N-K$.

- Corte mínimo: Es un K corte que cumple la siguiente condición:

$$c(K) \leq c(H)\ para\ todo\ corte\ (H,\ N-H)\ en\ G.$$

4.2 Teorema de Ford-Fulkerson

En 1956, L. R. Ford Jr. y D. R. Fulkerson publicaron un teorema que muestra la relación entre el flujo máximo y el corte mínimo. Trabajando conjuntamente demostraron que en cualquier red el valor del flujo máximo es igual a la capacidad de un corte mínimo.

Para probar este resultado, vamos a demostrar previamente el lema del valor del flujo y un corolario que prueba que el valor del flujo siempre es menor que la capacidad de cualquier corte.

Lema del valor del flujo: Sea f un flujo y sea $(K, N-K)$ un corte de G. Entonces,

$$\sum_{arista\ sale\ de\ K} f(arista)\ -\ \sum_{arista\ entra\ en\ K} f(arista) = val(f).$$

Demostración:

$val(f) = \sum_{arista\ sale\ de\ s} f(arista) = \sum_{n \in K} (\sum_{arista\ sale\ de\ n} f(arista) - \sum_{arista\ entre\ en\ n} f(arista))$, por la propiedad de conservación del flujo, ya que excepto cuando n=s todos los términos son 0.

Por tanto,

$$val(f) = \sum_{arista\ sale\ de\ K} f(arista) - \sum_{arista\ entra\ en\ K} f(arista).$$

Corolario: Sea f un flujo y sea $(K, N-K)$ un corte de G. Entonces,

$$val(f) \leq c(K, N-K).$$

Demostración:

Por el lema anterior, $val(f) = \sum_{arista\ sale\ de\ K} f(arista) - \sum_{arista\ entra\ en\ K} f(arista)$, entonces

$$val(f) \leq \sum_{arista\ sale\ de\ K} f(arista) \leq \sum_{arista\ sale\ de\ K} c(arista) = c(K, N-K).$$

Demostración del teorema de Ford-Fulkerson:

Sea G una red. Supongamos que f es el flujo máximo y que $(K, N-K)$ es el corte mínimo, entonces, por el corolario anterior, $val(f) \leq c(K, N-K)$. Cómo el flujo es máximo, no existen más caminos de aumento en la red residual G_f, esto implica que el valor del flujo de las aristas que apuntan desde K a $N-K$ es igual a la capacidad de dichas aristas y que el flujo de sus aristas residuales es 0.

Por el lema anterior, $val(f) = \sum_{arista\ sale\ de\ K} f(arista) - \sum_{arista\ entra\ en\ K} f(arista)$, entonces,

$$val(f) = \sum_{arista\ sale\ de\ K} c(arista) = c(K, N-K).$$

4.3 Algoritmo de Ford-Fulkerson

Dado un conjunto de nodos N, un conjunto de aristas E, una función capacidad c, y los nodos fuente y sumidero, "s" y "t", podemos definir nuestra red, que denotaremos con la letra G.

El siguiente pseudocódigo muestra el funcionamiento del algoritmo de Ford-Fulkerson.

```
Ford-Fulkerson(G,s,t) {
    for (cada arco (u,v) de E) {
        f[u,v]= 0;
        f[v,u]= 0;
    }
    while (exista un camino p desde s a t en la red residual Gf) {
        cf(p) = min{cf(u,v): (u,v) está sobre p};
        for (cada arco (u,v) en p) {
            f[u,v]= f[u,v] + cf(p);
            f[v,u]= - f[u,v];
        }
    }
 }
```

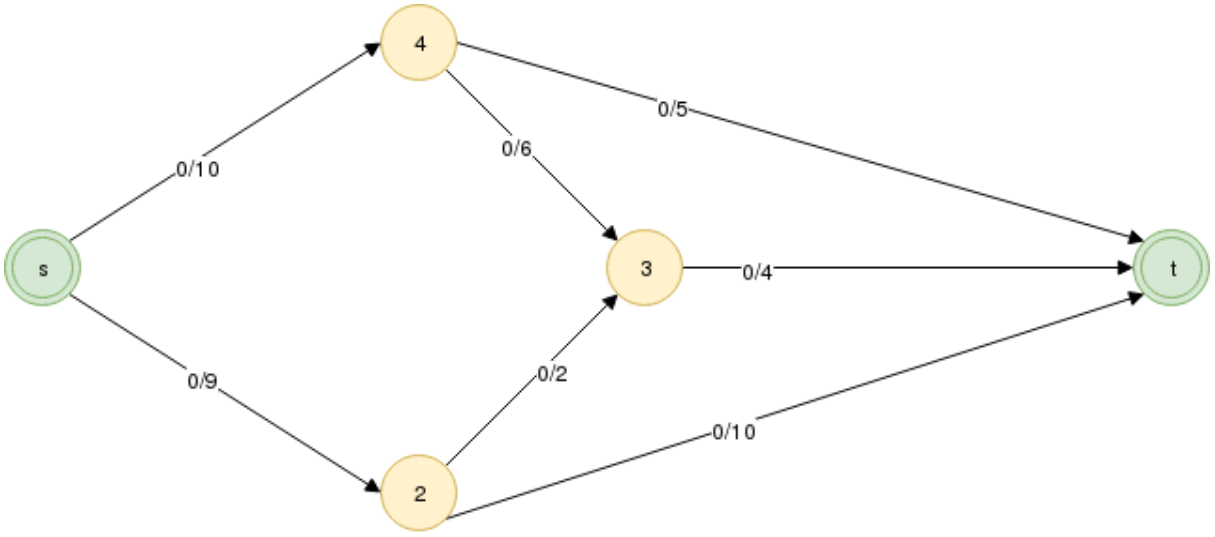
A continuación, vamos a mostrar los pasos que sigue el algoritmo para llevar a cabo su cometido:

1. Crear la red residual G_f.
2. Inicializar a 0 el flujo de todas las aristas de G_f y el valor del flujo máximo.
3. Buscar un camino de aumento p en G_f.
4. Calcular su cuello de botella.
5. Actualizar el valor del flujo máximo sumando el cuello de botella.
6. Actualizar el valor del flujo de las aristas de p y el de sus aristas residuales.
7. Repetir los pasos 3, 4 y 5 hasta que no se puedan encontrar más caminos de aumento.
8. Devolver el valor del flujo máximo.

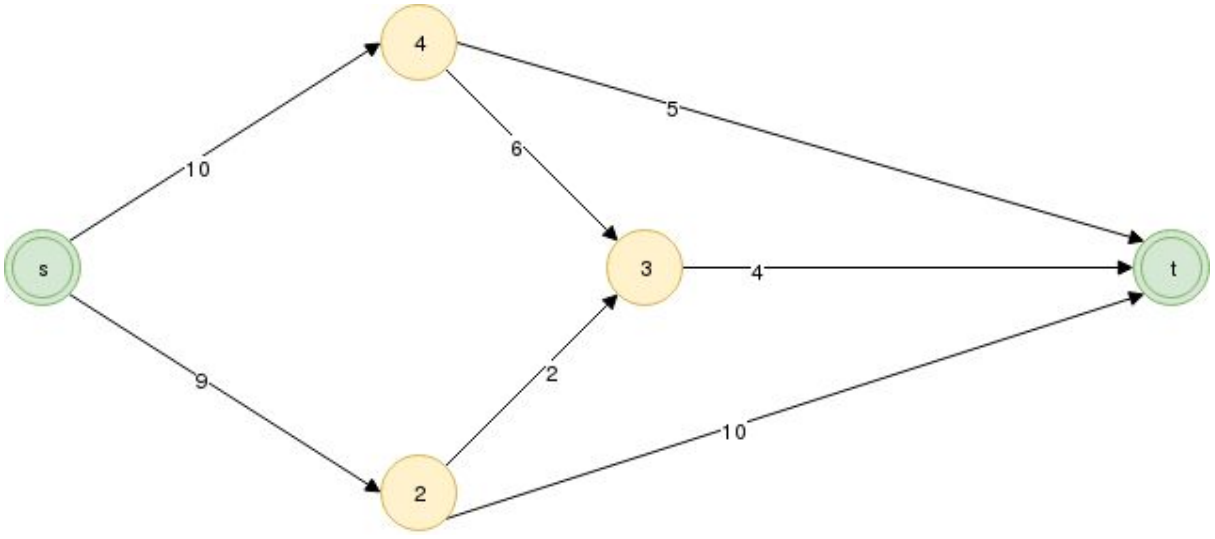
Por el teorema de Ford-Fulkerson, demostrado en el apartado anterior, sabemos que el valor del flujo máximo es igual a la capacidad del corte mínimo. Por tanto, una vez encontrado el flujo máximo podremos saber el corte mínimo.

4.4 Ford-Fulkerson aplicado a una red

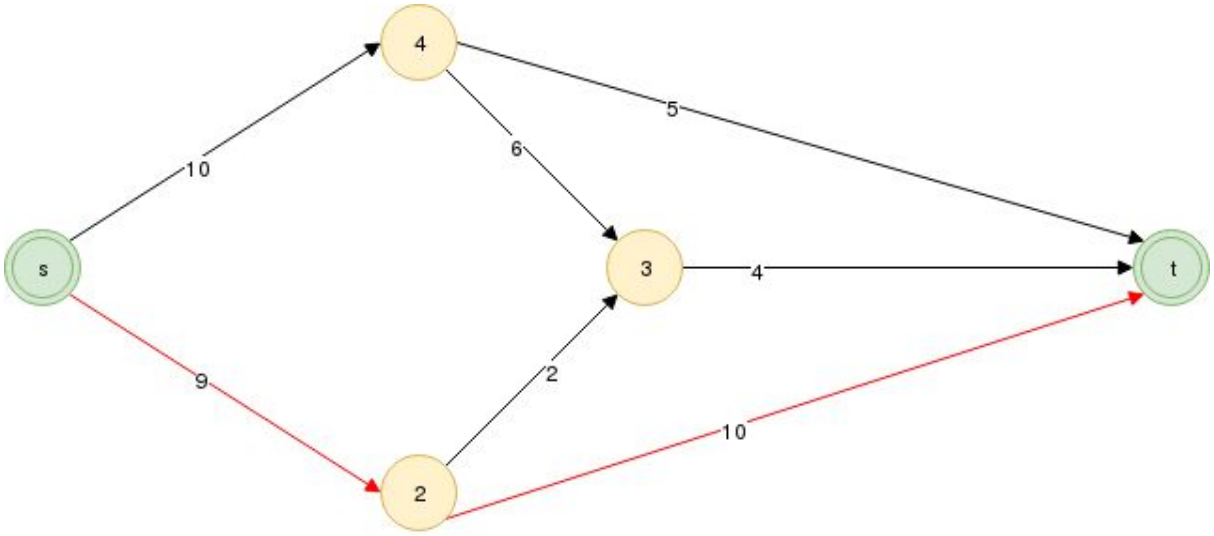
En este apartado, vamos a visualizar el funcionamiento del algoritmo en un sencillo ejemplo. Nuestra red está formada por un conjunto de 5 nodos y 7 aristas y una función capacidad. Empezamos poniendo todos los flujos a 0 y mostramos nuestra red inicial:



-A continuación, creamos nuestra red residual:



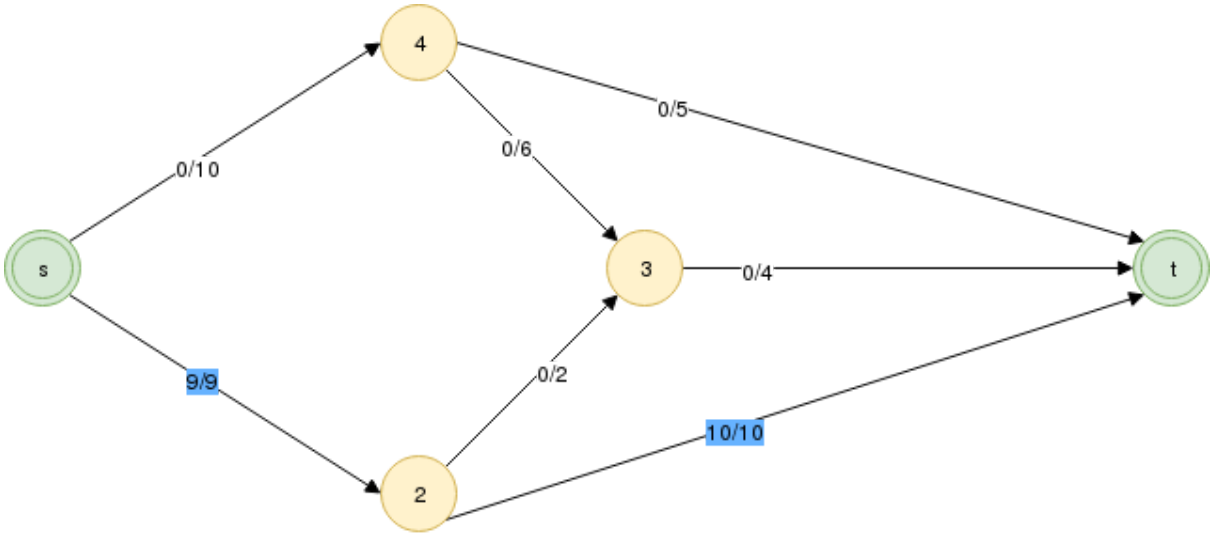
-Buscamos un camino de aumento:



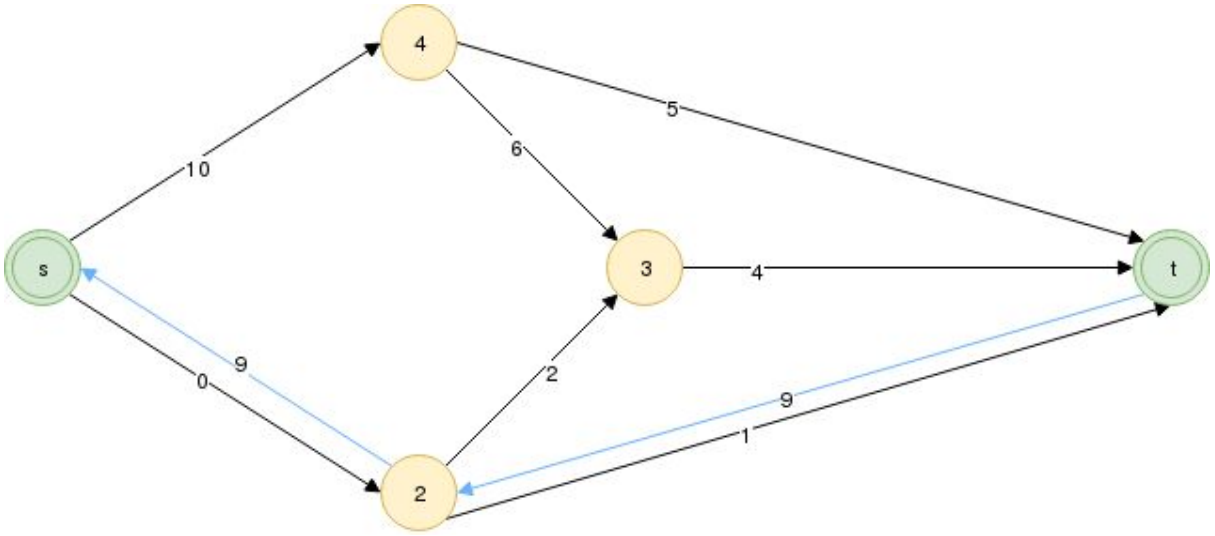
-Calculamos su cuello de botella: $b = min(9, 10) = 9$.

-El flujo máximo es 0 + 9 = 9.

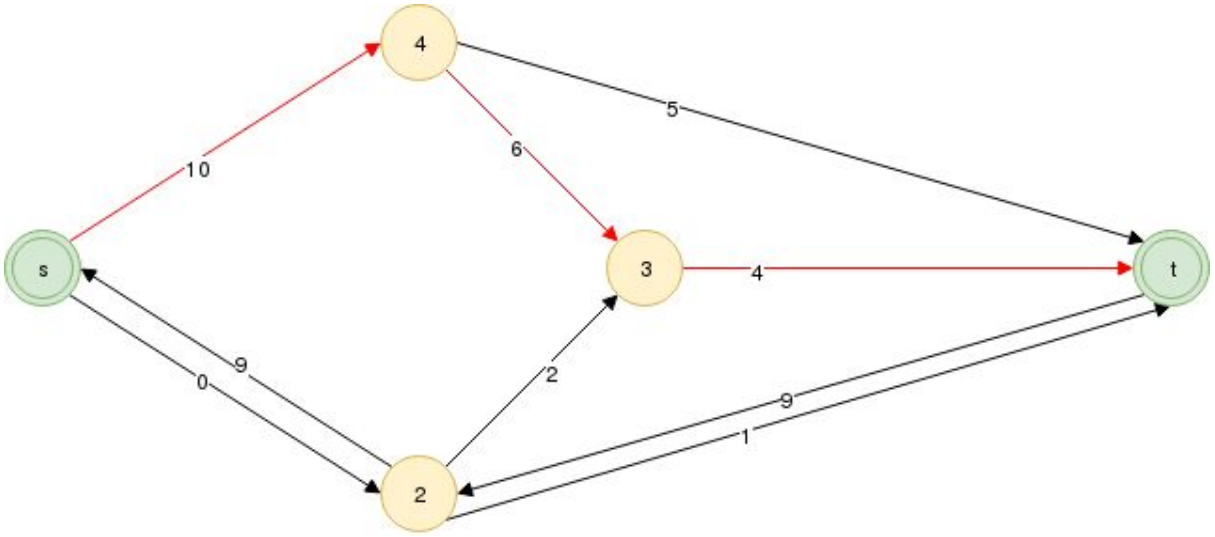
-Actualizamos el flujo de sus aristas en la red:



-Actualizamos la red residual:



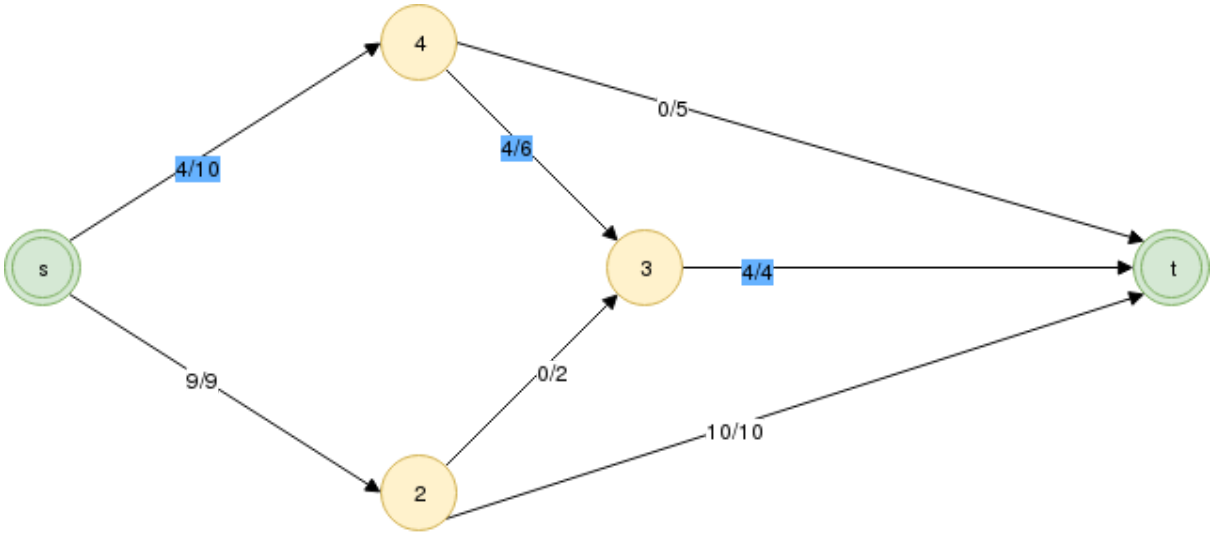
-Buscamos un camino de aumento:



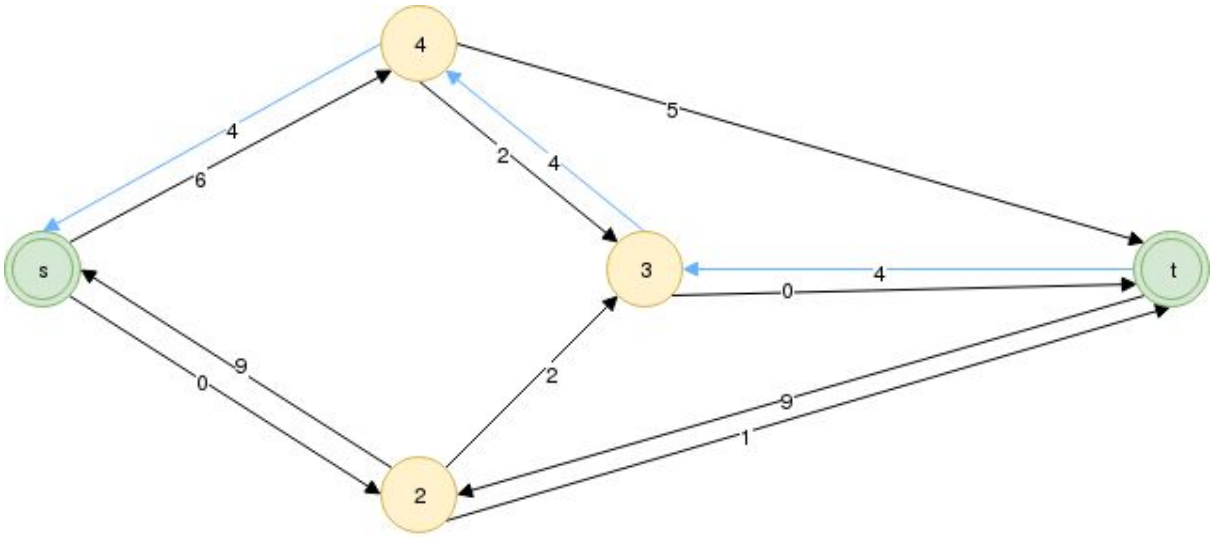
-Calculamos su cuello de botella: $b = min(10, 6, 4) = 4$.

-El flujo máximo es 9 + 4 = 13.

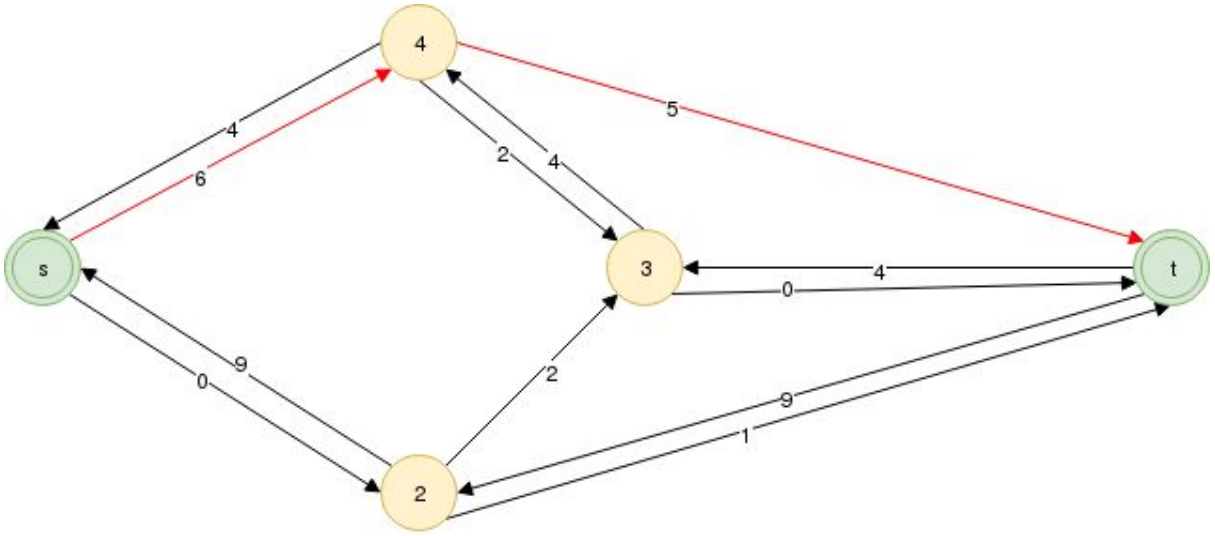
-Actualizamos el flujo de sus aristas en la red:



-Actualizamos la red residual:



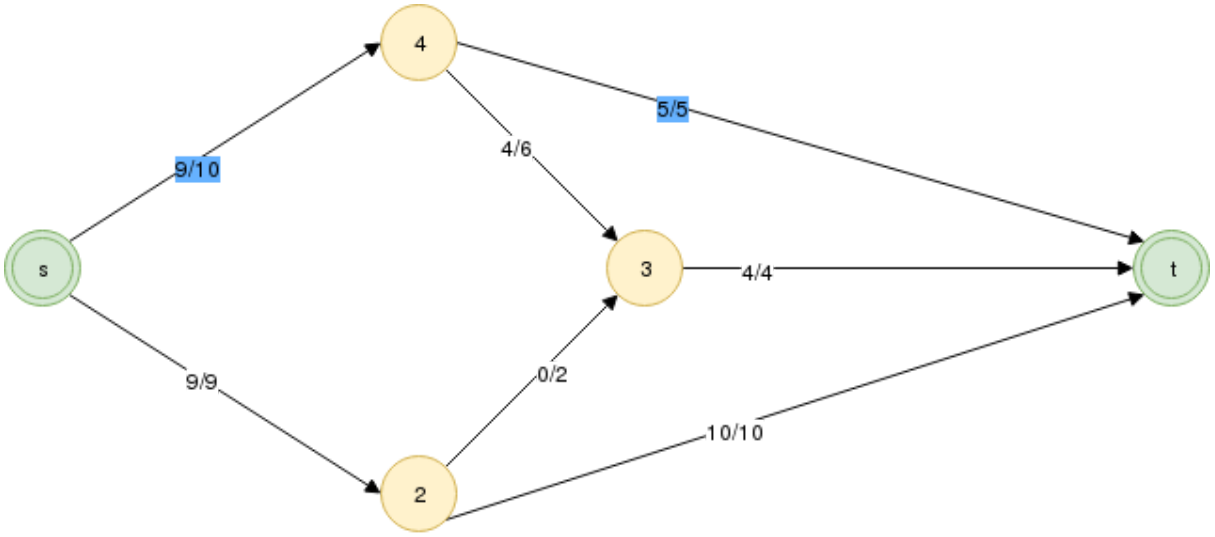
-Buscamos un camino de aumento:



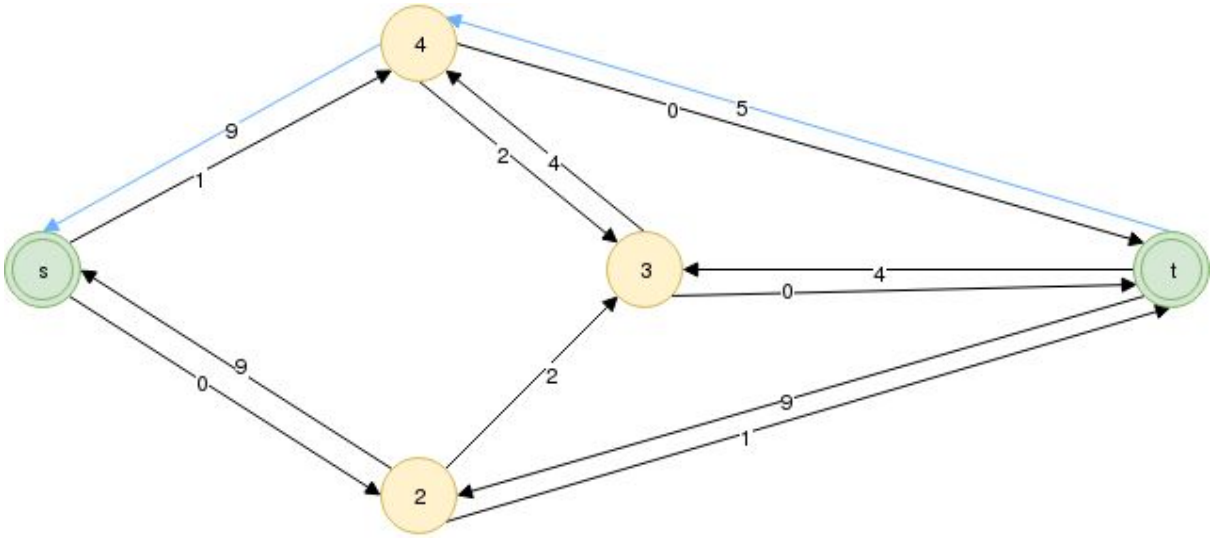
-Calculamos su cuello de botella: $b = min(6, 5) = 5$.

