Estudio y comparación de algoritmos para el problema de flujo de coste mínimo

Diego Coto Fernández

Resumen

En este trabajo presentaré las conceptos de la teoría de grafos necesarias para plantear el problema de flujo de coste mínimo, un problema de flujo en redes de gran interés práctico. Estudiaré de forma teórica varios algoritmos para resolver este problema y los compararé en la práctica aplicándolos a los grafos diseñados por Klingman et al en el generador aleatorio NETGEN.

Índice

1. Introducción

1.1. Introducción (placeholder)

El problema del flujo de coste mínimo ha sido llamado el modelo más fundamental de las redes de flujo. Las aplicaciones son innumerables (y hablaré de ellas), pero intuitivamente no es un problema difícil de entender si pensamos en una aplicación sencilla como es el transporte de un cierto producto. Tenemos un conjunto de fábricas desde las que queremos transportar una cierta cantidad de un producto a un conjunto de almacenes. Queremos escoger de entre las rutas posibles aquellas que nos resultan más baratas para satisfacer la demanda, minimizando así el coste del transporte. Hay que decidir, por tanto, qué cantidad del producto crear en cada fábrica, qué fábrica sumistra a qué almacén, y qué itinerario se sigue para llegar a él.

Este es un problema conceptualmente sencillo, pero encontrar la solución óptima cuando el número de fábricas, almacenes y rutas es muy grande requiere el uso de algoritmos eficientes para llegar a la solución en un tiempo razonable. La mayor parte de los avances se realizaron en la segunda mitad del siglo XX, entre la década de los cuarenta y la de los noventa. Durante este tiempo se crearon multitud de algoritmos. Sin embargo, la principal forma de comparar algoritmos es fijándose en el peor escenario teóricamente posible, y esto no siempre nos da una idea acertada de qué algoritmo se comporta mejor en los problemas reales, donde el peor escenario puede no aparecer nunca.

En la primera sección de este trabajo empezaré definiendo los conceptos fundamentales de la teoría de grafos, formalizando el concepto de grafo, red, y estableciendo la terminología y notación que utilizaré en adelante. Continuaré con la presentación de problemas básicos de la teoría de grafos que son necesarios para el estudio del problema de flujo de coste mínimo. En particular, hablaré de cómo encontrar caminos de coste mínimo, algo que se utilizará en varios algoritmos, y cómo encontrar el flujo máximo, que puede considerarse como un problema previo al de coste mínimo. También definiré formalmente el problema que nos ocupa.

Finalmente, hablaré de cómo voy a comparar los algoritmos en la teoría y en la práctica y la metodología que utilizaré para el estudio. Esta sección me ha requerido consultar una amplia bibliografía con el fin de seleccionar los conceptos que utilizaré y unificar terminología y notaciones. La falta de bibliografía en español ha sido problemática, haciendo que algunos de los nombres utilizados sean una traducción propia. Es posible que en otros libros y artículos la terminología cambie.

En la segunda parte explicaré todos los conceptos y propiedades necesarias para definir e implementar los algoritmos que utilizaré en mi estudio. El comportamiento teórico de estos algoritmos irá mejorando a medida que avanzo. Veremos dos ejemplos de algoritmos pseudo-polinomiales: el algoritmo de sucesivos caminos más cortos, en el que se basarán algoritmos posteriores, y el algoritmo de relajación, un algoritmo más moderno que a pesar de no ser de los mejores teóricamente da muy buenos resultados en la práctica. Trataremos después un algoritmo débilmente polinomial: el algoritmo de ajuste de capacidad, que puede considerarse como una versión mejorada del algoritmo de sucesivos caminos más cortos. Acabaremos viendo un algoritmo fuertemente polinomial: el algoritmo de ajuste repetido de capacidad, la continuación lógica del algoritmo de ajuste de capacidad. Esta sección me ha requerido estudiar bastantes más algoritmos de los presentados en este trabajo con el fin de seleccionar aquellos que mejor se adecúan a éste.