-El flujo máximo es 13 + 5 = 18.

-Actualizamos el flujo de sus aristas en la red:



-Actualizamos la red residual:



-Llegados a este punto ya no podemos encontrar más caminos de aumento, por tanto el algoritmo termina y nuestro flujo máximo es,

$$val(f) = \sum_{arista\ que\ sale\ de\ s} f(arista) = 9 + 9 = 18\ .$$

4.5 Complejidad computacional

El análisis de algoritmo depende de cómo se encuentren los caminos de aumento, si los elegimos mal se puede dar una situación de no convergencia y el algoritmo podría no terminar, es decir, el valor del flujo irá aumentando de forma sucesiva pero no estará obligado ni a converger al valor del flujo máximo. En el caso de elegir arbitrariamente los caminos y tener capacidades enteras, entonces la complejidad computacional es del orden $O(E * val(f))$, donde f es el flujo máximo encontrado. Este coste viene dado por la multiplicación del coste de inicializar todos los flujos en la red residual, que es del orden $O(E)$, por el $val(f)$, ya que en el peor de los casos podremos añadir 1 unidad de flujo en cada iteración.

J. Edmonds y R. Karp en 1972 propusieron una modificación del algoritmo que consiste en elegir el camino de aumento utilizando BFS(breadth-first search). Este algoritmo trata de realizar un recorrido en anchura sobre la red para escoger un camino basándose en el número de aristas que lo forman, si es mínimo lo elijo, si no busco el que sea mínimo.

La complejidad computacional de dicha modificación es del orden $O(N * E^2)$. Este coste viene dado por la multiplicación del coste de inicializar todos los flujos en la red residual , $O(E)$, y el coste del BFS, que al recorrer todos los nodos y todas las aristas en busca del camino más corto es del orden $O(N * E)$.

5. Problema del árbol de expansión de coste mínimo

Este problema plantea la situación de recorrer todos los nodos de una red no dirigida con la condición de que el coste sea mínimo. Recordemos que al emplear una red no dirigida para modelar el problema, se representarán aristas no dirigidas.

A continuación, vamos a definir los elementos necesarios para llevar a cabo el modelado del problema y entender el algoritmo de Prim:

- Red con coste: Es un conjunto (N, E, p), donde $p : E \to \mathbb{R}$ es la función encargada de asociar, actualizar y mostrar el coste de cada arista.

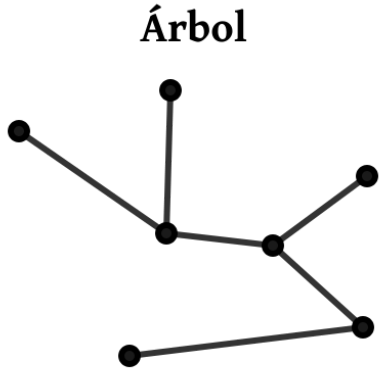
Nota: En este apartado siempre que hagamos referencia a una red nos estaremos refiriendo a una red con coste.

- Red simple: Es una red que verifica que para todo par de nodos, existe a lo sumo una única arista que los une.

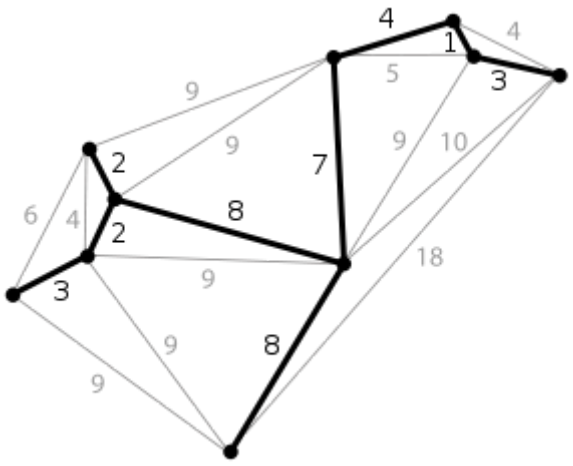
$$RS = [\forall n_1, n_2 \in N \Rightarrow \exists ! \{n_1, n_2\} \in E]$$

- Red conexa: Es una red donde entre cualquier par de nodos existe un camino para ir entre ellos.

- Red sin ciclos: Es una red que no contiene ningún camino que empiece y termine en el mismo nodo.



- Árbol: Es un par (N_a, E_a) de una red, tal que $N_a \subseteq N \, y \, E_a \subseteq E$. Cumple las propiedades de red simple, conexa y sin ciclos.



- Árbol de expansión de coste mínimo: Es el árbol que cumple las siguientes condiciones:

    - Debe contener a todos los nodos de la red, $N_a = N$.
    - La suma del coste de cada una de las aristas que lo forman es mínima.

5.1 Algoritmo de Prim

Dado un conjunto de nodos N, un conjunto de aristas E y una función coste p, podemos definir nuestra red G. Para poder aplicar el algoritmo de Prim y resolver el problema del árbol de expansión mínimo, es necesario que todos los costes asociados a las aristas sean no negativos.

Es un algoritmo que incrementa continuamente el tamaño de un árbol, comenzando por un nodo inicial, elegido al azar, al que se le van agregando sucesivamente nodos cuya distancia a los anteriores es mínima. En cada paso, consideraremos las aristas que contienen nodos que ya pertenecen al árbol. El árbol de expansión de coste mínimo está completamente construido cuando no quedan más nodos por agregar.

El siguiente pseudocódigo muestra el funcionamiento del algoritmo de Prim, utilizando la **cola de prioridad** como estructura de datos auxiliar. Una cola es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción se realiza por un extremo y la operación de extracción por el otro. La cola de prioridad es una estructura de tipo cola que permite ordenar sus elementos según la propiedad asignada como prioridad. En este caso, asignaremos la propiedad “distancia” como prioridad y consideraremos los nodos con mayor prioridad aquellos cuya distancia sea menor.

```
Prim (Red G(N,E,p))
     por cada u en N hacer
         distancia[u] = INFINITO
         nodo_padre[u] = NULL
         Insertar(cola,<u, distancia[u]>)
     distancia[u]=0
     mientras !esta_vacia(cola) hacer
         u = extraer_minimo(cola)
         por cada v adyacente a 'u' hacer
             si ((v ∈ cola) && (distancia[v] > p(u, v)) entonces
                 nodo_padre[v] = u
                 distancia[v] = p(u, v)
                 Actualizar(cola,<v, distancia[v]>)
```

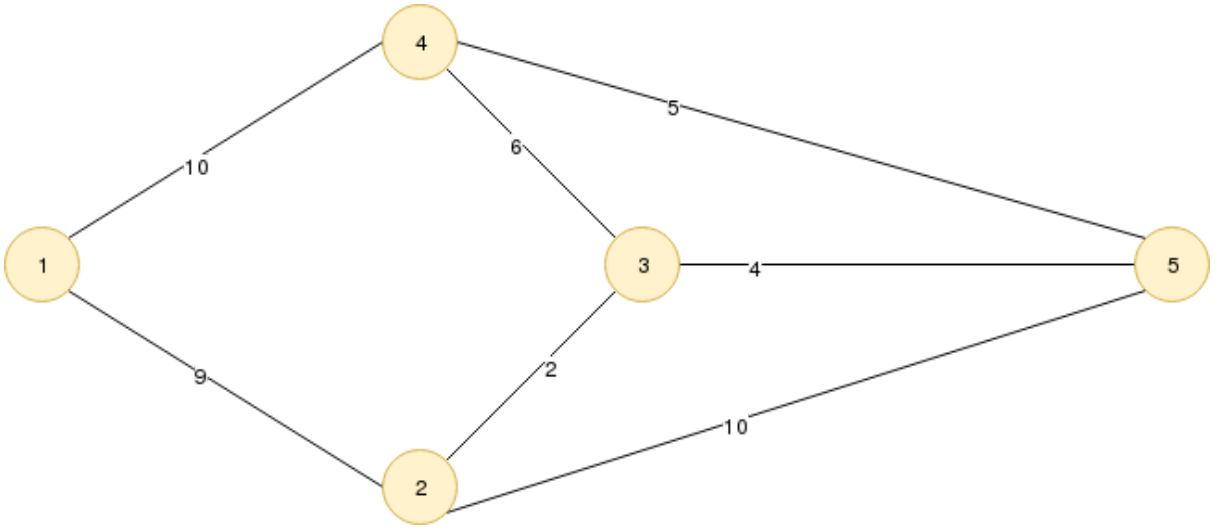
Nota: El algoritmo de Prim puede ser implementado de diversas formas. Por ello los pasos descritos a continuación se basan en la implementación que utiliza una cola de prioridad y no deben considerarse como genéricos.

Pasos que sigue el pseudocódigo:

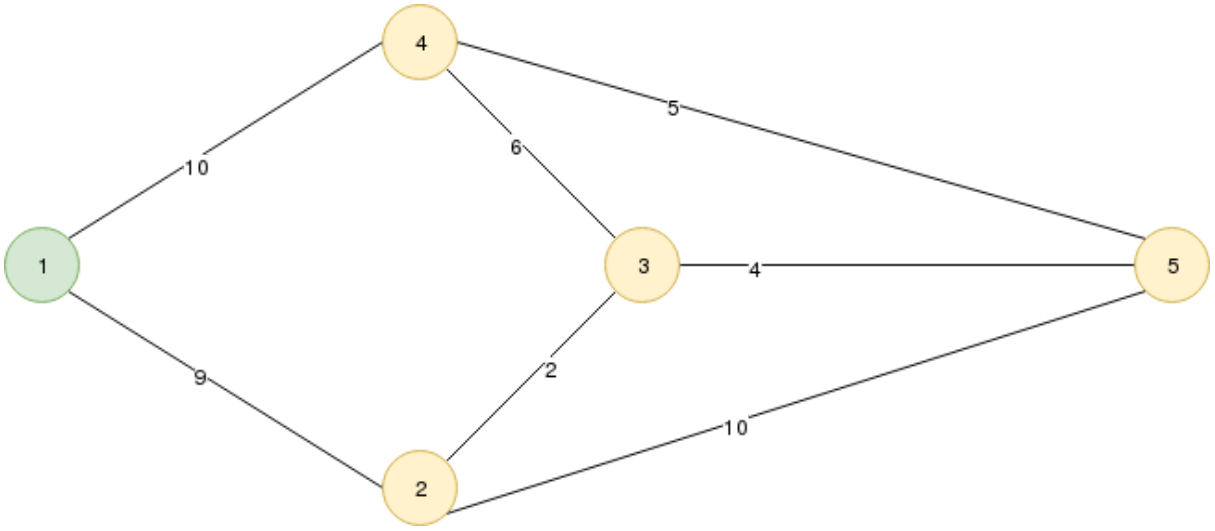
1. Añadir a la cola todos los nodos junto con una distancia mayor al máximo coste de una arista. Actualizar la distancia al valor 0 de un nodo elegido al azar.
2. Extraer el nodo cuya distancia sea la menor de todas las presentes.
3. Para cada nodo adyacente, si el nodo está en la cola y el coste devuelto por la función p es menor que la distancia:
    a. Enlazar el nodo inicial como padre del nodo adyacente.
    b. Actualizar la distancia al coste devuelto por p.
    c. Añadir a la cola el nodo con la distancia actualizada.
4. Si quedan nodos en la cola, repetir los pasos 2 y 3. En caso contrario devolver el árbol de expansión de coste mínimo.

5.2 Prim aplicado a una red

En este apartado, vamos a visualizar el funcionamiento del algoritmo en un sencillo ejemplo. Nuestra red está formada por un conjunto de 5 nodos y 7 aristas y una función coste.



- Para empezar, seleccionamos un nodo al azar.

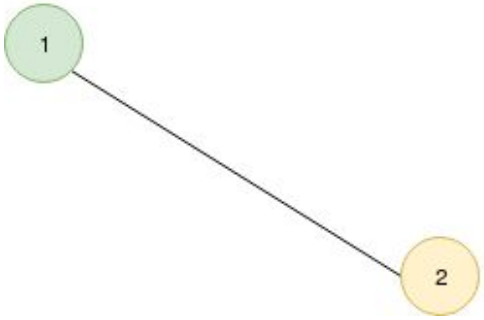


- A continuación, comparamos el coste de llegar a cada uno de sus nodos adyacentes, escogemos la arista con coste mínimo, enlazamos los nodos y las restantes las añadimos a la cola de prioridad.

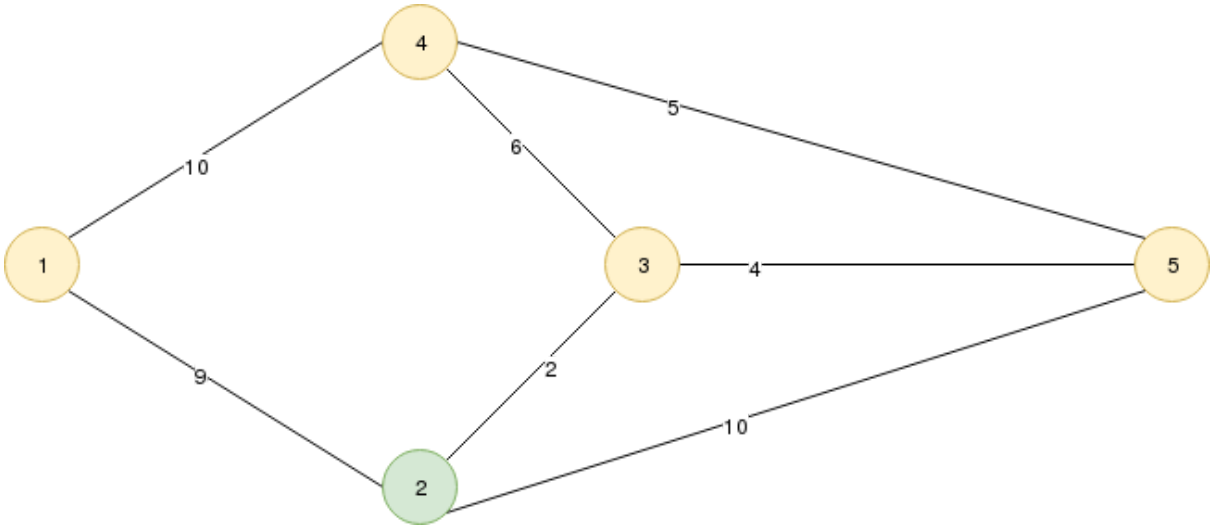
- ¿$9 < 10$? Sí. Enlazamos el nodo número 2 y añadimos el 4 a la cola.

Árbol

CP = [(4, 10)]



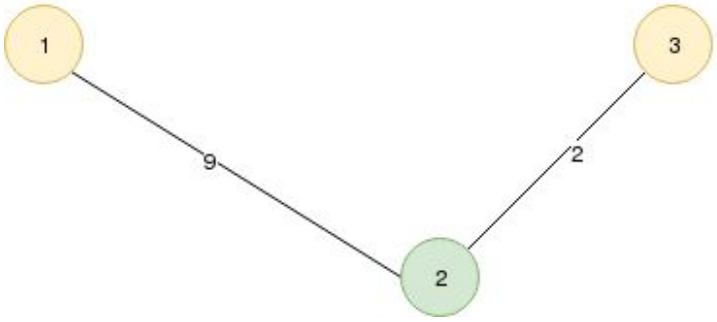
- Ahora nos posicionamos en el nodo que hemos enlazado. Comparamos el coste de llegar a sus nodos adyacentes, escogemos la arista con menor coste y la comparamos con la primera arista de la cola de prioridad. Si la arista de la cola tiene un coste menor, entonces la extraemos, enlazamos los nodos y insertamos las aristas restantes a la cola de prioridad.



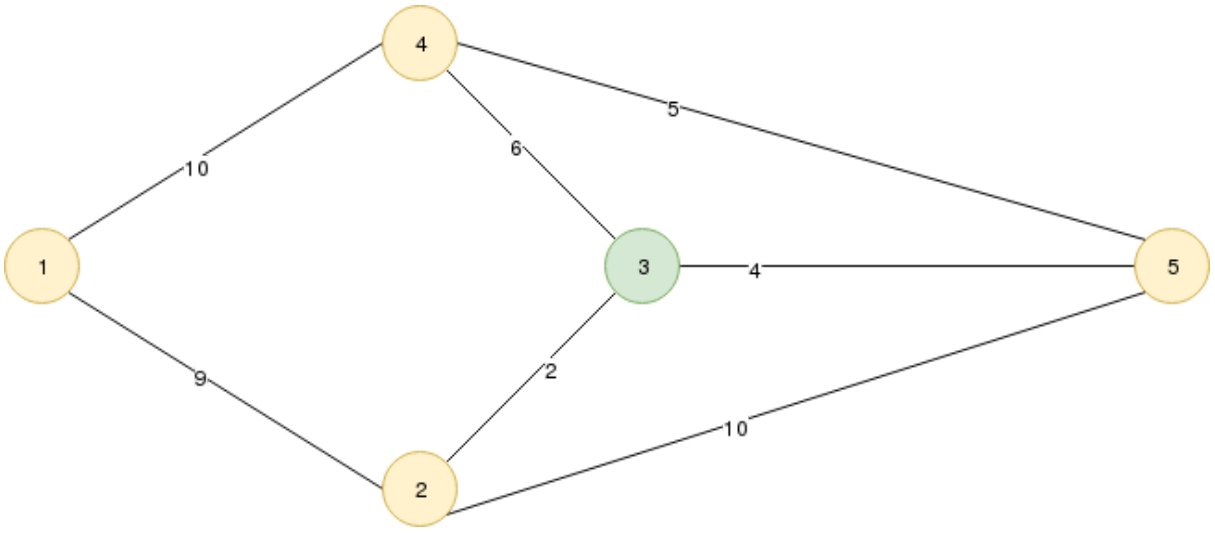
- ¿$2 < 10$? Sí. Enlazamos el nodo número 3 y añadimos el 5 a la cola.

Árbol

CP = [(4, 10), (5, 10)]



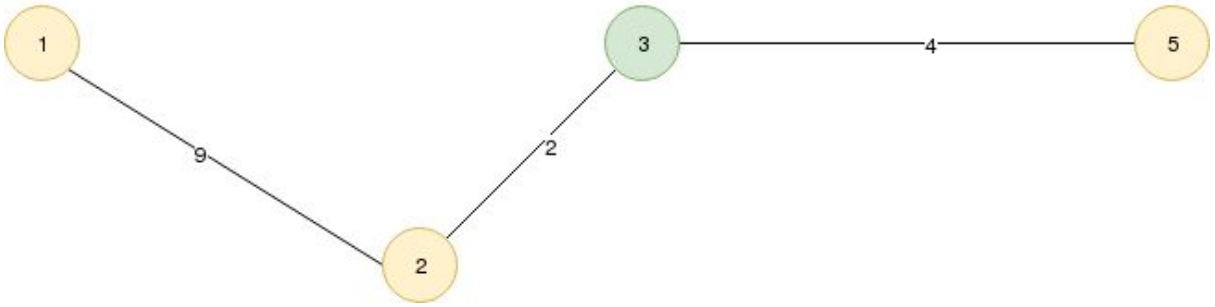
- Repetimos el último paso hasta que no queden más nodos que enlazar:



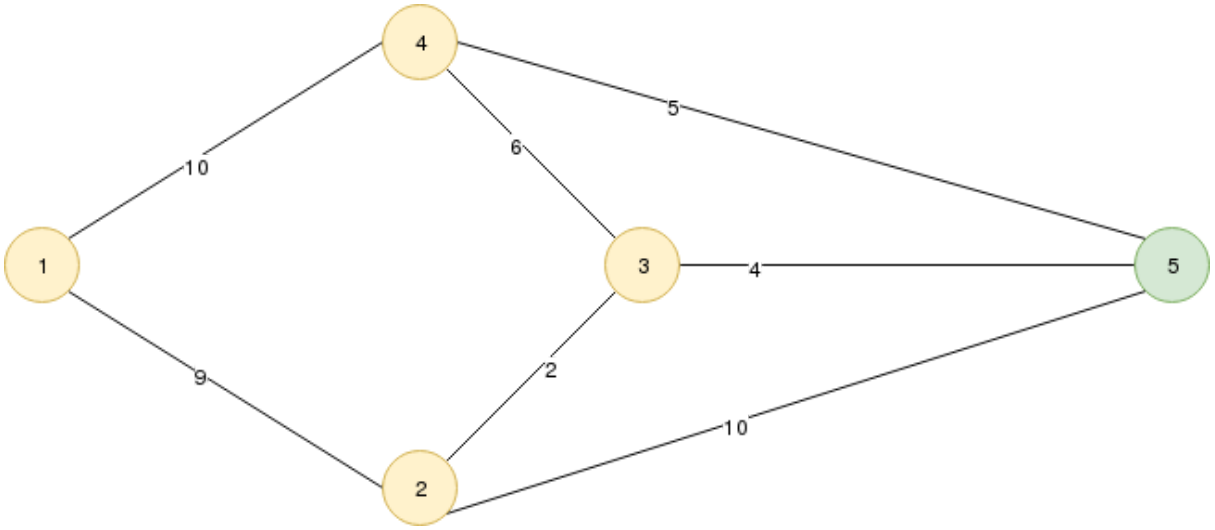
- ¿$4 < 6$? Sí. ¿$4 < 10$? Sí. Enlazamos el nodo número 5 y añadimos el 4 a la cola.

Árbol

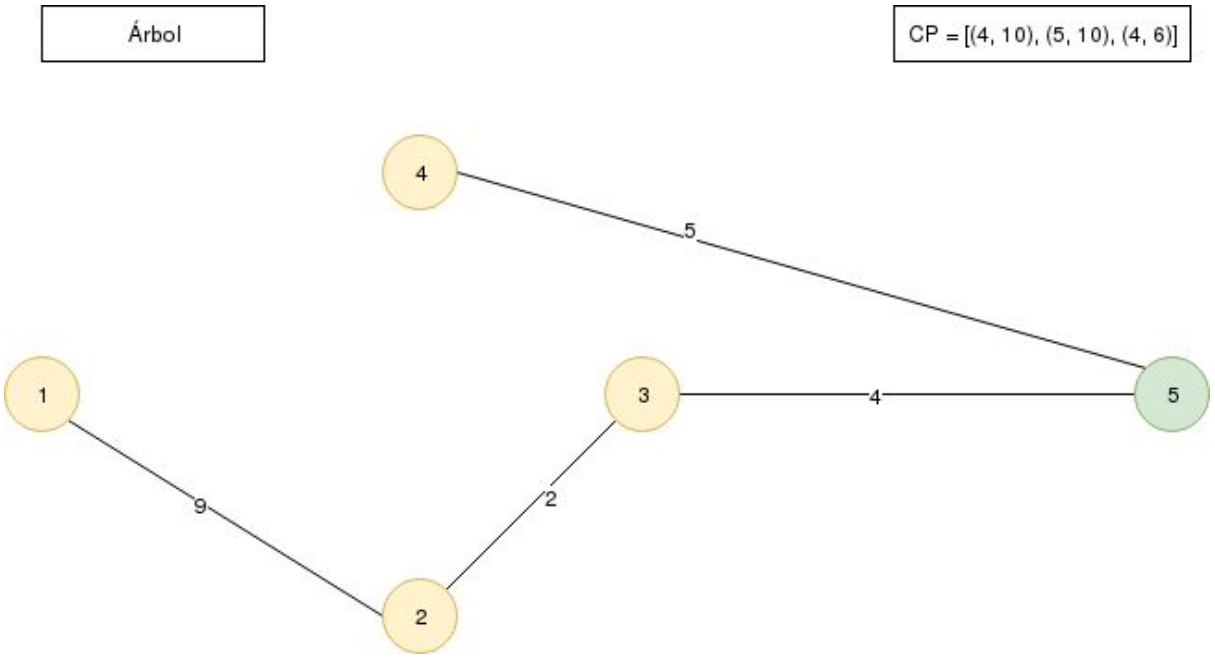
CP = [(4, 10), (5, 10), (4, 6)]



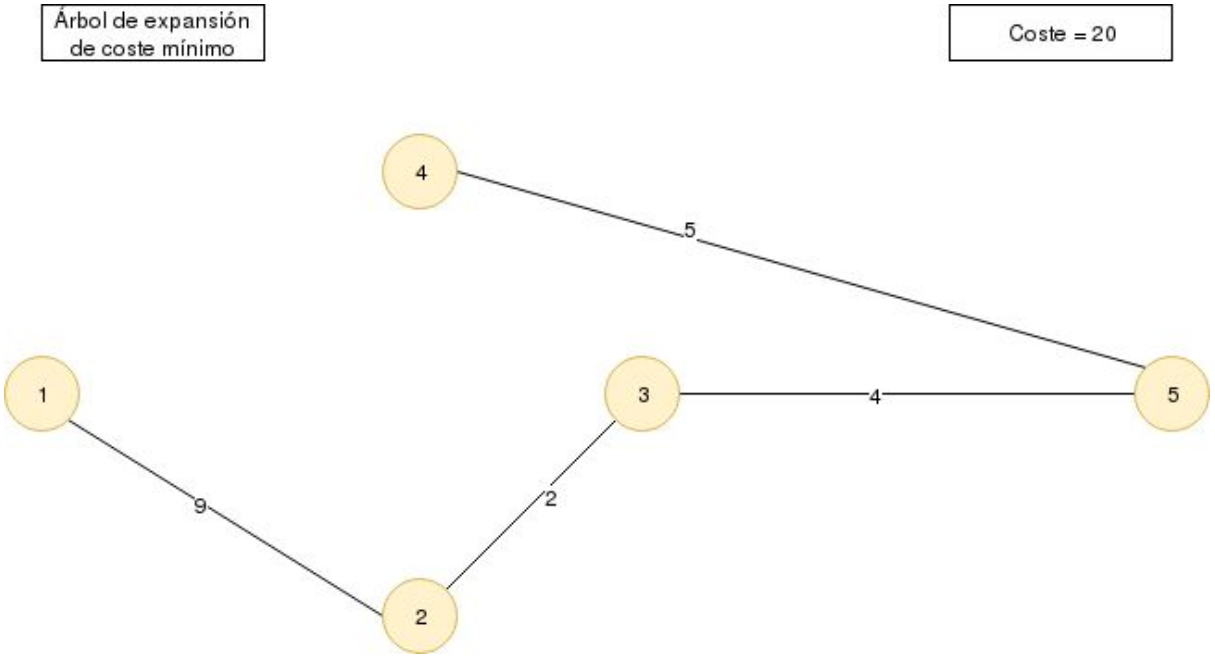
- Nos posicionamos en el nodo 5 y seguimos iterando.



- ¿ $5 < 6$? Sí. Enlazamos el nodo número 4 y terminamos.



- El resultado es nuestro árbol de expansión de coste mínimo.



5.3 Complejidad computacional

Nota: La complejidad computacional del algoritmo de Prim está condicionada a la forma de programación del algoritmo, en nuestro caso, al utilizar una cola de prioridad, la complejidad es la descrita en los siguientes párrafos.

En primer lugar, tenemos el bucle que encola todos los nodos junto a la distancia. Dicho bucle tiene una complejidad del orden $O(N * log(N))$ debido a que debemos añadir a la cola, $O(log(N)),$ todos los nodos de la red, $O(N)$.

En segundo lugar, tenemos la instrucción que elimina de la cola el nodo con menor distancia. En este caso, la complejidad volverá a ser del orden $O(N * log(N))$ ya que el algoritmo termina cuando se han eliminado todos los elementos de la cola.

Para finalizar, tenemos el bucle que analiza todos los nodos adyacentes al nodo eliminado de la cola. En este caso, la complejidad cambiará, ya que la función que devuelve el coste de una arista tiene que recorrerlas todas. Por tanto, será del orden $O(E * log(N))$.

Como nuestra red es conexa, esto implicará $|E| \geq |N| - 1$. Por tanto, sumando las tres complejidades, tenemos que la complejidad computacional del algoritmo es del orden $O(E * log(N))$.

6. Problema del camino más corto

El problema surge de la necesidad de recorrer todos los nodos de una red, con aristas dirigidas, partiendo desde un nodo fuente, con la condición de que la longitud del recorrido sea mínima. Para modelar este tipo de problema cada arista de la red debe tener una longitud asociada.

A continuación, vamos a definir los elementos necesarios para llevar a cabo el modelado del problema y entender el algoritmo de Dijkstra:

- Red con longitud: Es un conjunto (N, E, l), donde $l : E \to \mathbb{R}$ es la función encargada de asociar, actualizar y mostrar una longitud a cada una de las aristas.

Nota: En este apartado siempre que hagamos referencia a una red nos estaremos refiriendo a una red, dirigida, con longitud.

- Camino mínimo: Es el camino de menor longitud que pasa por todos los nodos de una red, iniciando su recorrido desde un nodo fuente (s).

6.1 Algoritmo de Dijkstra

Dado un conjunto de nodos N, un conjunto de aristas E y una función longitud l, podemos definir nuestra red G. Para poder aplicar el algoritmo de Dijkstra y resolver el problema del camino más corto, es necesario que todas las longitudes asociadas a las aristas sean no negativas.

Es un algoritmo que parte de un nodo fuente (s) y va explorando los caminos más cortos hacia el resto de los nodos.

El siguiente pseudocódigo muestra su funcionamiento, utilizando la **cola de prioridad** como estructura de datos auxiliar.

```
DIJKSTRA (Red G, nodo_fuente s)
    para u ∈ V[G] hacer
        distancia[u] = INFINITO
        nodo_padre[u] = NULL
        visto[u] = false
    distancia[s] = 0
    insertar (cola, (s, distancia[s]))
```

```
mientras que cola no esté vacía hacer
   u = extraer_mínimo(cola)
   visto[u] = true
   para todos v ∈ adyacencia[u] hacer
      si no visto[v] y distancia[v] > distancia[u] + longitud(u, v) hacer
         distancia[v] = distancia[u] + longitud(u, v)
         nodo_padre[v] = u
         insertar(cola,(v, distancia[v]))
```

Nota: El algoritmo de Dijkstra puede ser implementado de diversas formas, por ello los pasos descritos a continuación no deben considerarse como genéricos.

A continuación, vamos a describir los pasos que sigue el pseudocódigo:

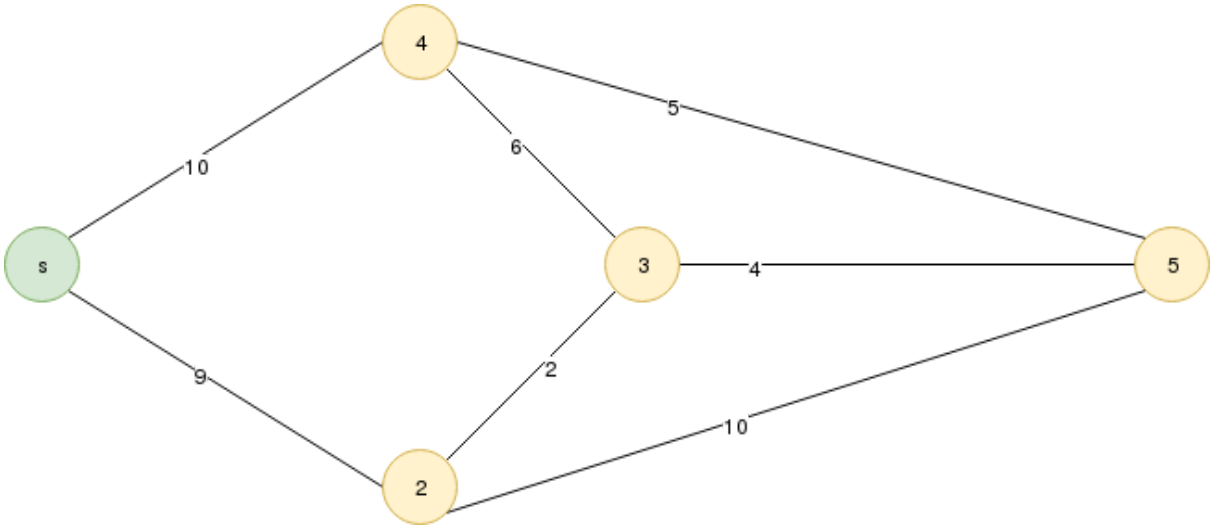
1. Insertar todos los nodos de la red a una cola de prioridad junto con una longitud igual a infinito. A la vez que los insertamos inicializamos el array "padre", que es el que mostrará el camino más corto, a **null** y el array "visto", que se encargará de marcar aquellos nodos que ya han sido visitados, a **false**.
2. Insertar a la cola el nodo fuente junto a una longitud 0.
3. Extraer el nodo cuya longitud sea mínima y marcarlo como visto.
4. Para cada nodo adyacente al nodo extraído, si:
   4.1. No ha sido marcado como visto.
   4.2. La suma de la distancia asociada al nodo extraído y la distancia del nodo extraído hacia el nodo adyacente es menor que la distancia del nodo adyacente.

   Entonces, actualizar la distancia del nodo adyacente, asociar el nodo extraído como padre del nodo adyacente y insertar en la cola el nodo adyacente con su distancia actualizada.
5. Repetir los pasos 3 y 4 hasta que la cola esté vacía.
6. Al finalizar el algoritmo el array "padre" nos mostrará el camino más corto.

6.2 Dijkstra aplicado a una red

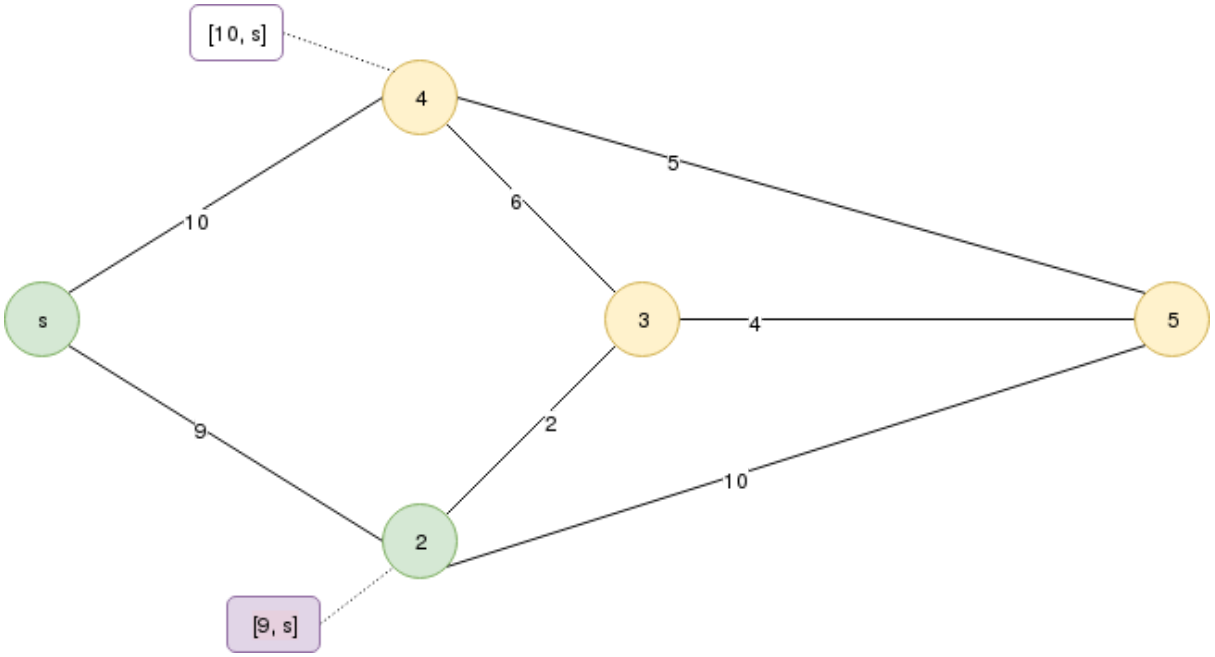
En este apartado, vamos a visualizar el funcionamiento del algoritmo en un sencillo ejemplo. Nuestra red está formada por un conjunto de 5 nodos y 7 aristas y una función longitud.

- Empezamos desde el nodo fuente (s).

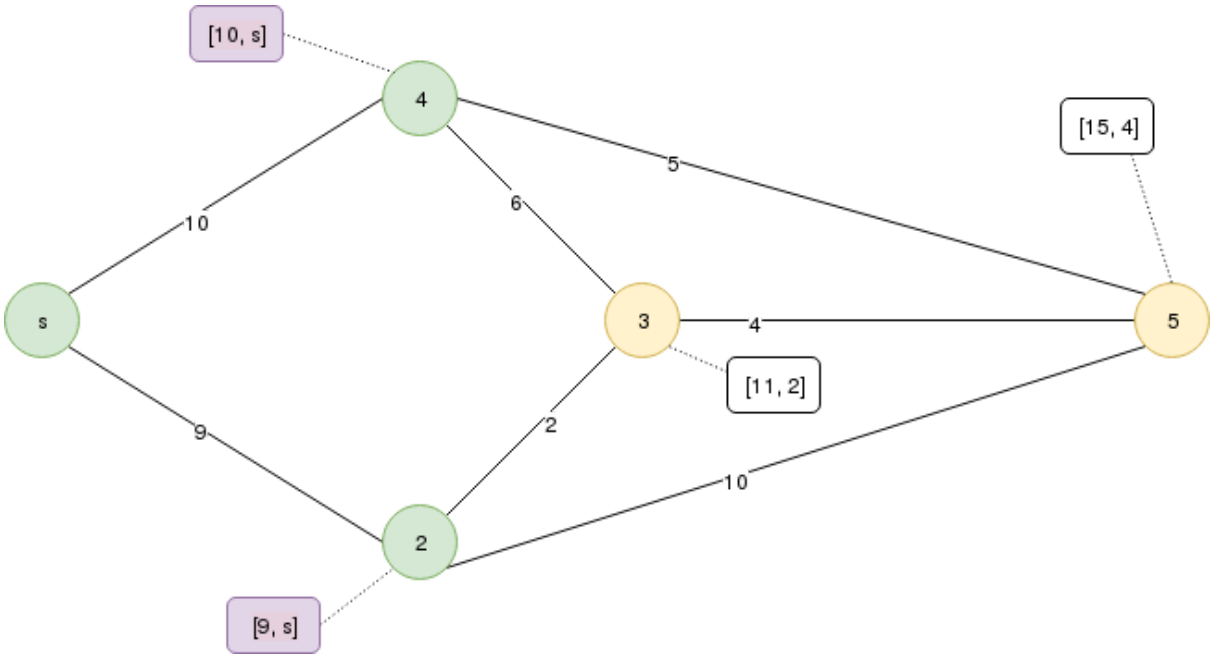


Nota: Vamos a emplear la notación **[longitud, nodo]** para indicar en cada momento cual es la longitud al llegar al nodo y de cual hay que salir para obtenerla.

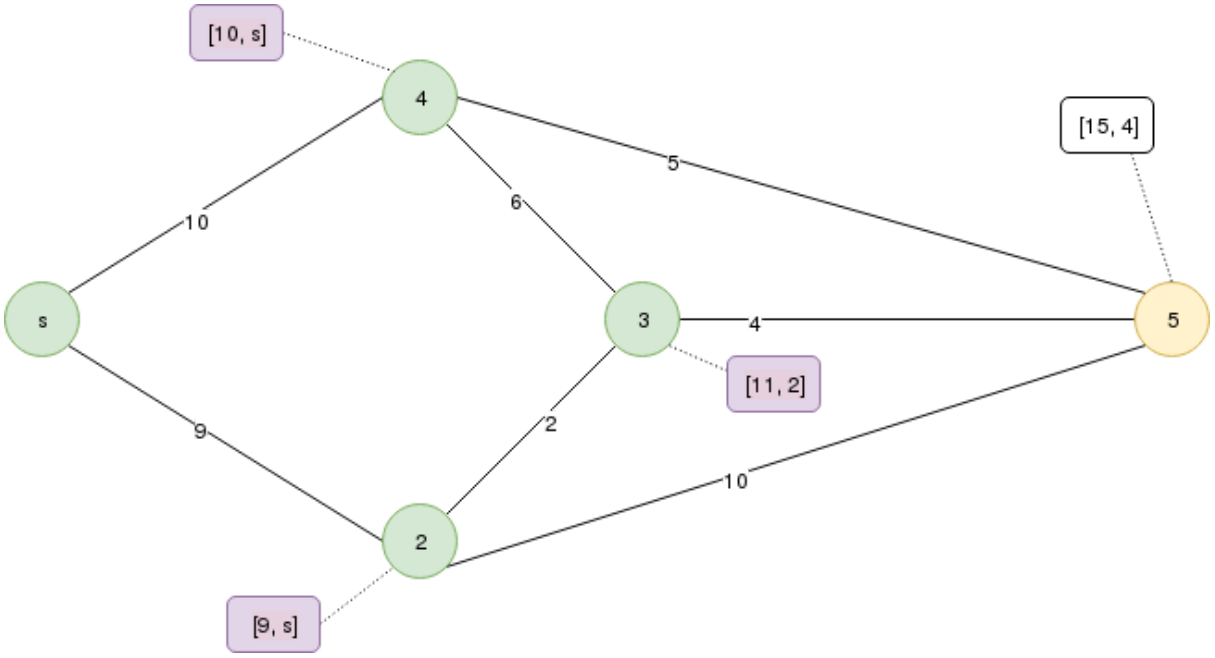
- Calculamos las longitudes de los nodos adyacentes, seleccionamos la de menor peso y insertamos las restantes a la cola. $CP = [(10,\ 4)]$.



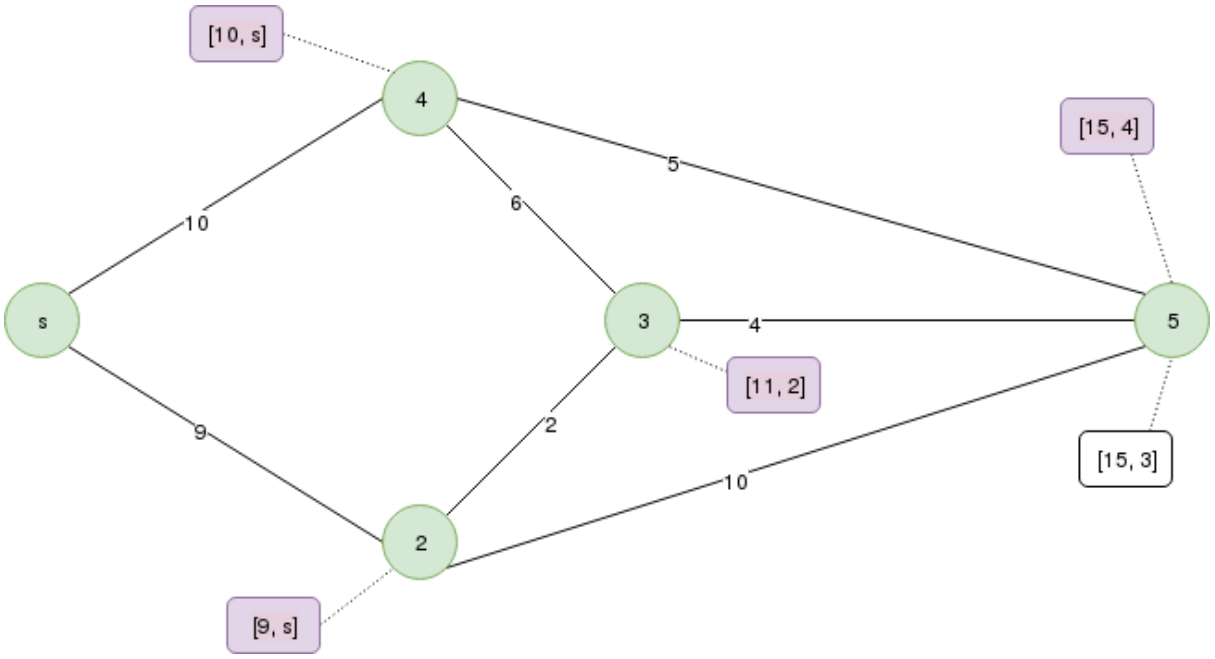
- Calculamos las longitudes de los nodos adyacentes al nodo 2, las comparamos junto a la del nodo 4 y elegimos la de menor peso. $CP = [(11, 3)]$.



- Calculamos las longitudes de los nodos adyacentes al nodo 4 que no estén visitados, las comparamos junto a la del nodo 2 y elegimos la de menor peso. $CP = [(15, 5)]$.



- Calculamos las longitudes de los nodos adyacentes al nodo 3 que no estén visitados, las comparamos junto a la del nodo 5 y elegimos la de menor peso.



- En nuestro caso tenemos dos rutas similares en longitud (15). Como la ruta 1-4-5 tiene menos nodos que la ruta 1-2-3-5, escogeremos la primera como solución al problema.

6.3 Complejidad computacional

Nota: La complejidad computacional del algoritmo de Dijkstra está condicionada a la forma de programación del algoritmo, en nuestro caso al utilizar una cola de prioridad la complejidad es la descrita en los párrafos siguientes.

En primer lugar, tenemos el bucle que inserta en la cola todos los nodos junto a la longitud. Dicho bucle tiene una complejidad del orden $O(N)$ ya que estamos utilizando una cola de prioridad ordenada según la longitud del nodo fuente hacia cada nodo.

En segundo lugar, tenemos la instrucción que elimina de la cola el nodo con menor longitud. El algoritmo finaliza cuando no queden nodos en la cola, por tanto extraemos un elemento de la cola $O(log(N))$ tantas veces como nodos existan $O(N)$. La complejidad de esta parte es del orden $O(N * log(N))$.

A continuación, tenemos el bucle que analiza todos los nodos adyacentes al nodo eliminado de la cola. En este caso, la complejidad vendrá dada por la función que devuelve la longitud de una arista y por el coste de actualizar dicha longitud en la cola. Por tanto, será del orden $O(E * log(N))$.

Sumando las tres complejidades obtenemos una complejidad del orden $O((N+E) * log(N))$. En el caso de que la red sea conexa, es decir, $|E| \geq |N| - 1$, la complejidad computacional del algoritmo será del orden $O(E * log(N))$.

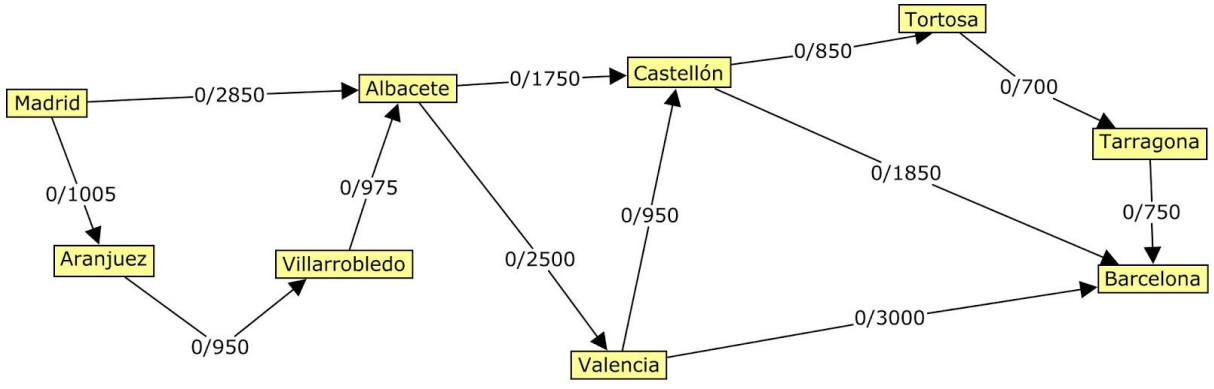
7. Aplicaciones de los algoritmos

7.1 Solución del problema de distribución de la gasolina aplicando FF

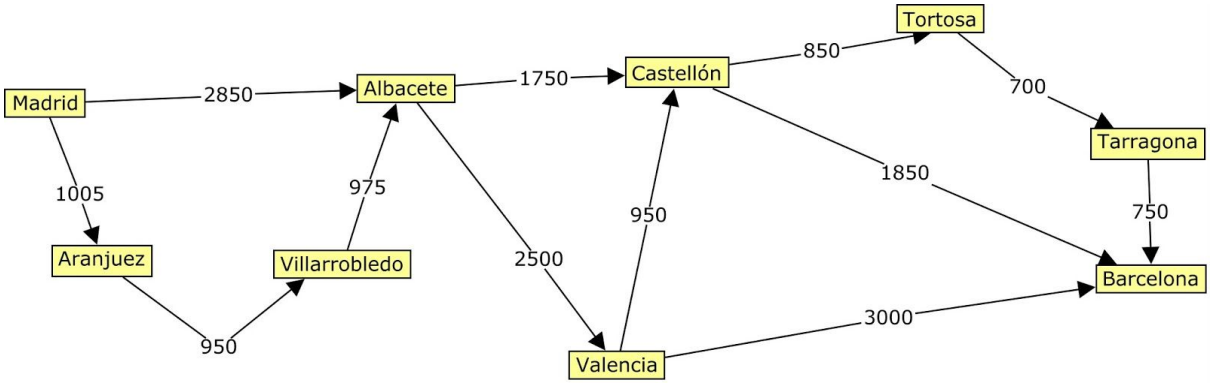
Una compañía petrolera tiene una refinería en Madrid. Una vez el petróleo ha sido refinado y se ha obtenido la gasolina, esta se envía a Barcelona para que sea almacenada en tanques. Para transportar la gasolina de Barcelona a Madrid se utiliza una red de oleoductos con estaciones de bombeo en las siguientes ciudades: Aranjuez, Villarrobledo, Albacete, Valencia, Castellón, Tortosa y Tarragona. La red está segmentada de tal forma que cada segmento une dos ciudades y por el cual no pueden circular más galones por hora de los especificados. La empresa desea conocer el número máximo de galones por hora que pueden enviarse a través de la red, ya que se aproximan días festivos y deben tener suficiente gasolina para abastecer a todos sus clientes.

Nota: La representación de la red difiere de la representación mostrada durante todo el trabajo debido a que este problema lo resolví utilizando el MATLAB durante la estancia de prácticas.

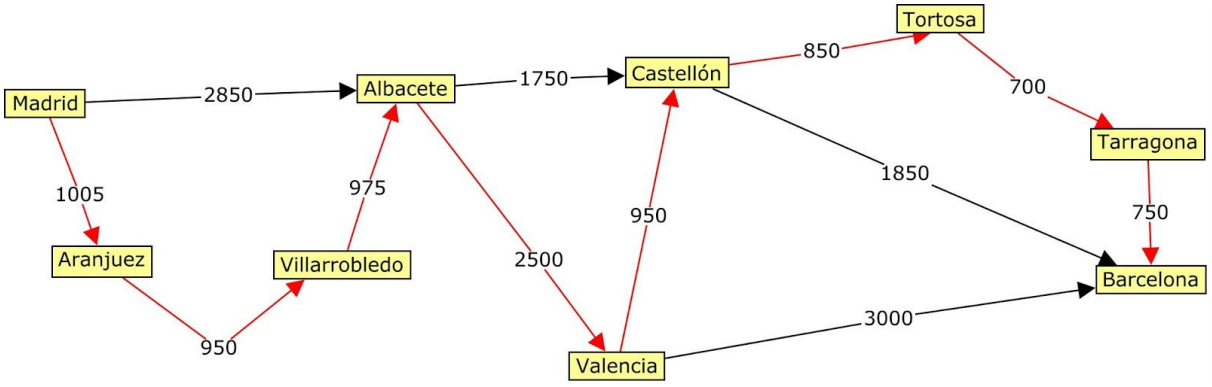
- Para modelar el problema utilizaremos una red dirigida con capacidad, donde las capacidades máximas representan los galones por hora de cada segmento de la red.



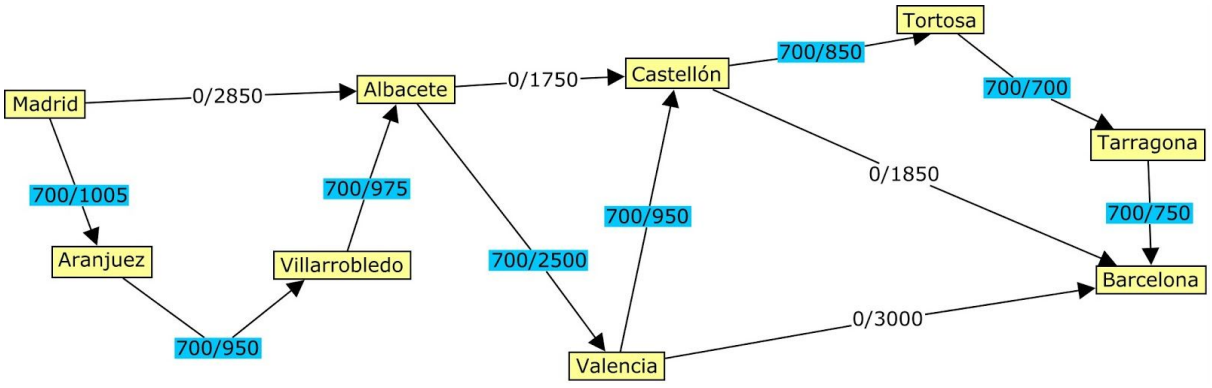
- A continuación obtenemos la red residual.

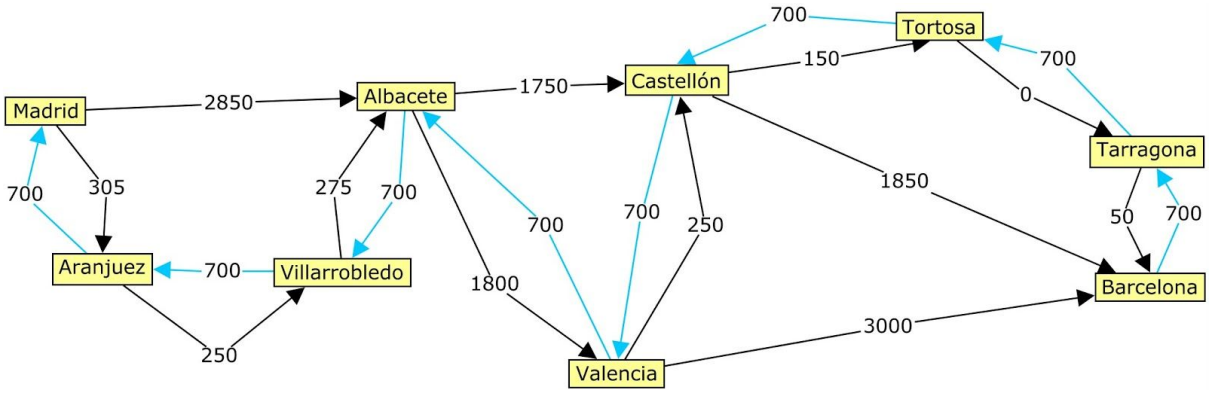


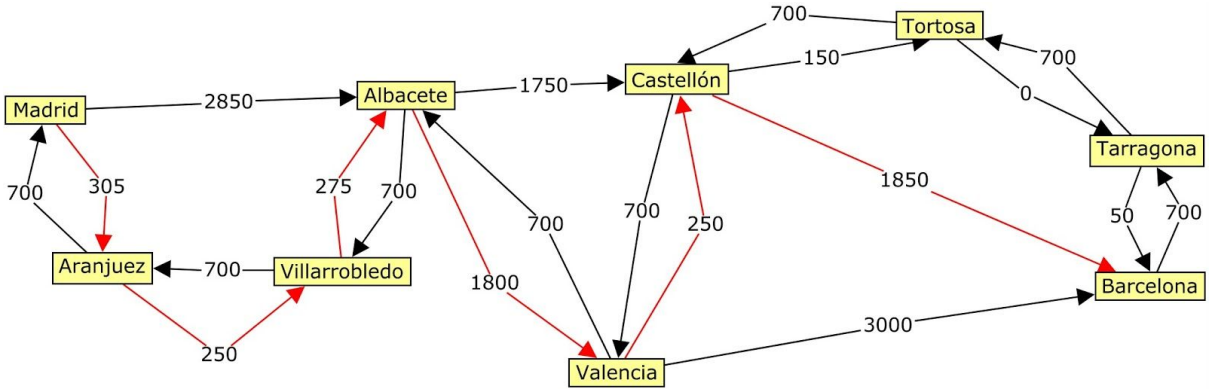
- Continuamos con este proceso, repitiendo los mismos pasos que en el ejemplo anterior, hasta que no podamos encontrar más caminos de aumento.