En la tercera y última parte del trabajo haré el estudio comparativo práctico de los algoritmos. Usaré el generador de algoritmos aleatorios NETGEN con los parámetros propuestos por sus creadores para crear cincuenta redes de flujo. A cada redes le aplicaré mi implementación en octave de los algoritmos estudiados en la sección anterior y anotaré el tiempo empleado por la CPU en cada uno de ellos. Con estos datos analizaré en qué casos es superior cada algoritmo y cómo evoluciona el tiempo de ejecución de éstos al aumentar el número de nodos, ejes y la capacidad y coste máximos. El trabajo terminará con una exposición de las conclusiones que podemos sacar de este análisis. La implementación de los algoritmos me ha requerido mejorar mucho en algoritmia y programación. Durante el grado habíamos implementado algoritmos en octave, pero nunca a esta escala. Además,

algunos pasos requerían estructuras de datos que no existen por defecto en octave, lo que me ha llevado a aprender cosas nuevas de este lenguaje. Finalmente, el análisis comparativo ha requerido el uso de una amplia gama de herramientas estadísticas adquiridas a lo largo de los últimos años.

Además de lo que puede verse en esta memoria, existe también un anexo en el que aparecen todos los detalles de la implementación de los algoritmos, incluyendo el código de todos los programas y funciones usados. También aparecerá un análisis más detallado de los grafos generados por el programa NETGEN, que no afecta demasiado a las conclusiones del trabajo pero que creo que hay que incluir por completitud. INSERTAR AQUÍ OTRAS COSAS QUE NO ME COJAN EN EL TRABAJO.

En general, en el proceso de crear este trabajo he tenido que aplicar numerosos conceptos dados en el grado junto con otros nuevos que he adquirido a través de libros y artículos. El estudio previo ha requerido manejar una amplia bibliografía y gran capacidad de comprensión, análisis, selección y síntesis. El aspecto de programación ha necesitado la adquisición de numerosas herramientas nuevas, desde manejar scripts en fortran o bash hasta crear clases en octave. También he tenido que mejorar en mi habilidad para crear programas elegantes, legibles y, sobre todo, eficientes, y usar y combinar técnicas de varias ramas de las matemáticas, la estadística y la investigación operativa adquiridas durante los últimos años.

1.2. Bases de la teoría de grafos

HISTORIA APLICACIONES

Empezaré dando los conceptos básicos de la teoría de grafos, que me permitirán más adelante presentar el problema sobre el que trata este trabajo. Comenzaré explicando qué entendemos por grafo y sus elementos.

De forma intuitiva, un grafo es una forma de presentar ciertas relaciones entre objetos. Formalmente, un **grafo** G es un par de conjuntos finitos (V, E). A los elementos de V los llamamos **nodos** o vértices y a los elementos de E, **ejes** o aris-

tas. Un eje es un par $\{i, j\},\ i, j \in V$ que representa una relación entre los nodos i y j. Se dice que un eje ünedos nodos. A estos dos nodos se les llama **extremos** del eje. Denotaremos al número de nodos del grafo por n, y al número de ejes por m.

Un **subgrafo** de un grafo $G = (V, E)$ es un grafo $G' = (V', E')$ tal que $V' \subseteq V$ y $E' \subseteq E$, con $i, j \in V' \forall \{i, j\} \in E'$.

Gráficamente, podemos representar el conjunto de nodos mediante puntos en el espacio y los ejes mediante líneas o flechas que unen los nodos que tienen por extremos. La figura 1 muestra un ejemplo de un grafo.

Un grafo se dice no dirigido cuando no importa el orden de los extremos de sus ejes. Así, si invertimos ese orden obtenemos el mismo eje, $e = \{i, j\} = \{j, i\}$. Si nos importa el orden, $e_1 = \{i, j\} \neq \{j, i\} = e_2$, diremos que se trata de un grafo **dirigido**. En este caso, dado un eje $e = \{i, j\}$, diremos que i es el **nodo inicial** y j el **nodo final**. A los nodos de los que salen ejes que llegan al nodo i los llamaremos **ascendentes** de i. Al conjunto de ascendentes de i lo denotaremos $A(i)$. Al los nodos a los que llegan ejes que salen del nodo i los llamaremos **descendientes** de i. Al conjunto de descendientes de i lo denotaremos $D(i)$.