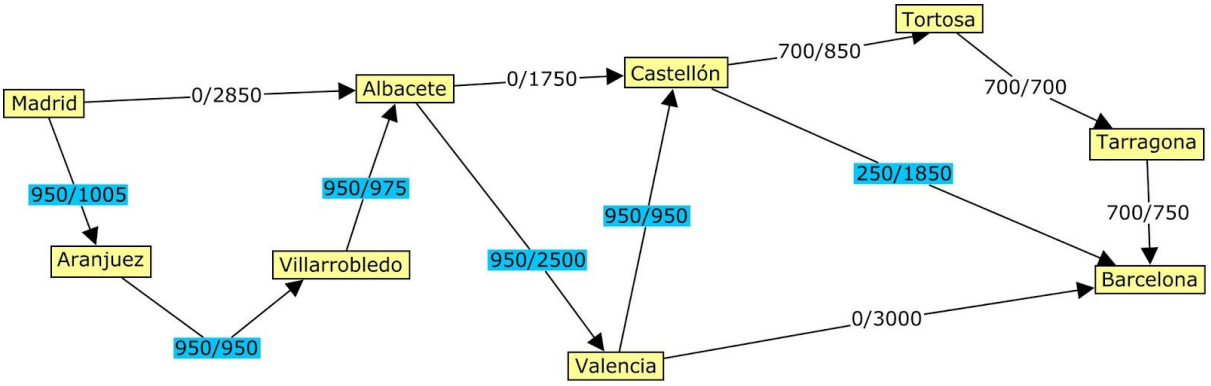
- Flujo = 0 + min(1005, 950, 975, 2500, 950, 850, 700, 750) = 700.

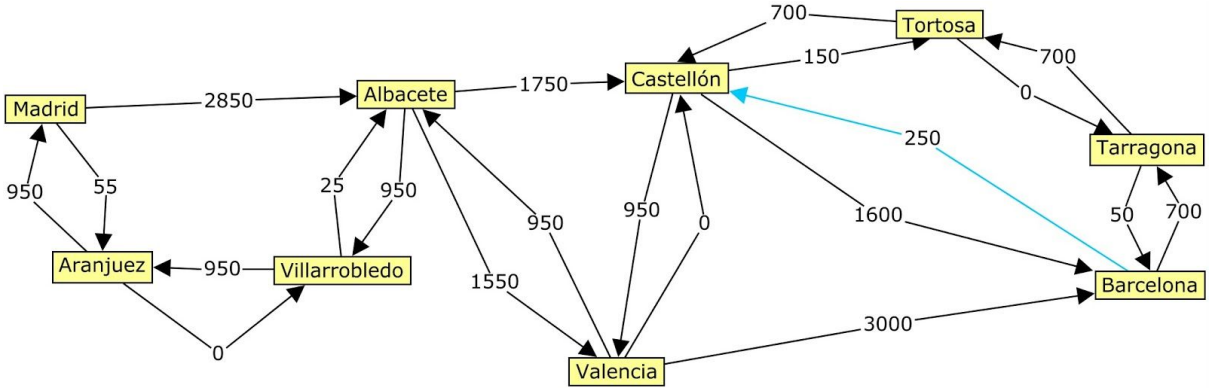


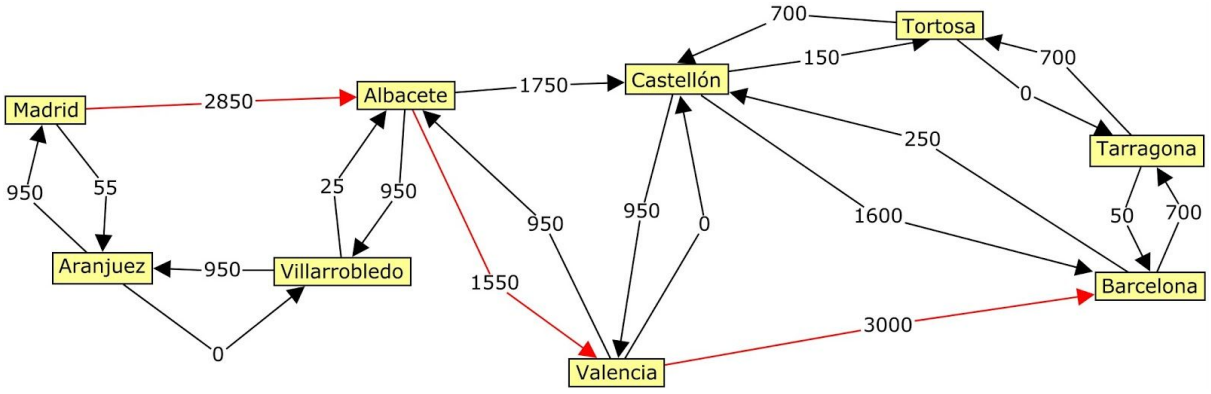




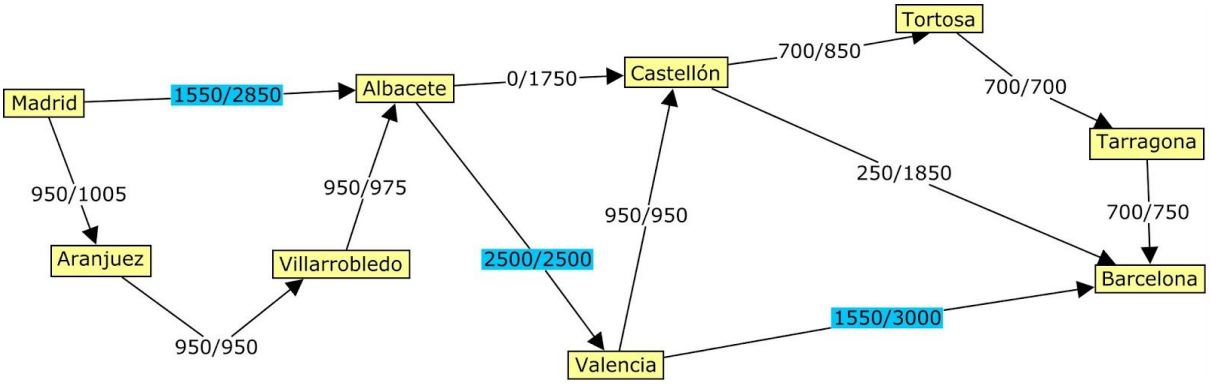
- Flujo = 700 + min(305, 250, 275, 1800, 250, 1850) = 950.

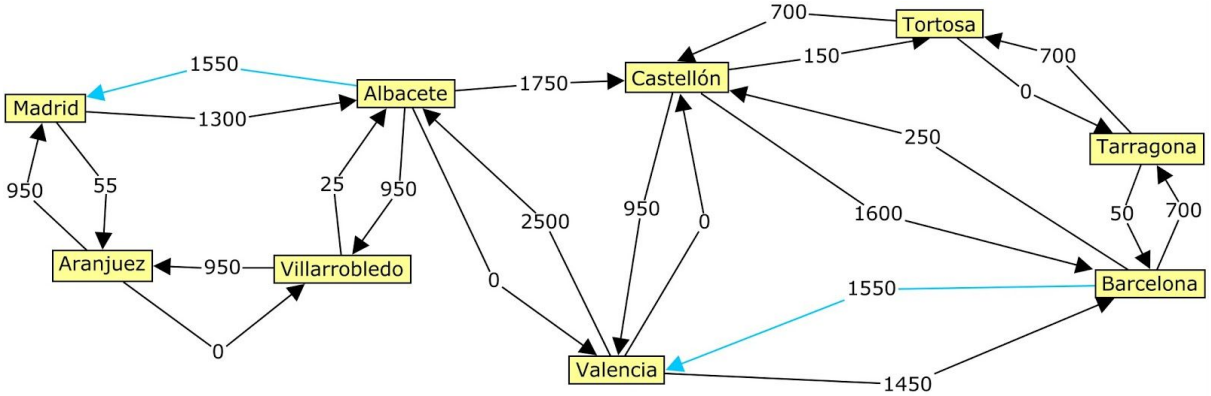


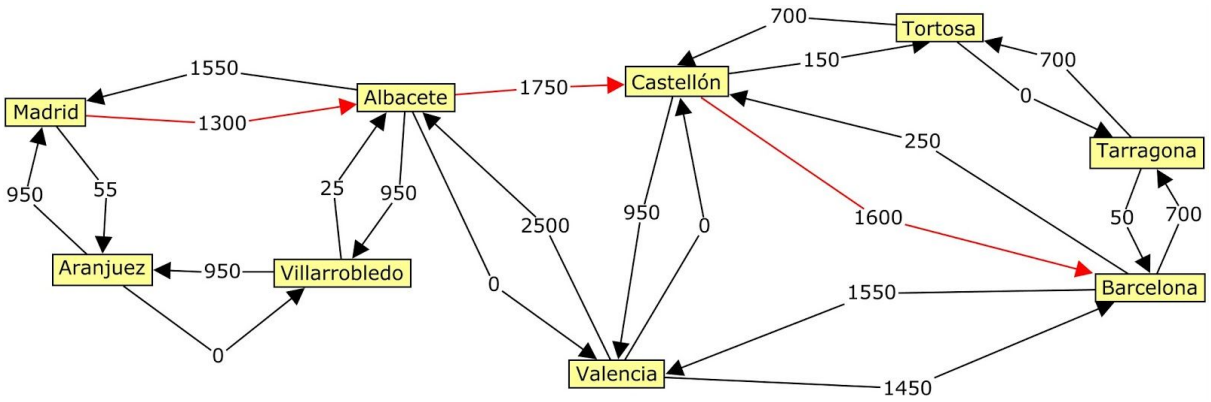




- Flujo = 950 + min(2850, 1550, 3000) = 2500.

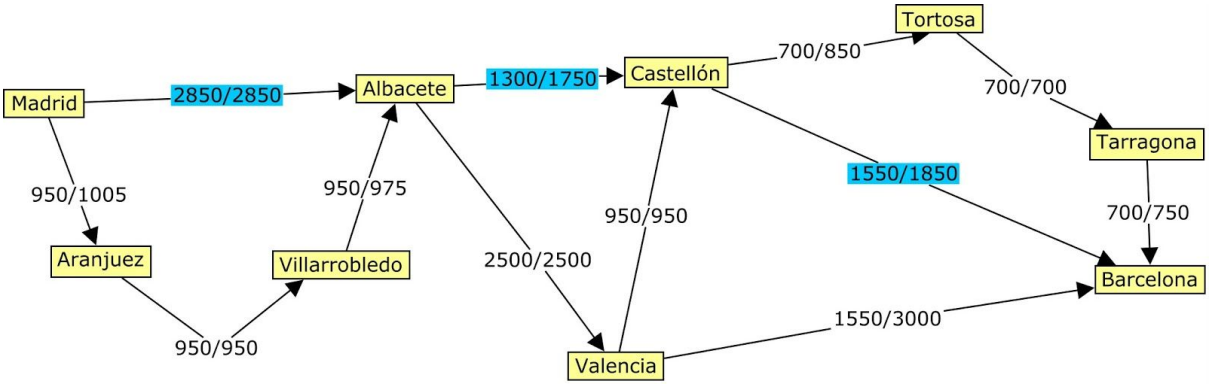


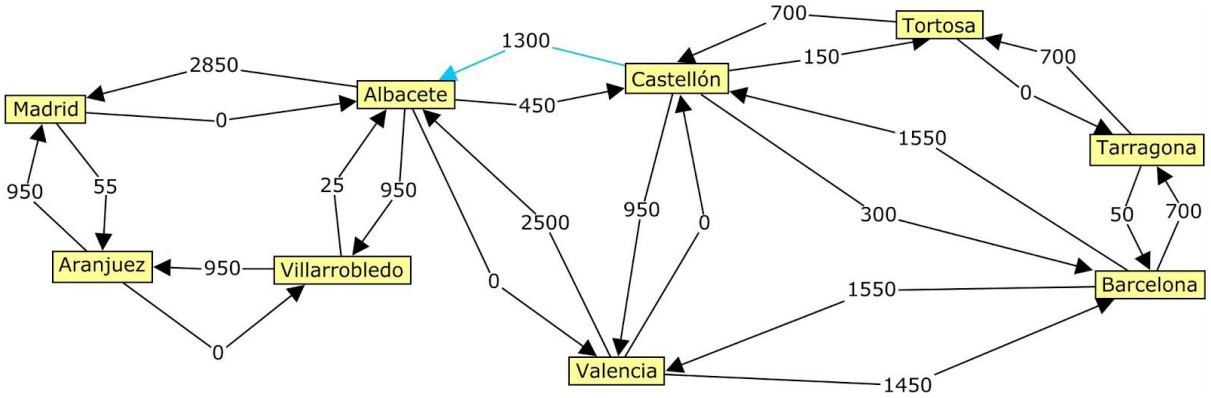




- Flujo = 2500 + min(1300, 1750, 1600) = 3800.

Nota: Llegados a este punto podemos observar que una vez estemos en Castellón podríamos haber escogido el camino hacia Valencia y de Valencia a Barcelona, en lugar de ir directamente hacia Barcelona. De escoger dicho camino hubiésemos tenido que realizar una iteración más para conseguir el flujo máximo.





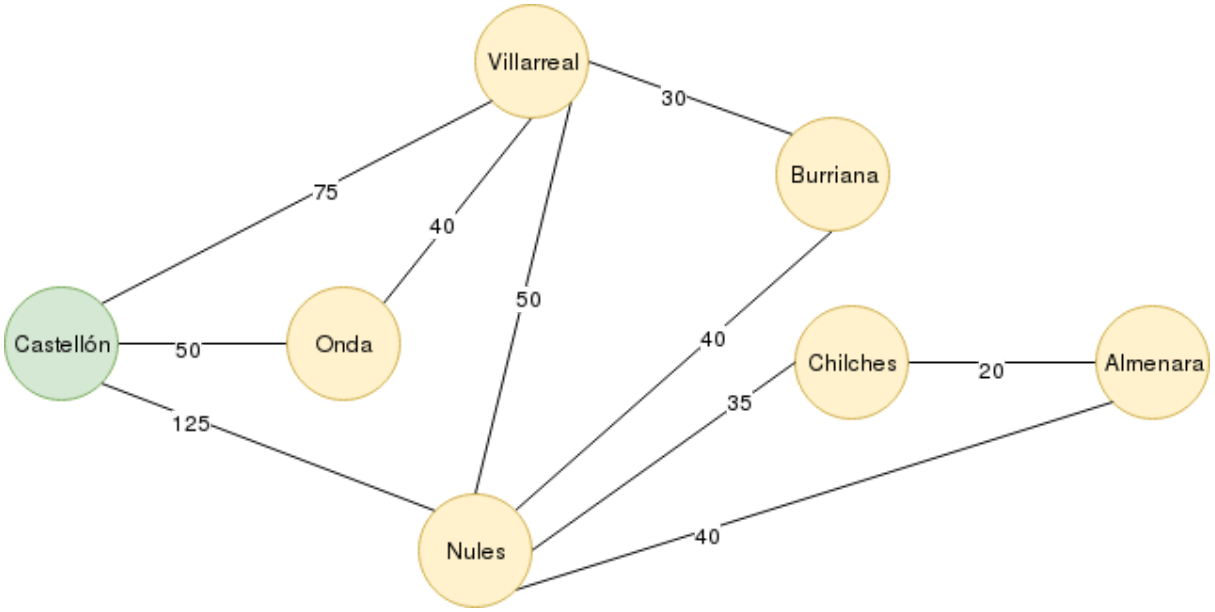
Llegados a este punto el algoritmo ya no puede encontrar más caminos de aumento y, por tanto, finaliza. Como resultado hemos obtenido la cantidad máxima de galones por hora que pueden circular por la red y por donde han de circular para llegar a esa cantidad.

$$val(f) = \sum_{arista\ que\ sale\ de\ s} f(arista) = 950 + 2850 = 3800\,.$$

7.2 Solución del problema de expansión de una red de fibra óptica aplicando Prim

Una compañía de fibra óptica desea expandir su red por la província de Castellón. Para ello ha escogido 7 localidades con diferentes caminos que las enlazan y a cada camino le ha asignado el coste correspondiente a la suma del coste por obras, que establece cada ayuntamiento, y del coste por la longitud de los materiales a emplear(tubos, cable de fibra, protecciones, etc). La empresa desea conocer qué caminos debe escoger para que el coste de expansión de la red de fibra óptica sea mínimo.

- Empezamos modelando el problema según las localidades y costes que nos ha facilitado la empresa:

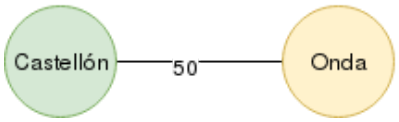


- Como podemos observar estamos ante una red no dirigida con coste. En el apartado 5 hemos estudiado cómo podemos obtener un árbol de expansión mínimo a partir de este tipo de modelado.

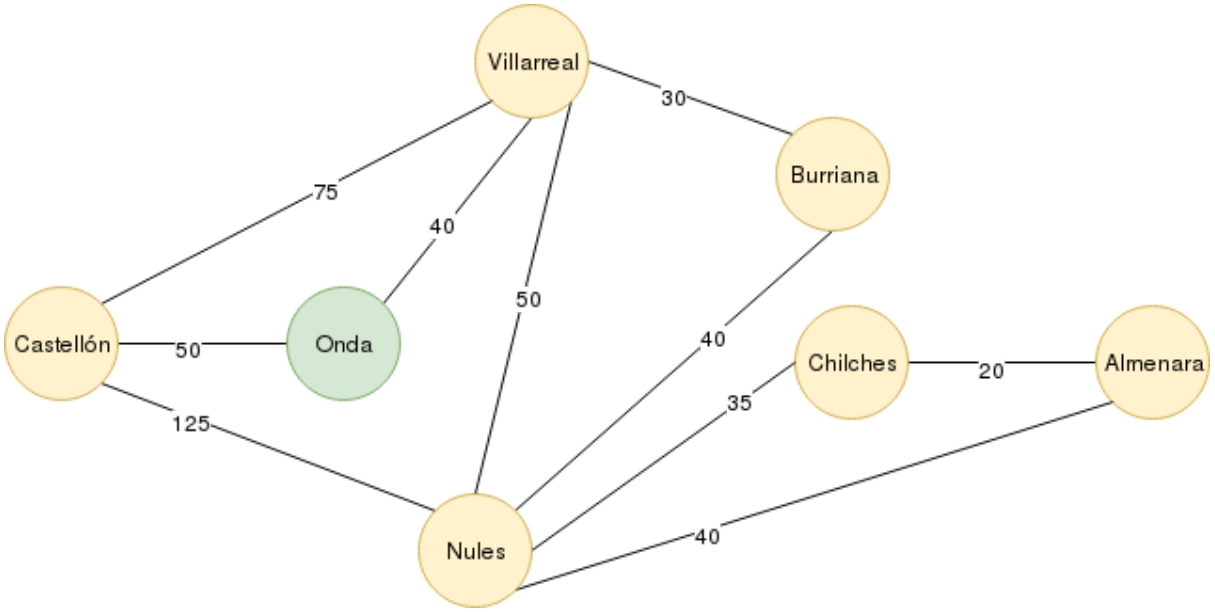
- A continuación, vamos a aplicar el algoritmo de Prim empezando desde Castellón. En primer lugar, comparamos el coste de cada una de las aristas que enlazan los pueblos adyacentes a Castellón. Escogemos Onda ya que es la de menor coste, enlazamos los pueblos en el árbol y insertamos el resto de pueblos adyacentes a la cola. Repetiremos este proceso en cada iteración.

Árbol

CP =[(Villareal, 75), (Nules, 125)]



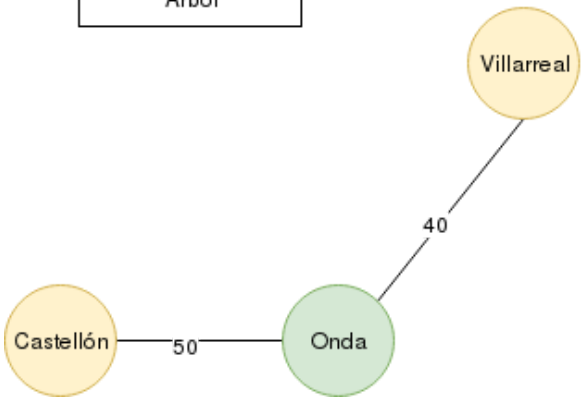
- Repetimos el paso anterior pero esta vez empezando desde Onda.



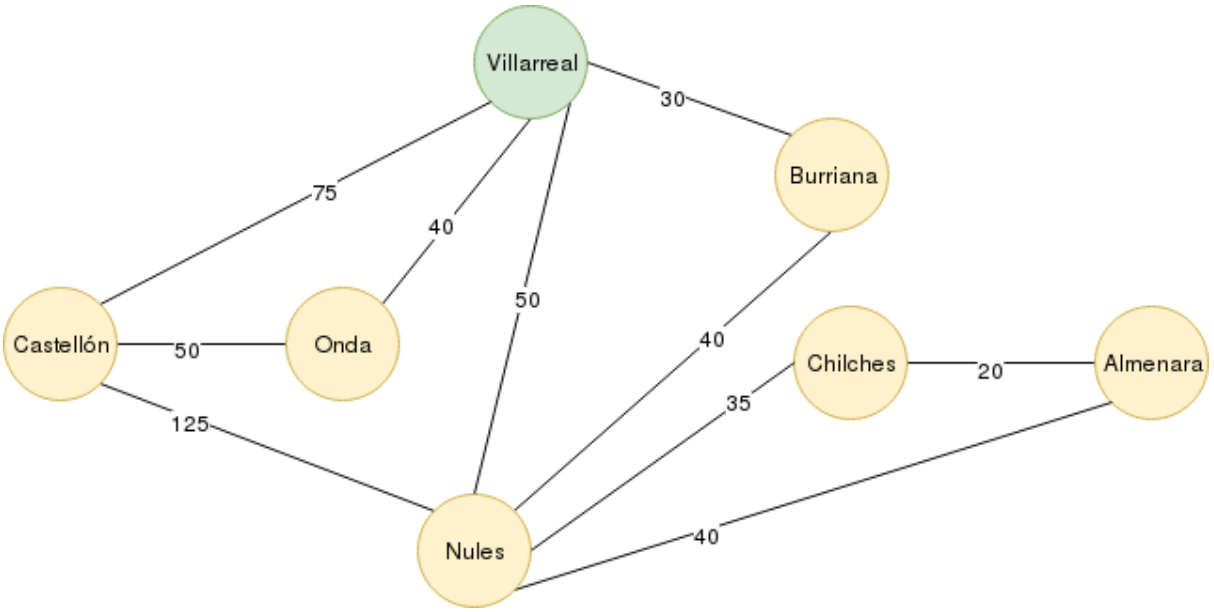
- Eliminamos de la cola a Villarreal ya que lo enlazaremos a través de Onda.

Árbol

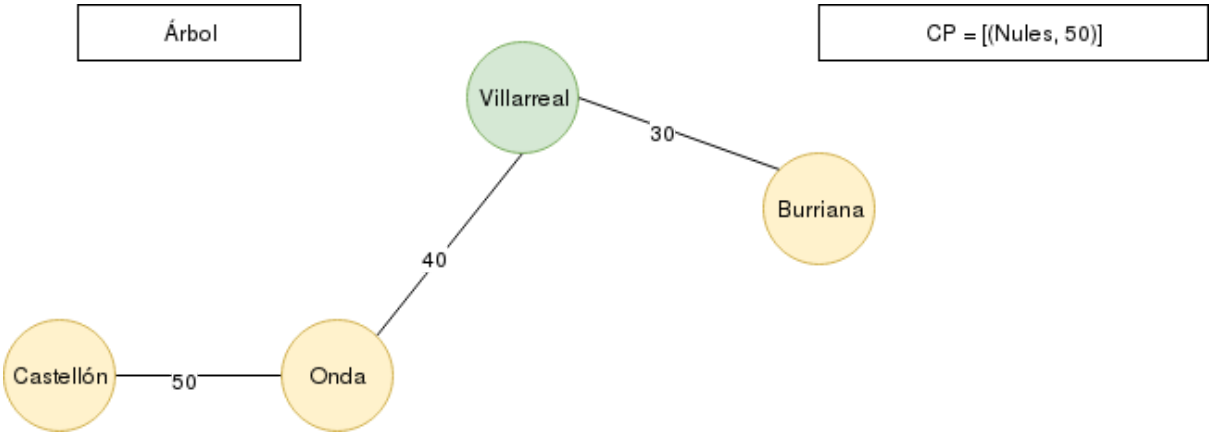
CP = [(Nules, 125)]



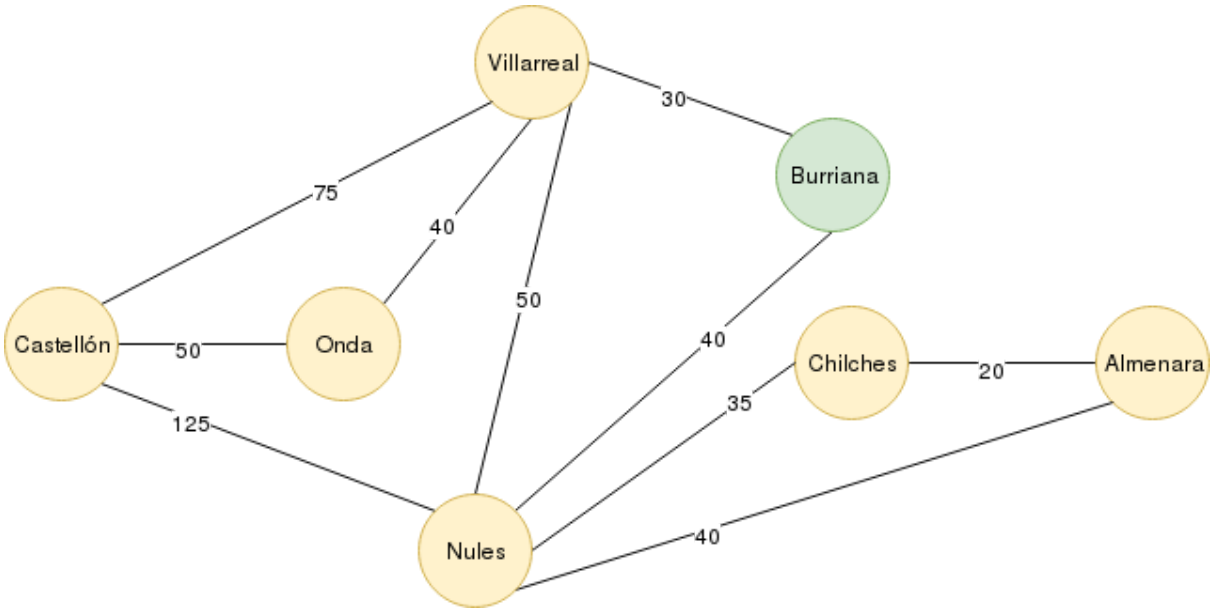
- Seguimos iterando, esta vez partiendo de Villarreal.



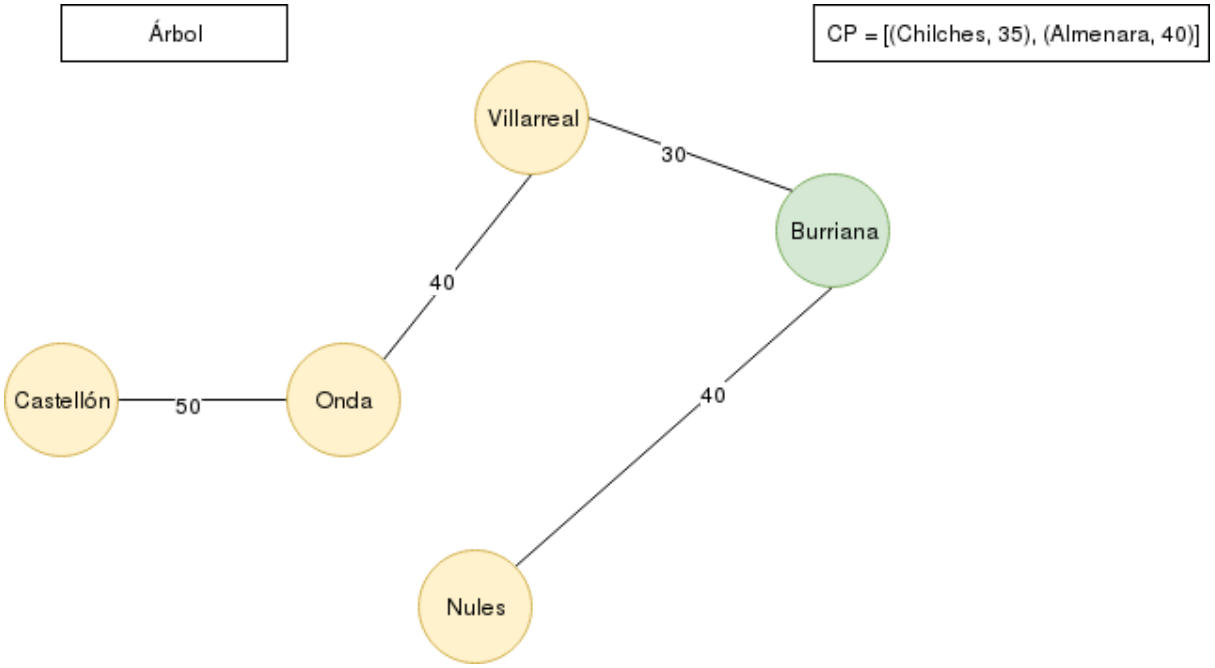
- Actualizamos la distancia de Nules en la cola a la obtenida partiendo de Villarreal.



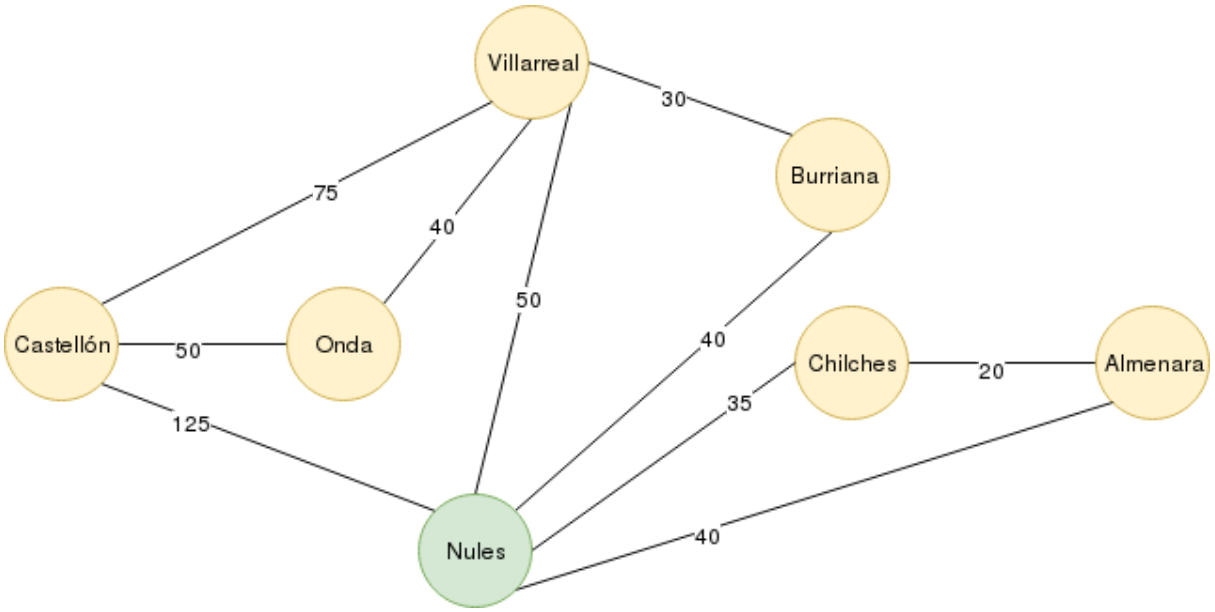
- La siguiente iteración la realizamos partiendo de Burriana.



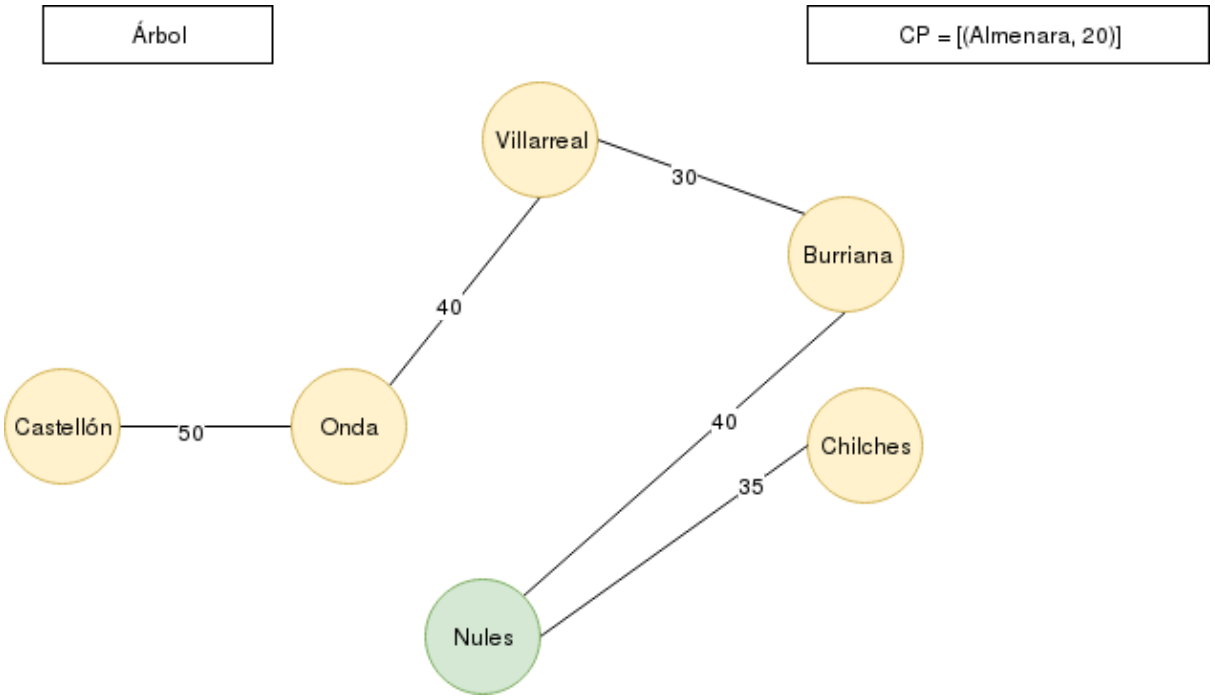
- Eliminamos a Nules de la cola ya que lo enlazamos a través de Burriana.



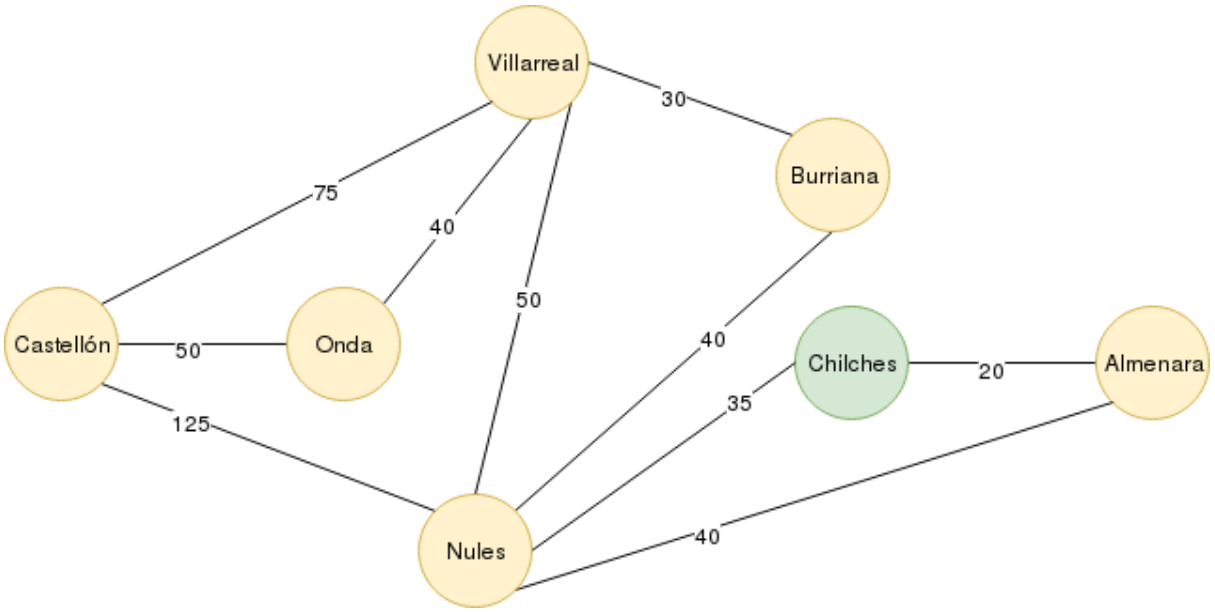
- Ahora, partiendo de Nules, seguimos iterando.



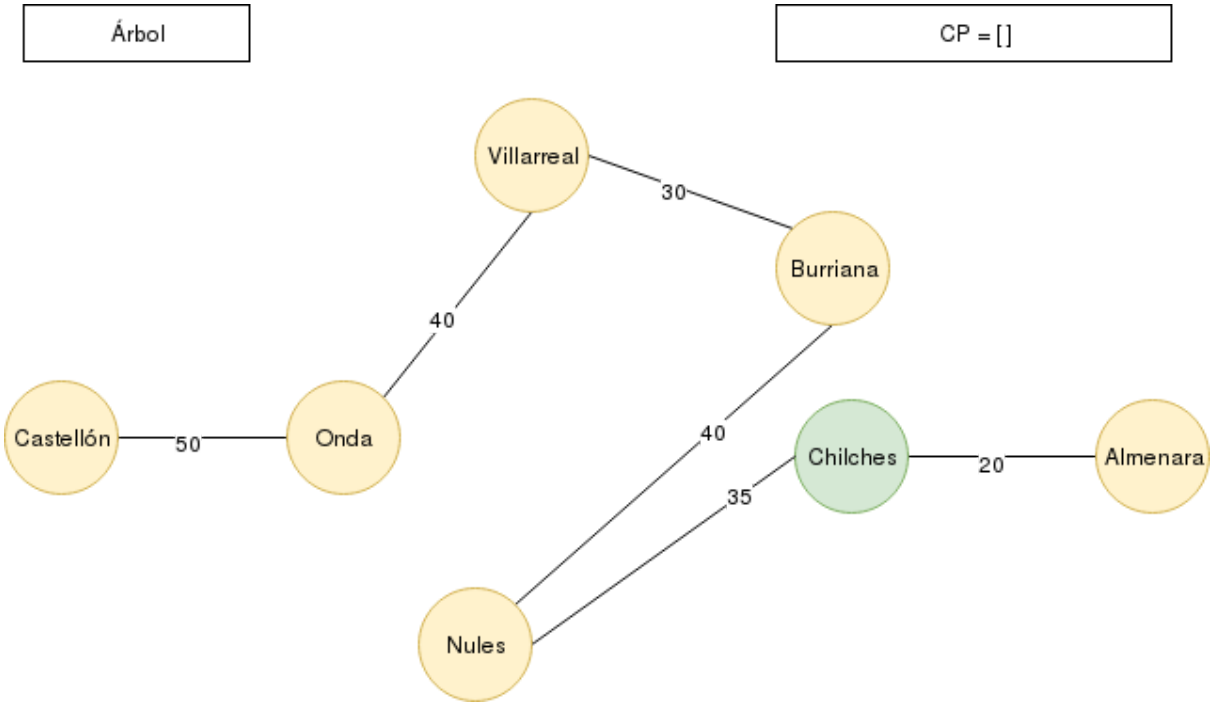
- Eliminamos a Chilches de la cola y actualizamos la distancia de Almenara en la cola a la obtenida partiendo de Nules.



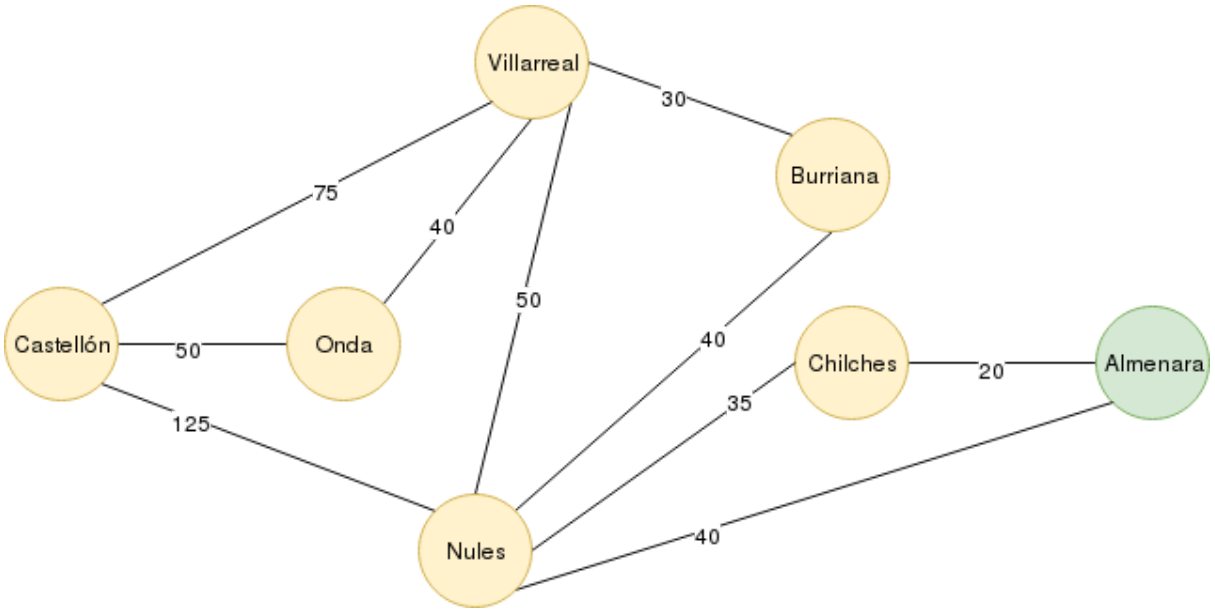
- La penúltima iteración la realizaremos partiendo de Chilches.



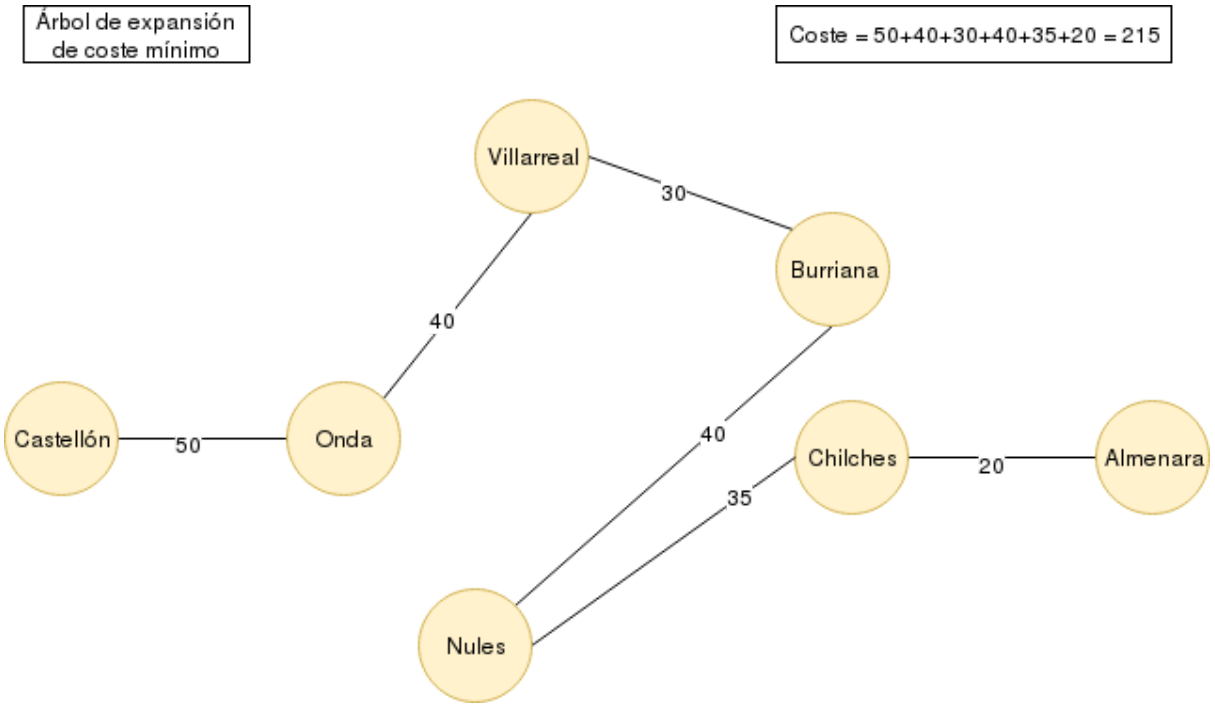
- Enlazamos el último pueblo de la red.



- Terminamos de iterar. Ya hemos visitado todos los pueblos.



- El resultado es el árbol de expansión de coste mínimo que representa el camino a seguir para que el coste de la expansión de la red de fibra óptica sea mínimo(215).

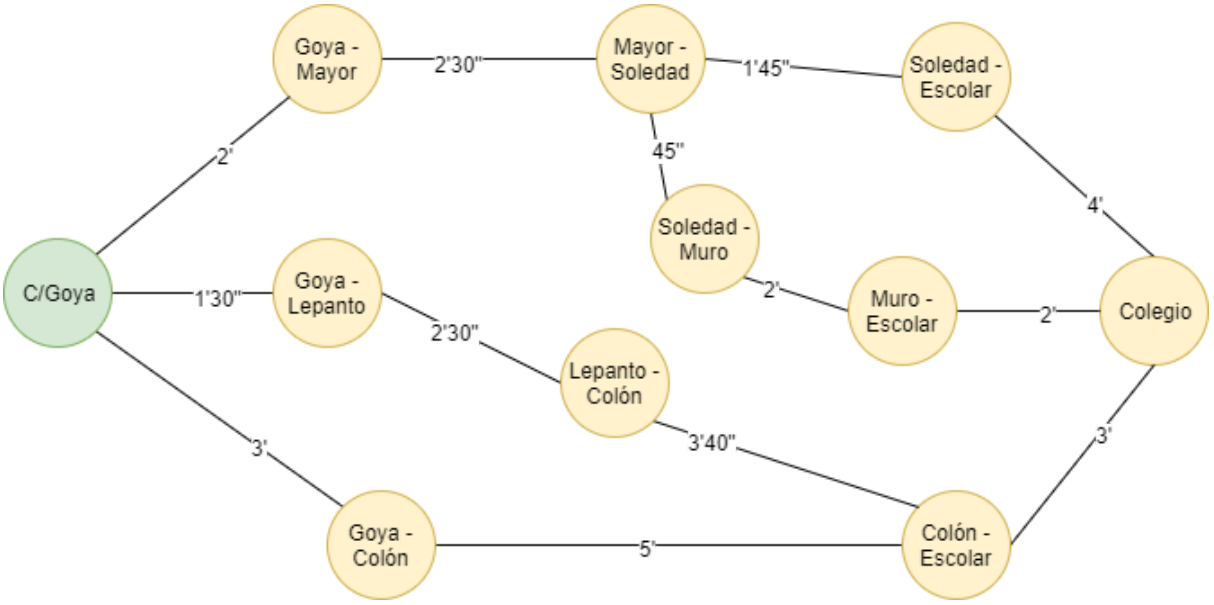


7.3 Solución al problema de selección de la ruta más corta aplicando Dijkstra

Un CEO de una multinacional quiere llevar a sus hijos al colegio con la condición de llegar puntual a la oficina. Para ello, ha recolectado información sobre las calles que enlazan su casa con el colegio. El tiempo medio para llegar del colegio a la oficina es de 10 minutos. Saliendo de casa a las 08:40 dispone de 10 minutos para dejar a sus hijos en la puerta del colegio. Los datos facilitados son las distancias de las calles y el tiempo medio que se tarda en recorrer 100 metros en cada una de ellas. ¿Es posible llegar a tiempo a la oficina saliendo de casa a las 08:40 y dejando a sus hijos en la puerta del colegio?

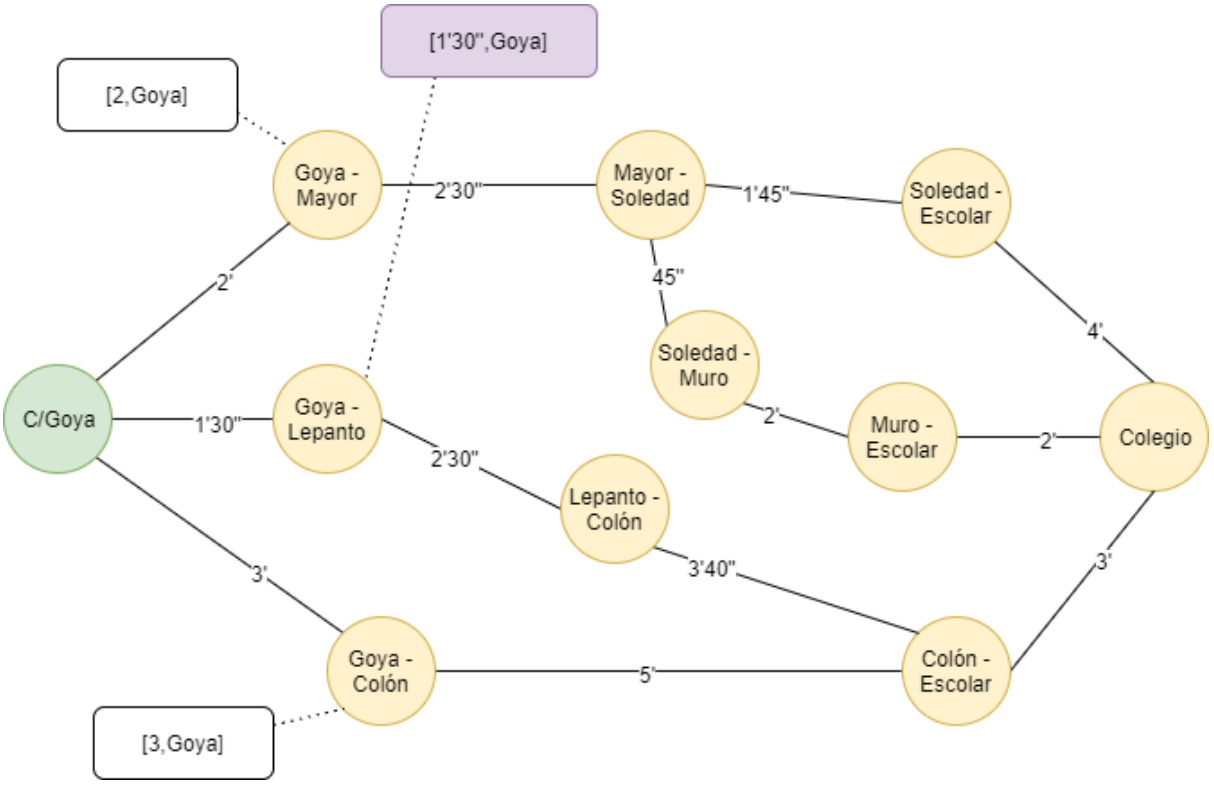
Para modelar este problema es necesario que definamos el valor longitud. Para ello, dividiremos todas las distancias entre 100 y las multiplicaremos por el tiempo medio asignado. De tal forma que la longitud representará un valor temporal en el que el algoritmo se apoyará para hallar el camino más corto.

- Empezamos desde la casa del CEO, ubicada en la calle Goya.

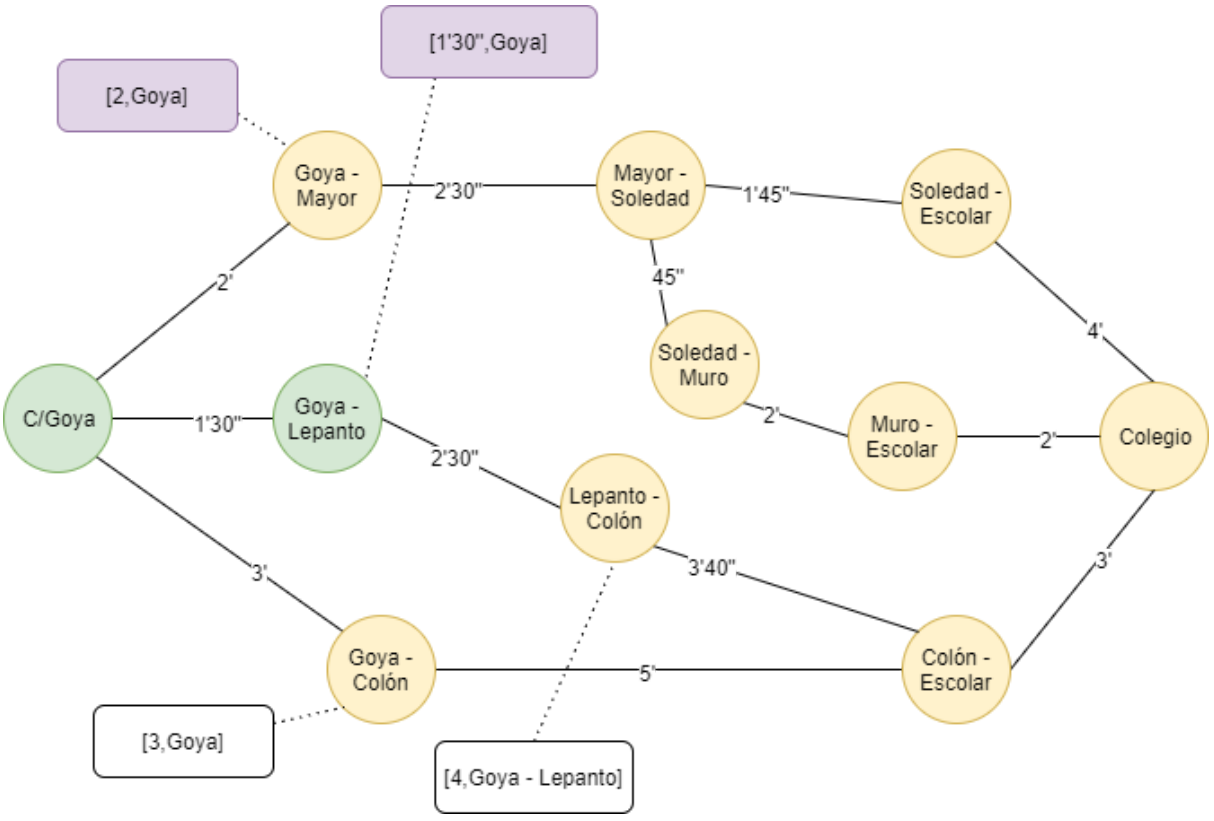


- Como hemos visto en el apartado 6, utilizaremos la notación **[longitud, calle]** para indicar el tiempo medio que se tarda en llegar a la intersección viniendo de la calle indicada.

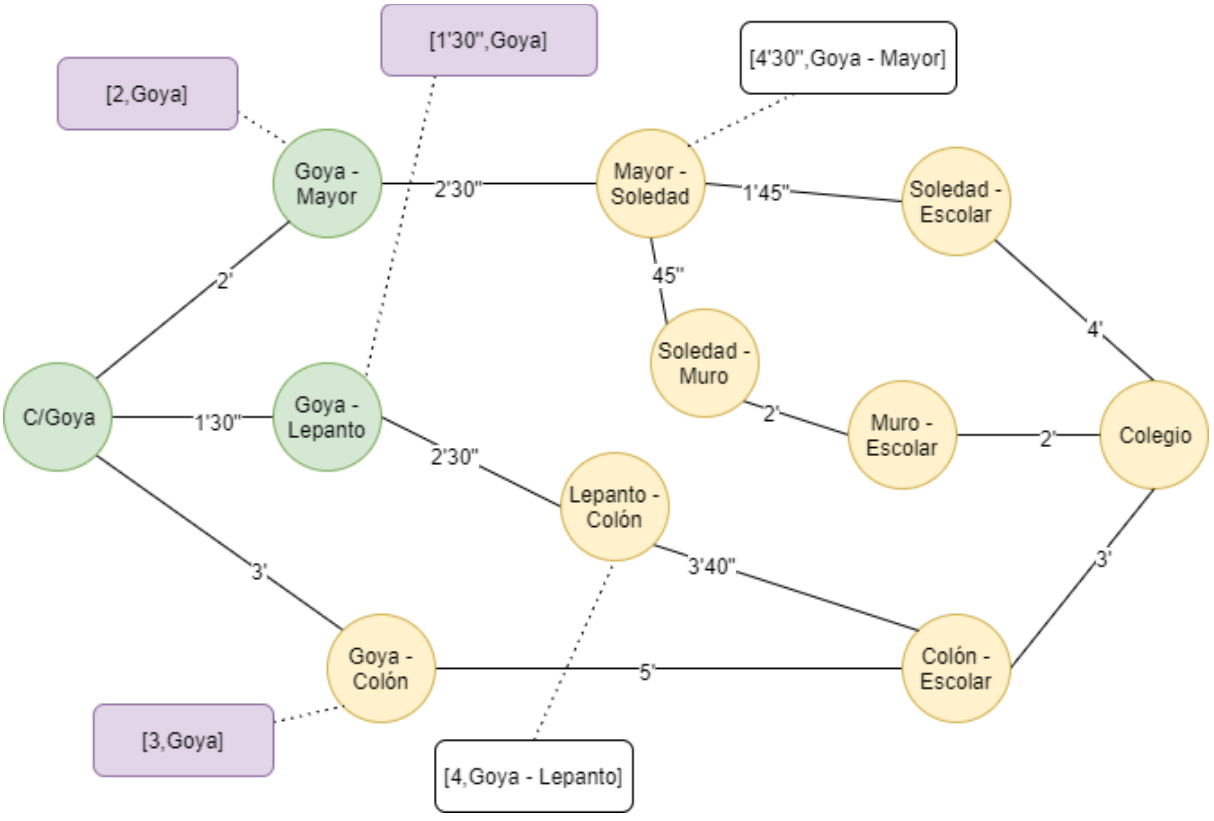
- Vamos a iterar el paso descrito a continuación hasta que hayamos visitado todas las intersecciones. En primer lugar, anotamos la suma de la longitud y la calle proveniente en cada una de las intersecciones adyacentes. En este caso, no habrá que sumar nada para obtener las longitudes, ya que la longitud inicial es 0. Una vez, calculada la longitud, seleccionamos la de menor peso.

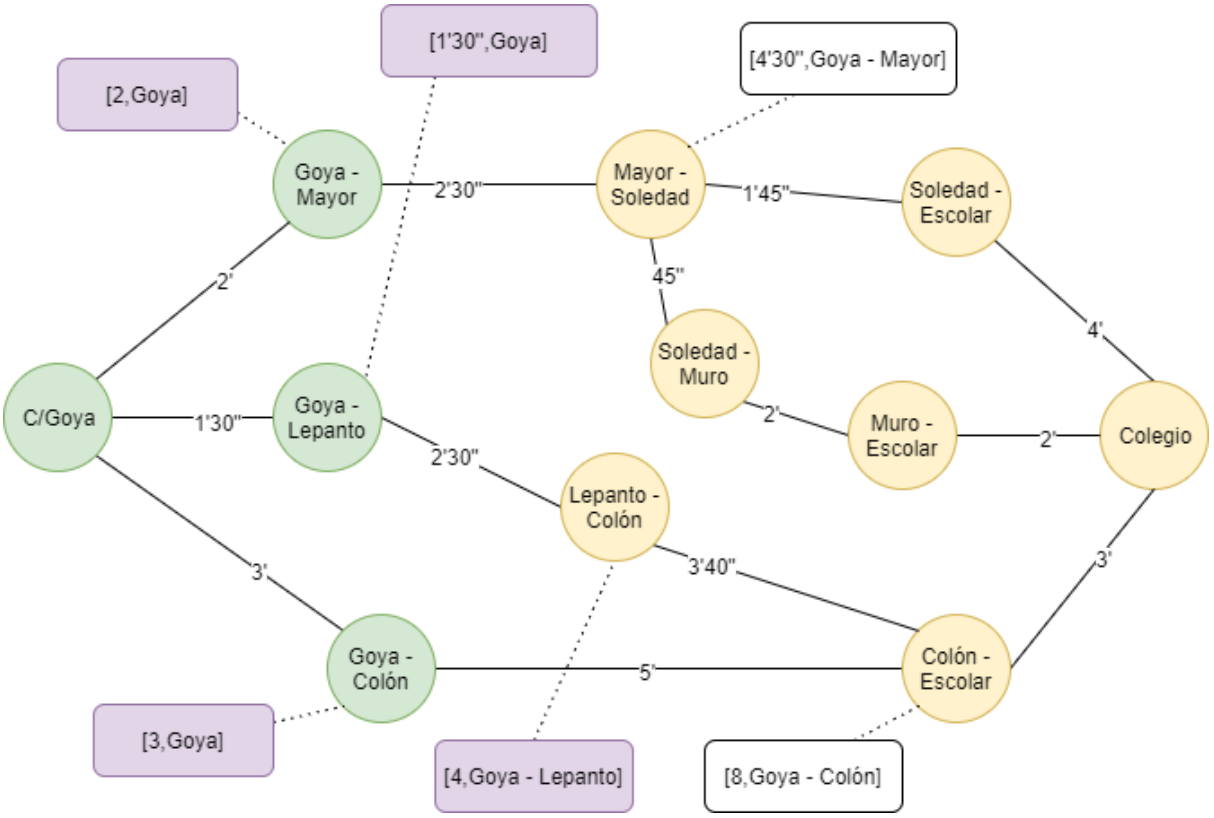


- A continuación, nos situamos sobre la intersección que posee la menor longitud, la marcamos como visitada y repetimos el paso anterior.

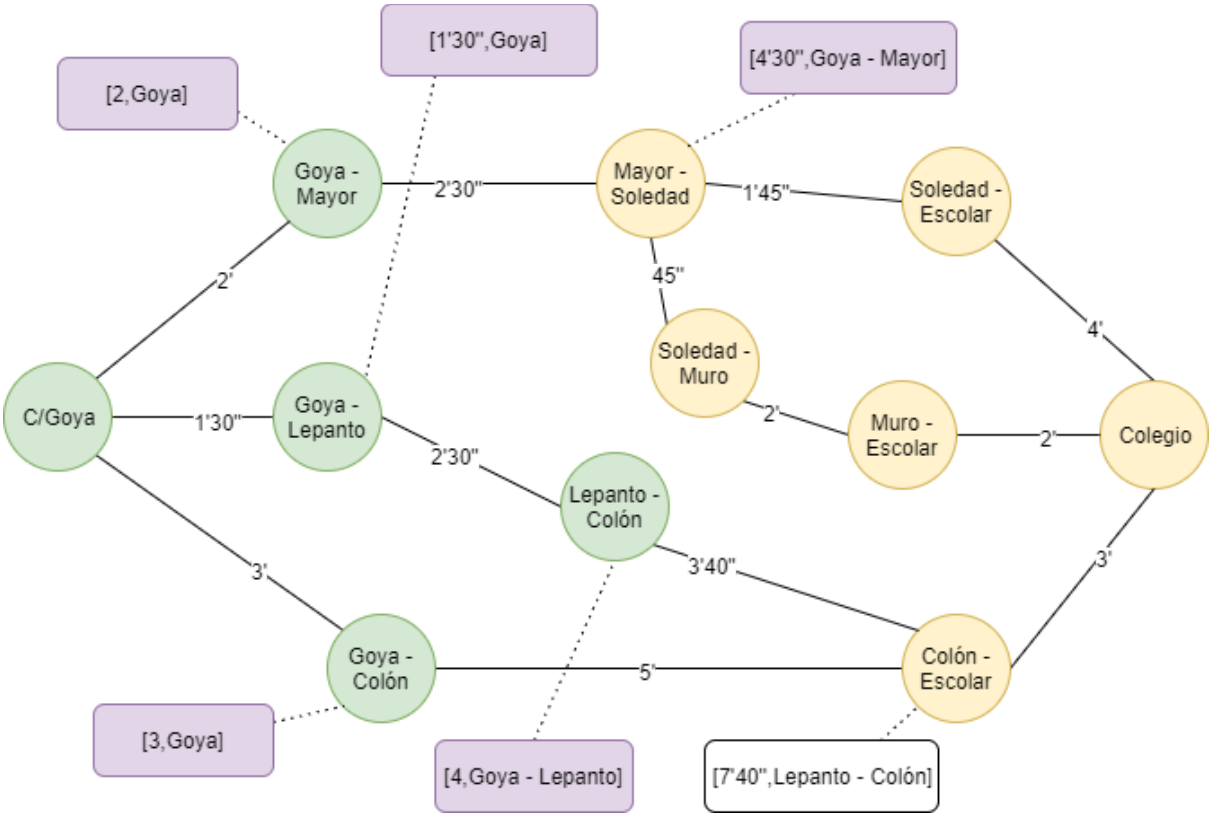


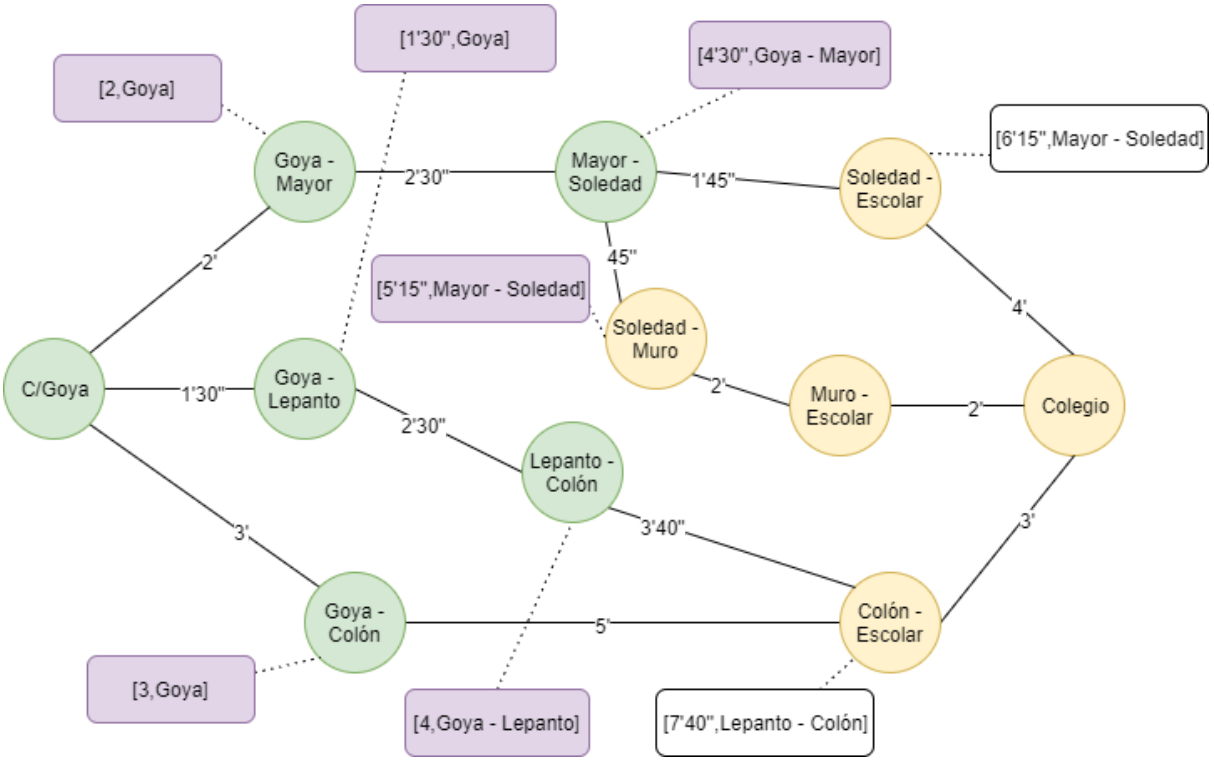
- Seguimos iterando hasta que no queden intersecciones que visitar o marquemos Colegio como visitado.

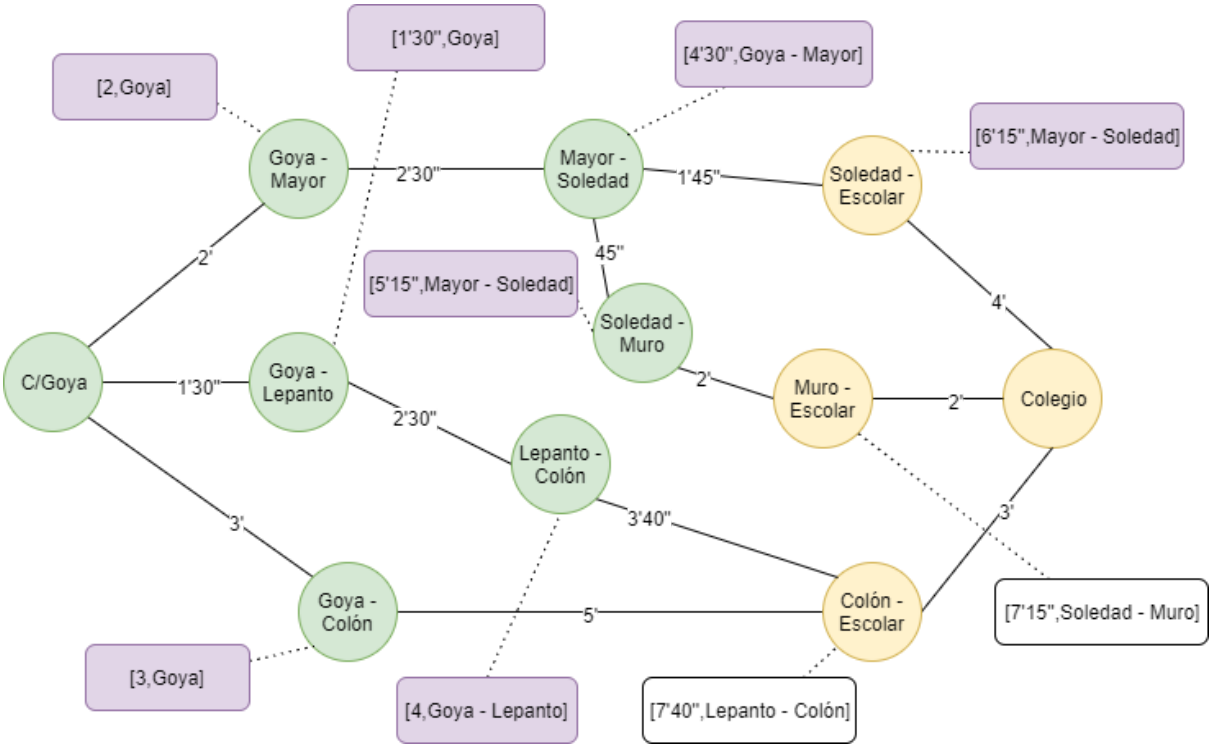


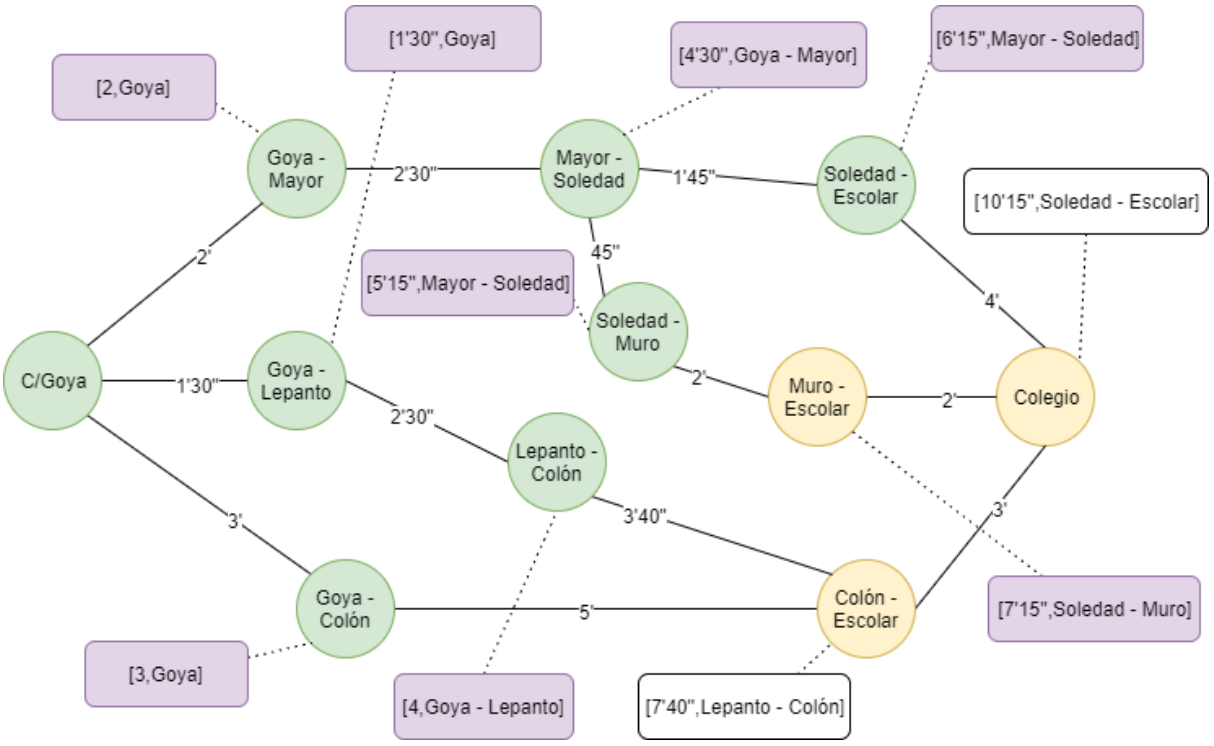


- Actualizamos la longitud de la intersección Colón-Escolar a la obtenida viniendo de Lepanto-Colón.

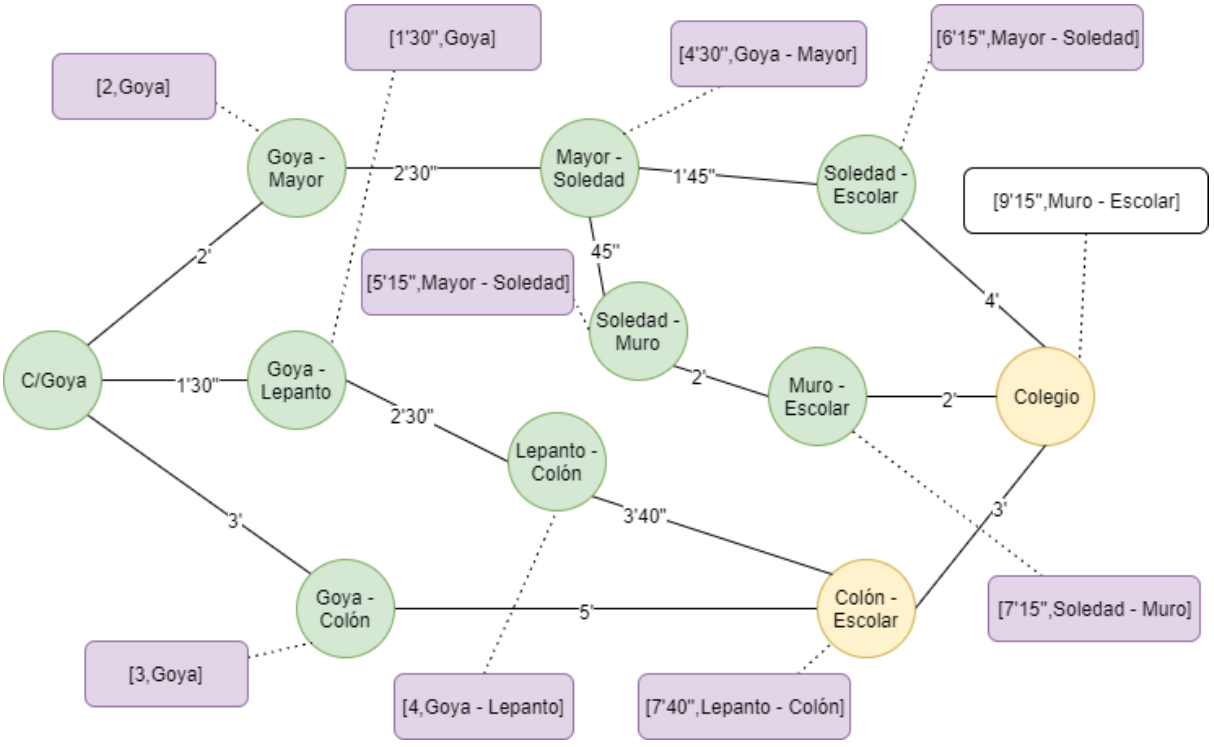



[1'30",Goya]
[2,Goya]
[4'30",Goya - Mayor]
[6'15",Mayor - Soledad]
Goya - Mayor
2'30"
Mayor - Soledad
1'45"
Soledad - Escolar
2'
45"
4'
[5'15",Mayor - Soledad]
Soledad - Muro
C/Goya
1'30"
Goya - Lepanto
2'
Muro - Escolar
2'
Colegio
2'30"
Lepanto - Colón
3'
3'40"
3'
Goya - Colón
5'
Colón - Escolar
[3,Goya]
[4,Goya - Lepanto]
[7'40",Lepanto - Colón]

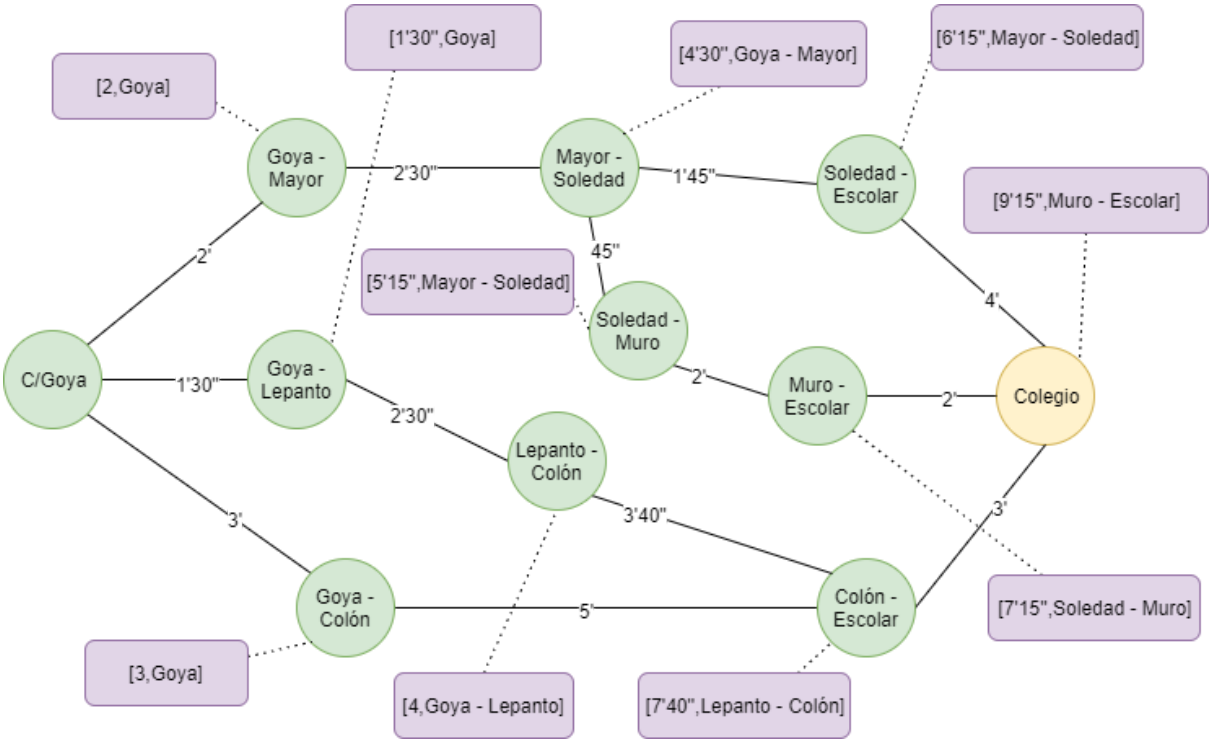

[1'30",Goya]
[2,Goya]
[4'30",Goya - Mayor]
[6'15",Mayor - Soledad]
Goya - Mayor
2'30"
Mayor - Soledad
1'45"
Soledad - Escolar
2'
45"
4'
[5'15",Mayor - Soledad]
Soledad - Muro
C/Goya
1'30"
Goya - Lepanto
2'
Muro - Escolar
2'
Colegio
2'30"
Lepanto - Colón
3'
3'40"
3'
Goya - Colón
5'
Colón - Escolar
[7'15",Soledad - Muro]
[3,Goya]
[4,Goya - Lepanto]
[7'40",Lepanto - Colón]



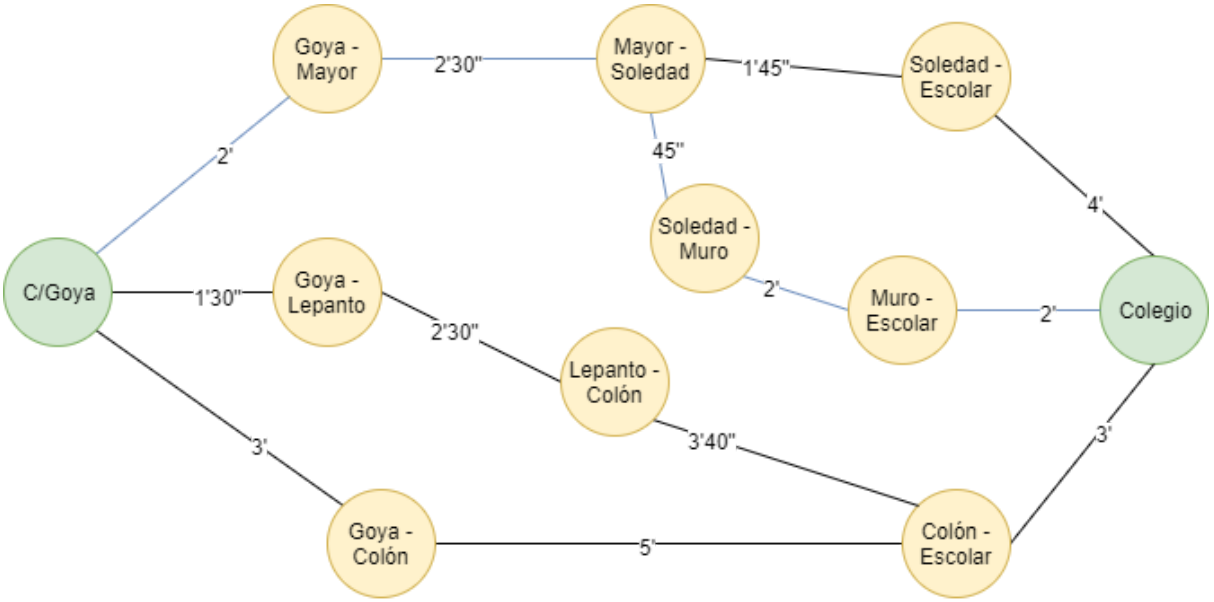
- Actualizamos la longitud de Colegio a la obtenida viniendo de Muro-Escolar.



- Visitamos la última intersección y finalizamos.



- El resultada es el camino mínimo y el tiempo medio de este.



Camino mínimo = Goya - Mayor - Soledad - Muro - Escolar - Colegio

Tiempo medio = 9'15"

- El CEO, siguiendo el camino mínimo, conseguirá llegar puntual a la oficina, ya que el tiempo límite eran 10 minutos.

8. Conclusiones

Empecé el trabajo con una idea que ha terminado siendo remodelada completamente y hecha realidad. El estudio sobre la investigación operativa ha resultado ser gratificante y me resultará de utilidad en el futuro. La forma en que se afrontan los problemas en esta disciplina es aplicable a los problemas que nos surgen en nuestro día a día; pudiendo tomar mejores decisiones. Durante el desarrollo del trabajo, Jorge y Álex han sido un guía en la que me he podido apoyar siempre que me ha sido necesario. Me han ayudado a comprender muchas definiciones y partes de los algoritmos que me resultaban difíciles y con ello poder completar el trabajo. Por lo que respecta a los algoritmos, decidí trabajar en ellos ya que son los más utilizados para resolver los problemas estudiados. Para finalizar, tenemos que tener en cuenta que existen más algoritmos para resolver un mismo tipo de problema y que dependiendo del modelado y el objetivo podemos aplicar el que nos resulte más conveniente.

9. Bibliografía

01. http://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/
02. http://flujomaximo.blogspot.com.es/#!
03. http://ma1.eii.us.es/Material/FTG_itis_Tema4.pdf
04. http://lic.mat.uson.mx/tesis/82TesisIrene.PDF
05. http://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/07NetworkFlowI.pdf
06. http://wcipeg.com/wiki/Max-flow_min-cut_theorem
07. http://www-2.dc.uba.ar/personal/fbonomo/grafos/curso_grafos_flujo_handout.pdf
08. http://www.webdelprofesor.ula.ve/ingenieria/ibc/ayda/c20flujoMaximo.pdf
09. Introduction to algorithms - Thomas H. Cormen, Charles E. Leiseron, Ronald R. Rivest, Clifford Stein.
10. https://es.wikipedia.org/wiki/Investigación_de_operaciones
11. http://www.uv.es/martinek/material/Tema7.pdf
12. https://es.wikipedia.org/wiki/Algoritmo_de_Prim

10. ANEXO: Memoria de prácticas

Resumen

Se presenta el proceso seguido para poder entender y analizar el algoritmo de optimización de Boykov. Como resultado se muestra el funcionamiento de cada parte del código según los datos iniciales que le facilitemos al algoritmo, así como el funcionamiento a nivel global. Con todo ello se exponen los diferentes modos de funcionamiento dependiendo de qué condiciones se utilicen para preparar los vectores *unary* y *labelcost* que emplea el algoritmo.

Índice

1. Introducción

En este documento vamos a mostrar todo lo aprendido durante la estancia en el INIT, así como las conclusiones que podemos sacar del trabajo realizado. Empezaremos con la instalación del software matemático MATLAB y terminaremos con las conclusiones del análisis del código de minimización de la energía, de Boykov, pasando por las lecturas de los artículos y documentos facilitados por el tutor de prácticas y el análisis de los algoritmos de optimización con MATLAB.

Todos los artículos, documentos y ficheros se adjuntarán junto a la memoria.

2. Instalación del software y puesta a punto

Durante la primera reunión con el tutor de prácticas establecimos los objetivos y las herramientas necesarias para poder llevarlos a cabo. A continuación, redacté la propuesta técnica para que todo lo hablado en la reunión estuviera establecido en un periodo de tiempo.

Una vez entregada la propuesta técnica pasé a realizar la instalación del software matemático MATLAB, donde tuve distintos problemas. El primero fue debido a que mi usuario en el ordenador del INIT no tenía suficientes permisos y no se podía proceder a la instalación. Para solucionarlo, el tutor de prácticas me facilitó sus datos identificativos y pude instalar el software con su usuario. Un par de días después tuvieron que formatear todos los ordenadores y los programas instalados se perdieron, por lo que tuve que volver a realizar la instalación completa del software.

Posteriormente, descargué todos los documentos, artículos y archivos comprimidos que el tutor me facilitó, los guardé en el ordenador y prepare todos los directorios y archivos necesarios para poder analizarlos con MATLAB.

El periodo de tiempo que supuso llevar a cabo la instalación del software y puesta a punto fue del **06/02/2017** al **12/02/2017**.

3. Algoritmos de optimización

Durante la semana del **13/02/2017** al **19/02/2017** estudié los distintos algoritmos de optimización que se explican en el documento cuya referencia web es : http://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/07NetworkFlowI.pdf .

En su lectura aprendí los conceptos de capacidad, flujo, red, red residual, camino de aumento, cuello de botella, flujo máximo, corte y corte mínimo, que explicaré a continuación:

- Capacidad: Es una función $c(arista)$ que devuelve la capacidad de unidades homogéneas que pueden circular por una arista.

- Flujo: Es una función $f(arista)$ que devuelve la capacidad actual de unidades homogéneas que están circulando por una arista.

- Red: Es un conjunto de un conjunto de nodos, N, y otro de aristas, E, al que se le asocia una función de capacidad y otra de flujo. La red la podemos identificar por $G(N, E, c, f)$.

- Red residual: Es una red compuesta por un conjunto de nodos, N, y otro de aristas, E_f, que es la unión del conjunto de aristas E con sus aristas residuales, a la que se le asocia una función de capacidad y otra de flujo.

- Camino de aumento: Es un conjunto ordenado de nodos y aristas, donde no puede haber nodos repetidos y debe empezar en el nodo fuente y terminar en el nodo sumidero.

- Cuello de botella: Es el mínimo de las capacidades de las aristas que forman el camino de aumento.

- Flujo máximo: Es la cantidad máxima de unidades homogéneas que pueden circular por una red.

- Corte: Es una división del conjunto de nodos, N, en dos subconjuntos disjuntos.

- Corte mínimo: Es el corte cuya suma de las capacidades de las aristas que apuntan de un subconjunto de N hacia el otro es mínima.

El funcionamiento del algoritmo de Ford-Fulkerson fue el que decidí estudiar en mayor profundidad ya que, desde mi punto de vista, es el que muestra de una manera más sencilla el funcionamiento general de los algoritmos de optimización, siendo el código siguiente su pseudocódigo:

```
FORD-FULKERSON (G, s, t, c)

FOREACH edge e ∈ E : f(e) ← 0.
    G_f ← residual graph.
    WHILE (there exists an augmenting path P in G_f)
        f ← AUGMENT (f, c, P).
        Update G_f.
    RETURN f.
}
```

Siendo la función AUGMENT(*f, c, P)* representado por el siguiente pseudocódigo:

```
AUGMENT (f, c, P)

b ← bottleneck capacity of path P.
FOREACH edge e ∈ P
    IF (e ∈ E ) f(e) ← f(e) + b.
    ELSE        f(e^R) ← f(e^R) − b.
RETURN f.
```

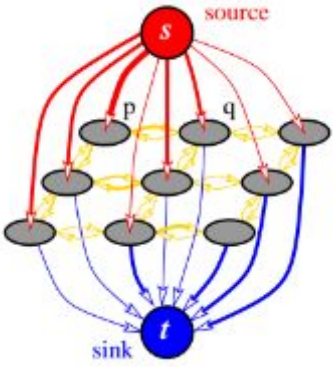
Además, pude observar el coste computacional de los distintos algoritmos y cómo han ido evolucionando a lo largo del tiempo.

year	method	worst case	discovered by
1951	simplex	$O(m^3 C)$	Dantzig
1955	augmenting path	$O(m^2 C)$	Ford-Fulkerson
1970	shortest augmenting path	$O(m^3)$	Dinic, Edmonds-Karp
1970	fattest augmenting path	$O(m^2 \log m \log(mC))$	Dinic, Edmonds-Karp
1977	blocking flow	$O(m^{5/2})$	Cherkasky
1978	blocking flow	$O(m^{7/3})$	Galil
1983	dynamic trees	$O(m^2 \log m)$	Sleator-Tarjan
1985	capacity scaling	$O(m^2 \log C)$	Gabow
1997	length function	$O(m^{3/2} \log m \log C)$	Goldberg-Rao
2012	compact network	$O(m^2 / \log m)$	Orlin
?	?	$O(m)$	?

4. Artículos sobre los algoritmos de optimización

José, mi tutor de prácticas, me facilitó un total de cinco artículos que contienen información sobre los algoritmos de optimización. Durante el periodo del **20/02/2017** al **12/03/2017** dediqué el tiempo a leer dichos artículos y a aprender los conceptos que me resultarían de mayor utilidad a la hora de analizar y entender los algoritmos.

El primer artículo que empecé a leer fue el de *Graph Cuts in Vision and Graphics: Theories and Applications* de Yuri Boykov y Olga Veksler. En él se muestra el funcionamiento teórico del código de Boykov que posteriormente analizamos. Empieza definiendo la estructura que tiene la red a la que le vamos a aplicar la minimización de energía; es una red en forma de árbol, con un nodo fuente y un nodo sumidero. Los nodos pertenecientes a la red se enlazan entre ellos utilizando *n-links* y con el nodo fuente y el nodo sumidero utilizando *t-links*. De tal forma que la red queda representada de la siguiente manera:



Continúa explicando el concepto de corte y el de minimización binaria de la energía, donde ya muestra los conceptos de unary y de labelcost y cómo utilizarlos para computar la energía:

$$E(f) = \sum_{p \in \mathcal{P}} D_p(f_p) \quad + \quad \sum_{(p,q) \in \mathcal{N}} V_{pq}(f_p, f_q)$$

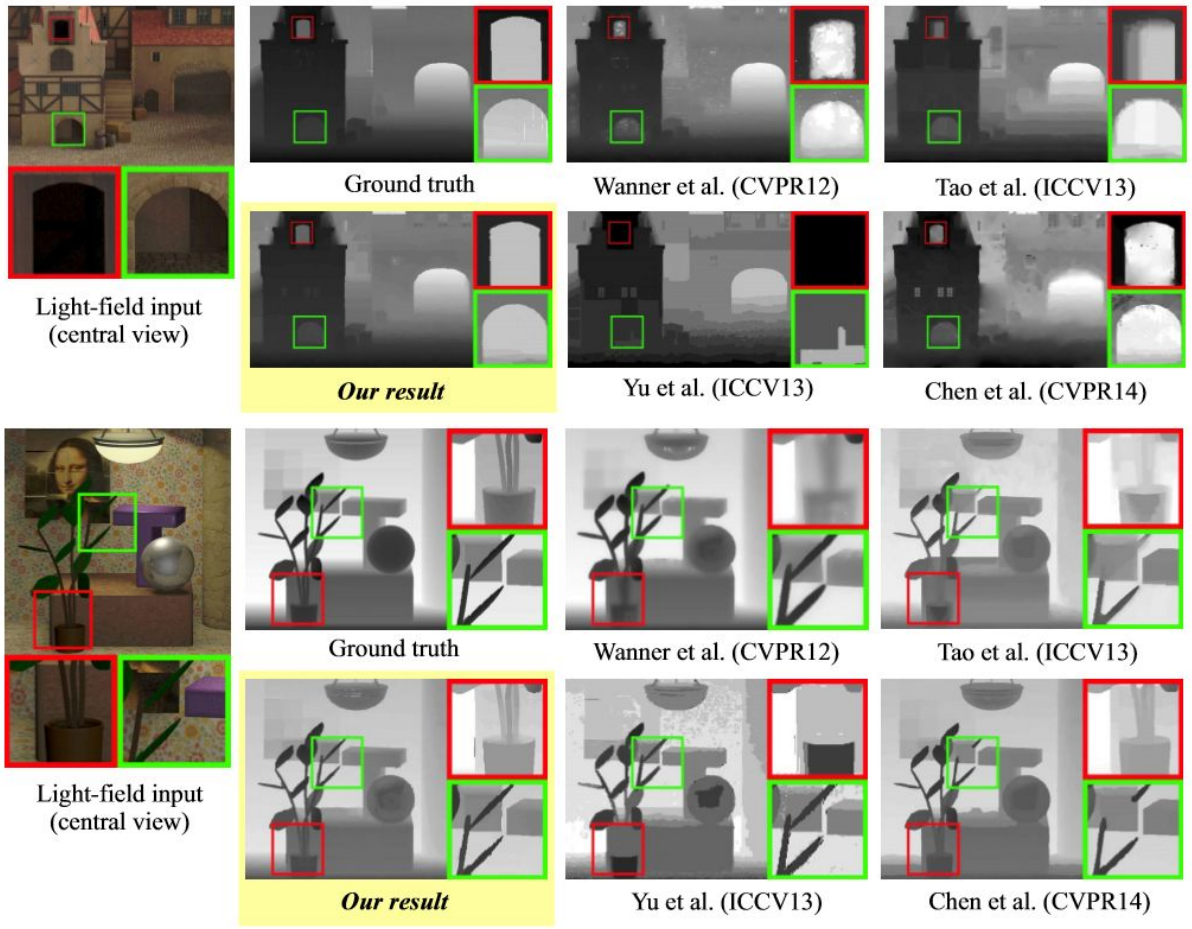
,

siendo D_p el unary y V_{pq} el labelcost.

Termina explicando el concepto de corte en las hipersuperficies.

A continuación, procedí a leer el artículo de Ting-Chun Wang, Alexei A. Efros y Ravi Ramamoorthi cuyo nombre es *Occlusion-aware Depth Estimation Using Light-field Cameras.* En él se utilizan conceptos como campo de luz, oclusiones y mapa de profundidad que explicaré en el siguiente apartado. El objetivo de este artículo es mostrar los resultados conseguidos por el algoritmo creado por los autores, explicando teóricamente las partes de este. Empieza realizando una detección de aristas sobre la imagen, posteriormente realiza un estimación de la profundidad, sigue calculando una constante que define la consistencia

del color y termina realizando la regularización. Todos estos pasos los vemos en mayor detalle en el análisis del algoritmo con MATLAB.
Los resultados de su algoritmo en comparación a otros son los siguientes:



A los artículos *Algorithm Design* de Jon Kleingerb y Éva Tardos y *Network Flow Algorithms* no les dediqué el mismo tiempo que a los dos anteriores ya que como nuestro objetivo es analizar y entender el funcionamiento del código de Boykov, no aportan mucho más de lo que ya sabemos. El artículo *Graph Based Algorithms For Scene Reconstruction From Two Or More Views* de Vladimir Kolmogorov, que desde mi punto de vista es el más completo y el más extenso, lo utilizaré como apoyo para poder explicar las distintas partes del código durante su análisis.

5. Conceptos sobre la imagen

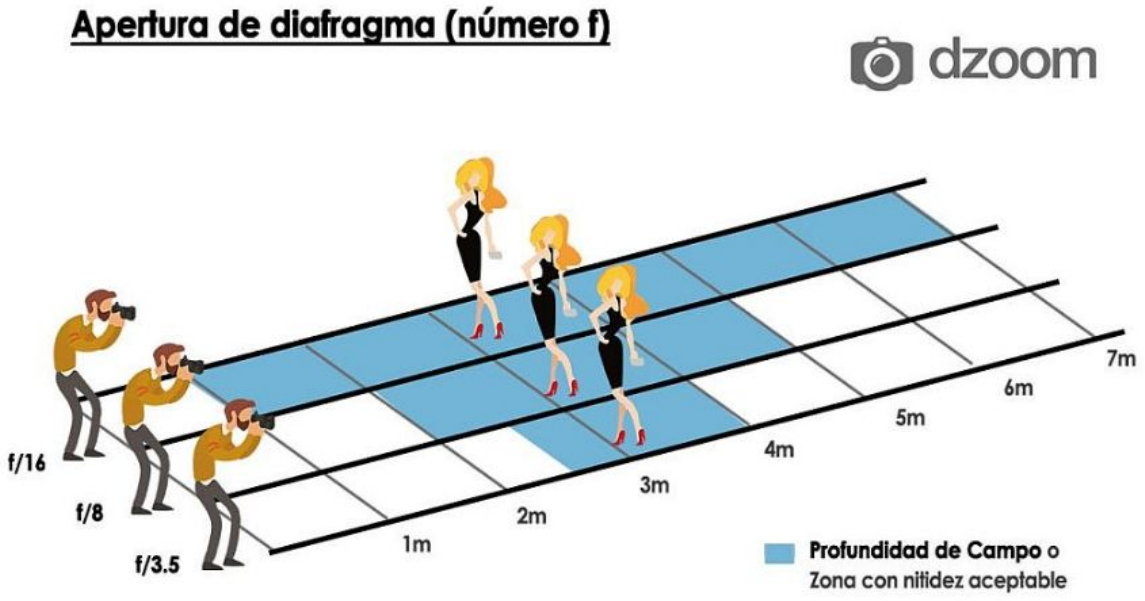
La semana del **13/03/2017** al **19/03/2017** la dediqué a investigar los conceptos de oclusiones, campo de luz y mapa de profundidad mencionados en el apartado anterior.

- Campo de luz: Es una función vectorial que determina la cantidad de luz que fluye en cualquier dirección a través de cualquier punto en el espacio.

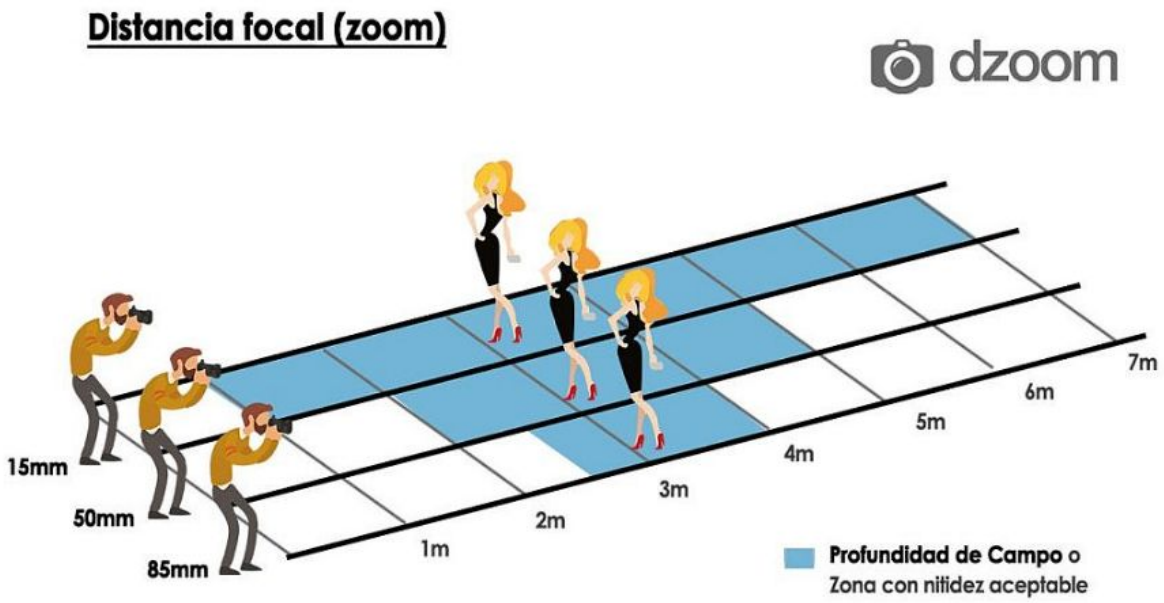
- Oclusiones: Se producen cuando un objeto tapa a otro parcialmente o en su totalidad. Pueden clasificarse en tres categorías: Oclusiones en las que un objeto tapa una porción del área de otro objeto que se quiere reconocer y que no es del todo visible (overlapping), oclusiones por opacidad y oclusiones por sombras.

- Mapa de profundidad: También conocido como profundidad de campo, es la parte de una imagen que se aprecia nítidamente. Para obtener la profundidad de campo deseada es necesario tener en cuenta los siguientes factores:

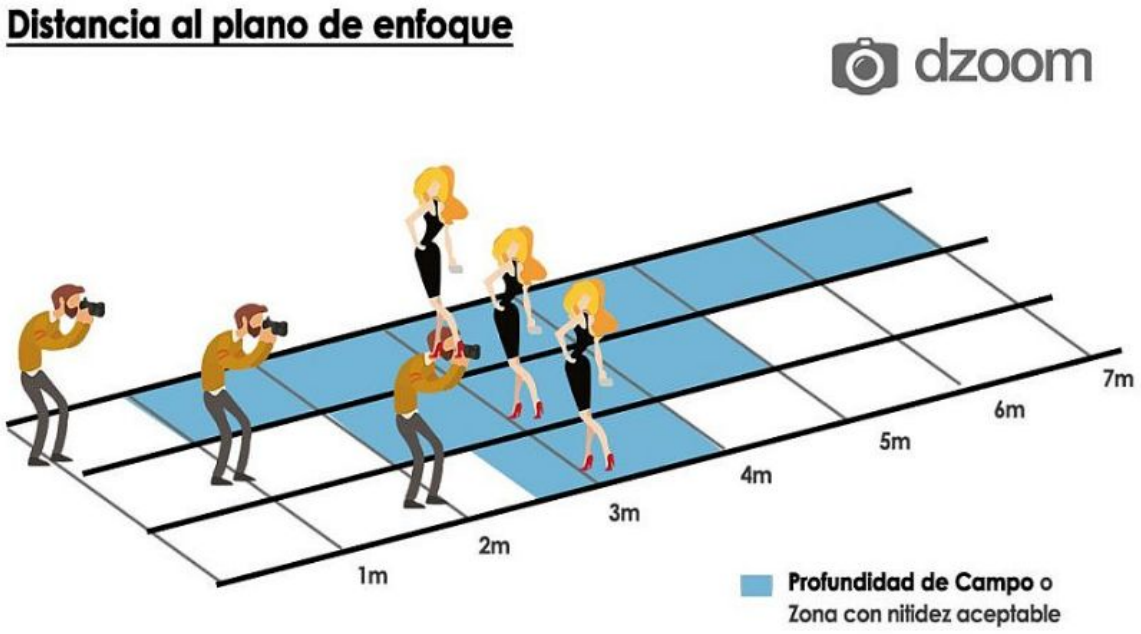
    - Apertura del diafragma: A menor nº f, menor profundidad de campo.



- Zoom: A mayor distancia focal, menor profundidad de campo.



- Plano de enfoque: A menor distancia entre el sujeto y la cámara, menor profundidad de campo.



6. Primeros pasos con MATLAB

La principal característica del MATLAB es que trabaja con matrices, es decir, tiene operadores y funciones implementadas para facilitarnos las operaciones que tengamos que realizar sobre las matrices.

Para definir una matriz es suficiente con la siguiente instrucción: A = [1 2 3; 3 4 5; 6 7 8], de esta forma la matriz A sería una matriz de 3 filas por 3 columnas, donde la fila 1 contendrá a los elementos [1 2 3], la fila 2 a [4 5 6] y la fila 3 a [6 7 8].

Para acceder a cualquier elemento de una matriz utilizaremos A[fila, columna], si queremos hallar una fila A[fila, :] y si queremos hallar una columna A[:, columna].

Para realizar la traspuesta de una matriz utilizaremos A' y para realizar la inversa inv(A).

Tras adquirir el funcionamiento teórico pase a realizar ejercicios sobre matrices, empezando por ejemplos sencillos y terminando en casos concretos sobre matrices empleadas en algoritmos de optimización.

7. Análisis del código de Ting-Chun con MATLAB

Empezaremos analizando el código del fichero demo.m y seguiremos analizando el código de cada fichero que forme parte del algoritmo.

- demo.m

```
%% Input: 5D double array of size [h x w x 3 x v x u]
% [h w 3] is spatial image size, and [v u] is angular size
% read Wanner's data
hinfo_data = hdf5info('input/bedroom.h5');
data = hdf5read(hinfo_data.GroupHierarchy.Datasets(2));
%data = permute(data, [3 2 1 5 4]);
data = im2double(data(:, :, :, :, end:-1:1));
% % read our data
% hinfo_data = hdf5info('input/livingroom.h5');
% data = double(hdf5read(hinfo_data.GroupHierarchy.Datasets(3)));

depth_output = computeDepth(data);
figure; imshow(depth_output);
```

Empieza cargando la imagen "bedroom.h5", extrae el dataset de su plano difuso y convierte los valores de dicho plano en valores de tipo double. A continuación llama a la función computeDepth y muestra el resultado.

- computeDepth.m

```
addpath required
addpath mex
addpath(genpath('regularize/'))
```

```
%% Parameters
d_max = 1;          % maximum disparity between neighboring cameras
d_res = 101;        % depth resolution
v_max = 100;        % maximum correspondence cue response
f_max = 0.01;       % maximum refocus cue response
dilate_amount = 4;  % dilate amount on edge detection
occ_thre = 0.1;     % threshold to be regard as occlusion

% parameters from input
UV_diameter = size(data, 4);                        % angular resolution
UV_radius   = floor(UV_diameter/2);                 % half angular resolution
h           = size(data, 1);                        % spatial image height
w           = size(data, 2);                        % spatial image width
LF_y_size   = h * UV_diameter;                      % total image height
LF_x_size   = w * UV_diameter;                      % total image width
LF_Remap    = reshape(permute(data, [4 1 5 2 3]), [LF_y_size LF_x_size 3]); % the remap image
im          = data(:,:,:,UV_radius+1,UV_radius+1);  % the pinhole image
```

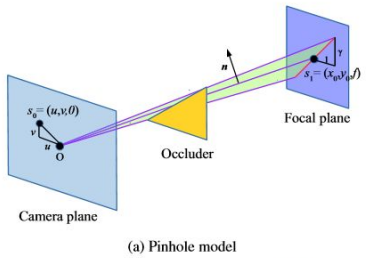
```
%% Edge detection and orientation computation
tic; disp('1. Edge detection')
im_edge = edge(im(:,:,1), 'canny') | edge(im(:,:,2), 'canny') | edge(im(:,:,3), 'canny'); % Aplica el 'canny' a los 3 canales
orient = skeletonOrientation(im_edge, [5 5]);
orient(~im_edge) = -100;
dir = imdilate(orient, strel('disk', dilate_amount));
fprintf('Edge detecion completed in %.2f sec\n', toc);
```

```
%% Initial depth estimation
tic; disp('2. Initial depth estimation')
[depth, d_var, d_cost, d_foc] = ...
    occlusionDetection_mex(w, h, UV_diameter, LF_Remap, -d_max, d_max, d_res, dir*pi/180);
depth = double(depth);
fprintf('Initial depth estimation completed in %.2f sec\n', toc);
```

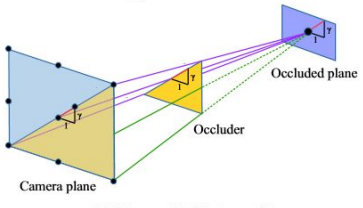
```
%% Occlusion cue computation
tic; disp('3. Occlusion cue computation')
occlusion = occCompute(d_var, d_foc, im_edge, dir, v_max, f_max, occ_thre);
fprintf('Occlusion cue computation completed in %.2f sec\n', toc);
```

```
%% Final depth estimation
tic; disp('4. Final depth estimation')
confidence   = confCompute(d_cost, h, w);
depth_output = DEPTH_MRF2_s(depth, confidence, occlusion, im, h, w, d_res) / d_res;
depth_output = medfilt2(depth_output, [3 3]);
fprintf('Final depth estimation completed in %.2f sec\n', toc);
```

En primer lugar fija los parámetros necesarios para poder realizar la detección de aristas y calcular la estimación inicial de la profundidad y las oclusiones. A continuación calcula los datos necesarios sobre la imagen, así como la imagen que denomina “pinhole” sobre la que realizará la detección de aristas. Esta imagen hace referencia al subapartado *Depth from Light-Field Cameras* del segundo apartado de su artículo:



(a) Pinhole model



(b) “Reversed” pinhole model

En la parte de detección de aristas utiliza el algoritmo “canny” para detectar las aristas en cada uno de los canales de la imagen y posteriormente los dilata para que puedan ser identificados correctamente.

Para realizar la estimación inicial de la profundidad llama a la función occlusionDetection_mex el cual no analizamos ya que está escrito en C# y este no es un lenguaje estudiado en profundidad durante el grado, pero por el artículo sabemos que utiliza la función vectorial del campo de luz para calcularla y posteriormente la convierte a tipo double.

Una vez obtenida la profundidad inicial calcula las oclusiones en base a los parámetros definidos inicialmente y a los obtenidos en la estimación inicial de la profundidad.

Para finalizar, calcula el nivel de confidencia y llama a la función DEPTH_MRF2_s para que regularice la imagen y muestra el resultado.

- DEPTH_MRF2_s.m

Esta función construye el vector CLASS y las matrices UNARY, LABELCOSTS y PAIRWISE y computa el etiquetado final, pero debido a su complejidad decidimos analizar la función DEPTH_MRF2_simple ya que su análisis resulta más sencillo y podemos alcanzar una mayor comprensión.

- DEPTH_MRF2_simple.m

```
% class = 1xN vector of initial estimates
% unary = depthlabels x N of potential terms.(robust norm diff from initial)
% pairwise = sparse NxN link cost, positive for links, confidence & edge
% labelcost = depthlabels x depthlabels , cost of adj. depth diff. robust
% expansion = 1
% tic;
%

min_depth=1;
max_depth=depth_resolution;
confidence = confidence.^0.1;

CLASS = depth(:)';
numnodes = size(CLASS,2);
numdepth= max_depth - min_depth + 1;
UNARY = zeros(numdepth,numnodes);

flat_clip = floor(depth_resolution/2);

for i = min_depth:max_depth
    UNARY(i,:) = min(abs(i - double(CLASS)),flat_clip);
end

LABELCOST = zeros(numdepth,numdepth);
for i = min_depth:max_depth
    for j = min_depth:max_depth
        LABELCOST(i,j) = min(abs(i-j),flat_clip);
    end
end
```

En este trozo de función se construye el vector CLASS, que es el vector 1xN que contiene las etiquetas iniciales, realizando la operación de transponer sobre las filas de la imagen.

La matriz UNARY se construye por filas, aplicando la función que calcula el mínimo entre la resta entre el número de fila actual y todos los elementos del vector CLASS, y depth_resolution/2. Es una matriz CxN que especifica los potenciales (término de datos) para cada una de las C clases posibles en cada uno de los N nodos.

La matriz LABELCOST CxC se construye en base al número máximo de etiquetas, aplicando la función que calcula el mínimo entre la resta entre el número de fila actual y el número de columna, y depth_resolution/2. Dicha matriz representa el coste fijo para cada etiqueta para cada nodo adyacente a la red.

```
h = fspecial('sobel');
image = im2double(image);
vertical = abs(conv2(image,h,'same'));
horizontal = abs(conv2(image,h','same'));

%the confidence is quite crappy now. maybe i shouldn't use it.
numentries = (width-1)*height + (height-1)*width;
numentries = numentries*2 ;% make it sym

%let ii =
ii = zeros(numentries,1);
jj = zeros(numentries,1);
ss = zeros(numentries,1);

con_bias = 1;

count = 1;
for col = 1:width
    for row = 1:height-1
        total_edgestr = vertical(row,col) + vertical(row+1,col) + 0.2;
        local_con = confidence(row,col) + confidence(row+1,col);

        %the more confident it is, the more weight on the edge
        %if it is a strong edge in image, lower it
        local_weight = (local_con + con_bias)/total_edgestr;
        ss(count) = local_weight;
        ii(count) = row + (col-1)*height;
        jj(count) = row + 1 + (col-1)*height;
        count = count + 1;
        ss(count) = local_weight;
        jj(count) = row + (col-1)*height;
        ii(count) = row + 1 + (col-1)*height;
        count = count + 1;
    end
end
```

```
for col = 1:width-1
    for row = 1:height
        total_edgestr = vertical(row,col) + vertical(row,col+1) + 0.2;
        local_con = confidence(row,col) + confidence(row,col+1);

        %the more confident it is, the more weight on the edge
        %if it is a strong edge in image, lower it
        local_weight = (local_con + con_bias)/total_edgestr;

        ss(count) = local_weight;
        ii(count) = row + (col-1)*height;
        jj(count) = row + (col)*height;
        count = count + 1;
        ss(count) = local_weight;
        jj(count) = row + (col-1)*height;
        ii(count) = row + (col)*height;
        count = count + 1;
    end
end
PAIRWISE = sparse(ii,jj,ss);

labels    = GCMex(CLASS, single(UNARY), PAIRWISE, single(LABELCOST),1)
depth_output        = reshape(labels,height,width)
```

Esta parte del código simplemente convierte la imagen en una red que es representada por la matriz NxN PAIRWISE.

Para finalizar llama a la función GCMex y llegados a este punto debemos dar por finalizado nuestro análisis del código de Ting-Chun ya que no es posible analizar la función GCMex desde MATLAB. Por este motivo, en el siguiente apartado intentaremos analizar el código de Boykov con MATLAB.

8. Análisis del código de Boykov con MATLAB

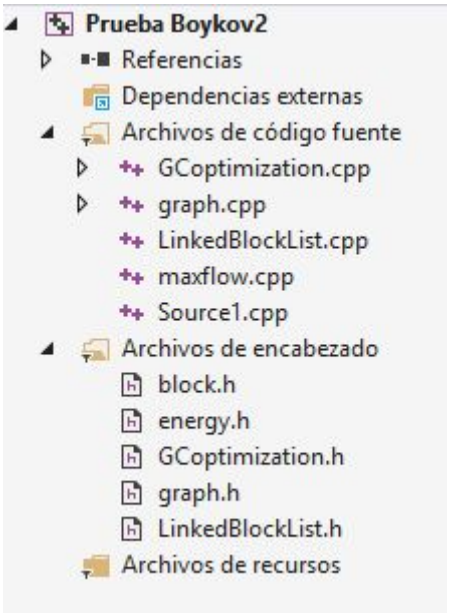
El primer paso fue buscar el algoritmo de Boykov preparado para MATLAB, una vez encontrado procedí a analizarlo y me encontré con el mismo problema que tuve en el apartado anterior, una vez construidas las matrices UNARY, LABELCOST y PAIRWISE se realiza una llamada a una función que está contenida en un archivo de tipo mex y no me es posible seguir con el análisis.

El siguiente paso fue buscar el algoritmo de Boykov en algún lenguaje de programación que hubiese dado durante el grado, es decir, python o java. Empecé buscandolo en java y no encontré ningún resultado que me sirviera. Posteriormente, lo busqué en python y obtuve lo que estaba buscando. Procedí a instalar todas las dependencias necesarias (python incluido) y empecé a analizar el código. Para mi desilusión me volví a tropezar con la misma piedra, tenía que analizar funciones escritas en C++.

Como no encontramos ninguna solución posible, decidimos que debía aprender a utilizar el Visual Studio y con él, poco a poco, ir analizando el código en C++.

9. Primeros pasos con Visual Studio y C++

Con la ayuda de José y de algunos tutoriales de youtube, conseguí aprender a crear un nuevo proyecto en consola y agregarle los archivos necesarios para su funcionamiento. Lo nombré “Prueba Boykov2” y esta es su estructura:



Para ir practicando con C++ fui implementando poco a poco el archivo Source1.cpp que es el que se encarga de llamar a una función para que cree la red, preparar las matrices UNARY y LABELCOST y imprimir la energía por consola. Una vez depurados todos los errores, esta fue la salida:

```
Before optimization energy is 250
After optimization energy is 44
Before optimization energy is 250
After optimization energy is 44
Before optimization energy is 250
After optimization energy is 44
Before optimization energy is 250
After optimization energy is 170
Before optimization energy is 250
After optimization energy is 44
Before optimization energy is 250
After optimization energy is 244
Finished 2 (2571) clock per sec 1000
```

10. Análisis del código de Boykov con Visual Studio

Para empezar consideremos las siguientes condiciones iniciales:

```
//  Optimization problem:
//  is a set of sites (pixels) of width 10 and hight 5. Thus number of pixels is 50
//  grid neighborhood: each pixel has its left, right, up, and bottom pixels as neighbors
//  7 labels
//  Data costs: D(pixel,label) = 0 if pixel < 25 and label = 0
//            : D(pixel,label) = 10 if pixel < 25 and label is not  0
//            : D(pixel,label) = 0 if pixel >= 25 and label = 5
//            : D(pixel,label) = 10 if pixel >= 25 and label is not  5
//
//  Data costs (n labels): D(pixel, label) = min(|label-pixel|, const)
//
// Smoothness costs: V(p1,p2,l1,l2) = min( (l1-l2)*(l1-l2) , 4 )
// Below in the main program, we illustrate different ways of setting data and smoothness costs
// that our interface allow and solve this optimizaiton problem

// For most of the examples, we use no spatially varying pixel dependent terms.
// For some examples, to demonstrate spatially varying terms we use
// V(p1,p2,l1,l2) = w_{p1,p2}*[min((l1-l2)*(l1-l2),4)], with
// w_{p1,p2} = p1+p2 if |p1-p2| == 1 and w_{p1,p2} = p1*p2 if |p1-p2| is not 1
```

Esta imagen anterior nos especifica cómo se va a construir la matriz Data costs, que nosotros conocemos como UNARY y la matriz Smoothness costs, la cual nosotros conocemos como LABELCOST.

A continuación, vamos a ver las funciones principales que difieren entre ellas por la forma de almacenar y enviar los datos de las matrices anteriormente descritas y por la forma de crear la red.

```
// smoothness and data costs are set up one by one, individually
GridGraph_Individually(width, height, num_pixels, num_labels);
// smoothness and data costs are set up using arrays
GridGraph_DArraySArray(width, height, num_pixels, num_labels);
// smoothness and data costs are set up using functions
GridGraph_DfnSfn(width, height, num_pixels, num_labels);
// smoothness and data costs are set up using arrays.
// spatially varying terms are present
GridGraph_DArraySArraySpatVarying(width, height, num_pixels, num_labels);
//Will pretend our graph is
//general, and set up a neighborhood system
// which actually is a grid
GeneralGraph_DArraySArray(width, height, num_pixels, num_labels);
//Will pretend our graph is general, and set up a neighborhood system
// which actually is a grid. Also uses spatially varying terms
GeneralGraph_DArraySArraySpatVarying(width, height, num_pixels, num_labels);
```

Mi objeto de análisis fue la primera función, ya que mi objetivo es analizar el funcionamiento del algoritmo y eligiendo la primera función me facilita su comprensión.

Los datos iniciales que se les pasa a las funciones son los siguientes:

```
int width = 10;
int height = 5;
int num_pixels = width*height;
int num_labels = 7;
```

Y la función GridGraph_Individually a analizar es:

```
void GridGraph_Individually(int width, int height, int num_pixels, int num_labels)
{

    int *result = new int[num_pixels];   // stores result of optimization

    try {
        GCoptimizationGridGraph *gc = new GCoptimizationGridGraph(width, height, num_labels);

        // first set up data costs individually
        for (int i = 0; i < num_pixels; i++)
            for (int l = 0; l < num_labels; l++)
                if (i < 25) {
                    if (l == 0) gc->setDataCost(i, l, 0);
                    else gc->setDataCost(i, l, 10);
                }
                else {
                    if (l == 5) gc->setDataCost(i, l, 0);
                    else gc->setDataCost(i, l, 10);
                }

                // next set up smoothness costs individually
                for (int l1 = 0; l1 < num_labels; l1++)
                    for (int l2 = 0; l2 < num_labels; l2++) {
                        int cost = (l1 - l2)*(l1 - l2) <= 4 ? (l1 - l2)*(l1 - l2) : 4;
                        gc->setSmoothCost(l1, l2, cost);
                    }

                printf("\nBefore optimization energy is %d", gc->compute_energy());
                gc->expansion(2);// run expansion for 2 iterations. For swap use gc->swap(num_iterations);
                printf("\nAfter optimization energy is %d", gc->compute_energy());

                for (int i = 0; i < num_pixels; i++) {
                    result[i] = gc->whatLabel(i);
                }

                delete gc;
    }
    catch (GCException e) {
        e.Report();
    }

    delete[] result;
}
```

Como podemos observar lo primero que realiza es construir una red mediante la función GCoptimizationGridGraph que además se encarga de inicializar las variables necesarias para la minimización de la energía. La red consta de 4 nodos vecinos por cada nodo.

A continuación aplica las condiciones definidas anteriormente para la matriz UNARY y va enviando elemento por elemento toda la matriz mediante la función setDataCost():

```
void GCoptimization::setDataCost(SiteID s, LabelID l, EnergyTermType e) {
    if (!m_datacostIndividual) {
        if ( m_datacostFn ) handleError("Data Costs are already initialized");
        m_datacostIndividual = new EnergyTermType[m_num_sites*m_num_labels];
        memset(m_datacostIndividual, 0, m_num_sites*m_num_labels*sizeof(EnergyTermType));
        specializeDataCostFunctor(DataCostFnFromArray(m_datacostIndividual, m_num_labels));
    }
    m_datacostIndividual[s*m_num_labels + l] = e;
}
```

Esta función se encarga de inicializar el vector m_datacostIndividual y si ya está inicializado almacena el elemento que recibe como argumento.

De igual manera ocurre con la matriz LABELCOST, mediante la función setSmoothCost envía elemento por elemento toda la matriz:

```
void GCoptimization::setSmoothCost(LabelID l1, GCoptimization::LabelID l2, EnergyTermType e){
    if (!m_smoothcostIndividual) {
        if ( m_smoothcostFn ) handleError("Smoothness Costs are already initialized");
        m_smoothcostIndividual = new EnergyTermType[m_num_labels*m_num_labels];
        memset(m_smoothcostIndividual, 0, m_num_labels*m_num_labels*sizeof(EnergyTermType));
        specializeSmoothCostFunctor(SmoothCostFnFromArray(m_smoothcostIndividual, m_num_labels));
    }
    m_smoothcostIndividual[l1*m_num_labels + l2] = e;
}
```

De esta forma tenemos dos vectores que contienen a todos los elementos de nuestras matrices UNARY y LABELCOST, respectivamente.

A continuación, se computa la energía mediante la función compute_energy():

```
GCoptimization::EnergyType GCoptimization::compute_energy()
{
    if (readyToOptimise())
        return( (this->*m_giveDataEnergyInternal)()+ (this->*m_giveSmoothEnergyInternal)());
    else{
        handleError("Not ready to optimize yet. Set up data and smooth costs first");
        return(0);
    }
}
```

Donde podemos comprobar que está implementada la ecuación descrita en el punto número 4 de este informe.

Por un lado tenemos la función giveDataEnergyInternal(), que se encarga de calcular la parte de la energía del UNARY dependiendo del etiquetado actual (m_labeling) que está inicializado a 0 y de la función compute() que fue definida cuando se inicializó el vector m_datacostIndividual.

```
template <typename DataCostT>
GCoptimization::EnergyType GCoptimization::giveDataEnergyInternal()
{

    EnergyType eng = (EnergyType) 0;

    DataCostT* tc = (DataCostT*)m_datacostFn;

    for ( SiteID i = 0; i < m_num_sites; i++ )
    {
        assert(m_labeling[i] >= 0 && m_labeling[i] < m_num_labels);
        eng = eng + tc->compute(i,m_labeling[i]);
    }

    return(eng);
}
```

```
struct DataCostFnFromArray {
    DataCostFnFromArray(EnergyTermType* theArray, LabelID num_labels)
        : m_array(theArray), m_num_labels(num_labels){}
    OLGA_INLINE EnergyTermType compute(SiteID s, LabelID l){return m_array[s*m_num_labels+l];}
private:
    const EnergyTermType* const m_array;
    const LabelID m_num_labels;
};
```

A continuación mostramos el cómputo de la parte de la energía del LABELCOST:

```
template <typename SmoothCostT>
GCoptimization::EnergyType GCoptimization::giveSmoothEnergyInternal()
{

    EnergyType eng = (EnergyType) 0;
    SiteID i,numN,*nPointer,nSite,n;
    EnergyTermType *weights;

    SmoothCostT* sc = (SmoothCostT*) m_smoothcostFn;

    for ( i = 0; i < m_num_sites; i++ )
    {
        giveNeighborInfo(i,&numN,&nPointer,&weights);

        for ( n = 0; n < numN; n++ )
        {
            nSite = nPointer[n];
            if ( nSite < i ) eng =
                        eng + weights[n]*(sc->compute(i,nSite,m_labeling[i],m_labeling[nSite]));
        }
    }

    return(eng);
}
```

Inicialmente el vector weights está inicializado a 1 por lo que no influye en el cómputo. Como podemos observar, en primer lugar halla los nodos vecinos del nodo actual y va acumulando los valores que devuelve la función compute(), definida el inicializar el vector m_smoothcostIndividual:

```
struct SmoothCostFnFromArray {
    SmoothCostFnFromArray(EnergyTermType* theArray, LabelID num_labels)
        : m_array(theArray), m_num_labels(num_labels){}
    OLGA_INLINE EnergyTermType compute(SiteID s1, SiteID s2, LabelID l1, LabelID l2){return m_array[l1*m_num_labels+l2];}
private:
    const EnergyTermType* const m_array;
    const LabelID m_num_labels;
};
```

A continuación vamos a analizar como se realiza la minimización de la energía.
La función que inicia dicha minimización es expansion():

```
GCoptimization::EnergyType GCoptimization::expansion(int max_num_iterations )
{
    int curr_cycle = 1;
    EnergyType new_energy,old_energy;


    new_energy = compute_energy();

    old_energy = new_energy+1;

    while ( old_energy > new_energy  && curr_cycle <= max_num_iterations)
    {
        old_energy = new_energy;
        new_energy = oneExpansionIteration();
        curr_cycle++;
    }

    return(new_energy);
}
```

Donde para cada iteración o en el caso de que la nueva energía sea superior a la vieja, llama a la función oneExpansionIteration():

```
GCoptimization::EnergyType GCoptimization::oneExpansionIteration()
{
    LabelID next;

    if (m_random_label_order) scramble_label_table();

    for (next = 0;  next < m_num_labels;  next++ )
    {
        alpha_expansion(m_labelTable[next]);
    }

    return(compute_energy());
}
```

Para comprender el funcionamiento de esta función debemos tener en cuenta dos cosas. La primera es que m_labelTable fue inicializada con las etiquetas ordenadas de menor a mayor cuando se creó la red y la segunda es que llamará a la función alpha_expansión() una vez por cada etiqueta y devolverà la nueva energía computada:

```
// alpha expansion on all sites not currently labeled alpha
void GCoptimization::alpha_expansion(LabelID alpha_label)
{
    SiteID i  = 0, size = 0;
    assert( alpha_label >= 0 && alpha_label < m_num_labels);

    SiteID *activeSites = new SiteID[m_num_sites];

    for ( i = 0; i < m_num_sites; i++ )
    {
        if ( m_labeling[i] != alpha_label )
        {
            activeSites[size] = i;
            m_lookupSiteVar[i] = size;
            size++;
        }
    }

    solveExpansion(size,activeSites,alpha_label);

    delete [] activeSites;
}
```

Esta función identifica los nodos que no están etiquetados con la etiqueta que se recibe como argumento y almacena su orden. Posteriormente, realiza la expansión para dicha etiqueta mediante la función solveExpansión() que mostraré en la siguiente página.

```
// Sets up the energy and optimizes it. Also updates labels of sites, if needed. ActiveSites
// are the sites participating in expansion. They are not labeled alpha currrently
void GCoptimization::solveExpansion(SiteID size,SiteID *activeSites,LabelID alpha_label)
{
    SiteID i,site;

    if ( !readyToOptimise() ) handleError("Set up data and smoothness terms first. ");
    if ( size == 0 ) return;


    Energy *e = new Energy(error_function);

    Energy::Var *variables = (Energy::Var *) new Energy::Var[size];

    for ( i = 0; i < size; i++ )
        variables[i] = e ->add_variable();

    set_up_expansion_energy(size,alpha_label,e,variables,activeSites);

    Energy::TotalValue Emin = e -> minimize();

    for ( i = 0; i < size; i++ )
    {
        site = activeSites[i];
        if ( m_labeling[site] != alpha_label )
        {
            if ( e->get_var(variables[i]) == 0 )
            {
                m_labeling[site] = alpha_label;
            }
        }
        m_lookupSiteVar[site] = -1;
    }

    delete [] variables;
    delete e;
}
```

En esta función es donde se encuentra la parte más importante del algoritmo.

En primer lugar, almacena tantos nodos como el argumento “size” indique, en el array variables[]. A continuación, vamos a analizar la función set_up_expansion_energy():

```
void GCoptimization::set_up_expansion_energy(SiteID size, LabelID alpha_label,Energy *e,
                                             VarID *variables, SiteID *activeSites )
{


    (this->*m_set_up_t_links_expansion)(size,alpha_label,e,variables,activeSites);
    (this->*m_set_up_n_links_expansion)(size,alpha_label,e,variables,activeSites);
}
```

Está función se encarga de crear y dar valores a los t_links y a los n_links.

La función encargada de gestionar los t_links es:

```
template <typename DataCostT>
void GCoptimization::set_up_t_links_expansion(SiteID size, LabelID alpha_label,Energy *e,
                                              VarID *variables,SiteID *activeSites )
{
    DataCostT* dc = (DataCostT*)m_datacostFn;

    for ( SiteID i = 0; i < size; i++ )
    {
        e -> add_term1(variables[i],dc->compute(activeSites[i],alpha_label),
                                    dc->compute(activeSites[i],m_labeling[activeSites[i]]));
    }
}
```

Donde la función add_term1() es:

```
inline void Energy::add_term1(Var x,
                              Value A, Value B)
{
    add_tweights(x, B, A);
}
```

Y la función add_tweights():

```
void Graph::add_tweights(node_id i, captype cap_source, captype cap_sink)
{
    register captype delta = ((node*)i) -> tr_cap;
    if (delta > 0) cap_source += delta;
    else           cap_sink   -= delta;
    flow += (cap_source < cap_sink) ? cap_source : cap_sink;
    ((node*)i) -> tr_cap = cap_source - cap_sink;
}
```

Esencialmente modifica el peso del enlace de todos los nodos que participan en la expansión dependiendo de su capacidad residual. Además calcula el flujo dependiendo de los argumentos que le facilitemos como argumentos.

La función que se encarga de los n_links es:

```
template <typename SmoothCostT>
void GCoptimization::set_up_n_links_expansion(SiteID size, LabelID alpha_label,Energy *e,
                                              VarID *variables,SiteID *activeSites )
{
    SiteID i,nSite,site,n,nNum,*nPointer;
    EnergyTermType *weights;

    SmoothCostT* sc = (SmoothCostT*)m_smoothcostFn;

    for ( i = size - 1; i >= 0; i-- )
    {
        site = activeSites[i];

        giveNeighborInfo(site,&nNum,&nPointer,&weights);
        for ( n = 0; n < nNum; n++ )
        {
            nSite = nPointer[n];


            if ( nSite < site )
            {
                if ( m_lookupSiteVar[nSite] != -1 )
                    e ->add_term2(variables[i],variables[m_lookupSiteVar[nSite]],
                    sc->compute(site,nSite,alpha_label,alpha_label)*weights[n],
                    sc->compute(site,nSite,alpha_label,m_labeling[nSite])*weights[n],
                    sc->compute(site,nSite,m_labeling[site],alpha_label)*weights[n],
                    sc->compute(site,nSite,m_labeling[site],m_labeling[nSite])*weights[n]);
                else   e ->add_term1(variables[i],sc->compute(site,nSite,alpha_label,m_labeling[nSite])*weights[n],
                                sc->compute(site,nSite,m_labeling[site],m_labeling[nSite])*weights[n]);
            }
            else
            {
                if (m_lookupSiteVar[nSite] == -1 )
                   e ->add_term1(variables[i],sc->compute(site,nSite,alpha_label,m_labeling[nSite])*weights[n],
                                              sc->compute(site,nSite,m_labeling[site],m_labeling[nSite])*weights[n]);
            }
        }
    }
}
```

Esta función crea y da valores a las aristas que enlazan los nodos, si el nodo no participa en la expansión, se modifica su t_link.

Llegados a este punto, únicamente nos queda minimizar la energía, que básicamente es calcular el flujo máximo, mediante la función minimize() y añadir la etiqueta a aquellos nodos que tengan padre y no estén conectados con el sumidero.

Con el nuevo etiquetado volvemos a computar la energía y finalizamos nuestro análisis.

11. Instalación de OpenCV en el Visual Studio

Enlaces:

- Descarga de OpenCV: http://opencv.org/releases.html
- Instalación: https://mecatronicauaslp.wordpress.com/2013/07/31/instalar-opencv-2-4-6-en-windows-78-x86x64-y-visual-studio-20102012/
- Configuración de un nuevo proyecto: https://mecatronicauaslp.wordpress.com/2013/07/31/configurar-un-proyecto-de-opencv-2-4-6-en-visual-studio-2010-x86x64/
- Vídeo con todos los pasos: https://www.youtube.com/watch?v=gXzshL7ZxeI

12. Algoritmo de Boykov con una imagen

Una vez instalado OpenCV en el Visual Studio, únicamente tenemos que añadir las siguientes líneas a nuestro archivo Source1.cpp:

```
#include "opencv2/opencv.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/imgproc.hpp"
```

```
cv::Mat image;
image = cv::imread("images/Camera0025.bmp", 1);
cv::imshow("Camera normal", image);
```

Con ello ya podríamos aplicar el algoritmo de Boykov a una imagen.

13. Conclusiones

La visión por computadora era un campo que desconocía y gracias a esta estancia lo he podido conocer y puedo dar buenas opiniones sobre él. He tenido que aprender muchos conceptos y dedicarle horas a analizar código, pero el resultado ha sido gratificante. Ahora se como funcionan los algoritmos de optimización computacionalmente y también conozco bastantes aplicaciones de estos. Los resultados que ofrece el algoritmo de Ting-Chun son bastante buenos para cierto tipo de imágenes, al igual que el de Boykov, por ello el objetivo era analizar dichos códigos para ver en qué influye el tipo de imagen. Y la respuesta es, que son dos algoritmos a los que les cuesta adaptarse a los cambios, por ejemplo el de Ting-Chun si le pasas una imagen de un tipo que no venga de una cámara de campo de luz, falla, porque al preparar los datos se basa en la estructura que tiene dicha imagen. El de Boykov, por defecto no viene con ninguna imagen como entrada, simplemente define la anchura, la altura y el número de etiquetas, y a no ser que instales el OpenCV no hay posibilidad de trabajar con imágenes. Termino la estancia con la esperanza de que a José le sea de utilidad el trabajo realizado.