Un grafo no dirigido puede transformarse en un grafo dirigido duplicando el número de ejes. De este modo, si $G_1 = (V, E_1)$ es un grafo no dirigido, podemos definir un grafo dirigido equivalente tomando el mismo conjunto de nodos V y haciendo que para cada $e = \{i, j\} \in E_1,\ e_1 = \{i, j\},\ e_2 = \{j, i\} \in E_2$, obteniendo un grafo con el mismo número de nodos y el doble de ejes. Sin embargo, en general no podemos transformar un grafo dirigido en uno no dirigido sin modificar las propiedades del grafo, por lo que podemos considerar el grafo dirigido como el caso más general. Éste es el principal motivo por el que en este trabajo utilizaré únicamente grafos dirigidos. Todo lo que podemos hacer para poder tratar un grafo dirigido como uno no dirigido es considerar todos sus ejes como no dirigidos. Esto será útil en algunos casos. Al grafo que obtenemos de esta manera se le llama **grafo subyacente**.

Gráficamente, en un grafo no dirigido los ejes los representamos mediante líneas. El grafo representado en la figura 1 es un grafo no dirigido. Si el grafo es dirigido, representamos los ejes mediante flechas que van desde el nodo inicial al nodo final. La figura 2 muestra un grafo dirigido.

Si un eje une un nodo consigo mismo, $e = \{i, i\}, i \in V$, decimos que ese eje es un **bucle**. Además, en general, dos nodos $i, j \in V$ pueden estar unidos por p ejes iguales $e_1 = e_2 = \cdots = e_p = \{i, j\}$. Un grafo $G = (V, E)$ se llama p-grafo si no hay ningún par de nodos en V unidos por más de p ejes iguales. Así, a un grafo que no contiene ningún par de nodos unidos por más de un eje se le llama 1-grafo. A los 1-grafos que no contienen ningún bucle se les llama **grafos simples**. En este trabajo utilizaré principalmente grafos simples. Esto no supone ninguna limitación para los problemas que nos preocupan. Si pensamos en el ejemplo del transporte vemos inmediatamente que no tiene sentido transportar un producto al lugar donde ya se encuentra mediante un bucle, y cada ruta de transporte entre dos puntos está representado por un único eje.

El grafo representado en la figura 2 es un grafo simple.

En un grafo dirigido simple, $m \leq n(n-1)$, ya que si hay n nodos, cada nodo i puede estar unido con a lo sumo $n-1$ nodos (por no poder estar unido consigo mismo) y ninguno de esos nodos puede estar repetido (por ser un 1-grafo).

Una **cadena** es una secuencia alternada de ejes de un grafo no dirigido y nodos del mismo en la que en cada eje tiene como extremos los nodos que le preceden y le prosiguen. Un ejemplo es la cadena $(1, \{1, 2\}, 2, \{2, 4\}, 4, \{3, 4\}, 3)$ del grafo de la figura 1. Un **camino** es una secuencia alternada de ejes de un grafo dirigido y nodos del mismo en la que para cada eje su nodo inicial es el nodo que le precede y su nodo final es el nodo que le prosigue. $(1, \{1, 2\}, 2, \{2, 4\}, 4, \{4, 3\}, 3, \{3, 1\}, 1)$ es un camino del grafo de la figura 2. Un **camino simple** es un camino en el que no se repite ningún eje. Un **camino elemental** es un camino en el que no se repite ningún nodo.

Un **circuito** de un grafo no dirigido es una cadena donde el primer y el último

nodo coinciden. Un **ciclo** de un grafo dirigido es un camino donde el primer y el último nodo coinciden. El camino puesto como ejemplo en el párrafo anterior es un ciclo del grafo de la figura 2.

Dado un grafo no dirigido, los nodos i, j se dice que están **conectados** si existe una cadena que va de i a j. Todo nodo se considera conectado a sí mismo, aunque no exista en él un bucle. Formalmente, podemos definir la relación de conexión C de la siguiente forma:

$$i\ C\ j \iff \begin{cases} i = j \\ \text{o los nodos } i \text{ y } j \text{ están conectados} \end{cases} \tag{1.2.1}$$

Es inmediato ver que ésta es una relación de equivalencia, lo que nos permite crear una partición de V en la que todos los nodos de cada subconjunto de la partición están conectados entre sí pero desconectados de todos los nodos que no están en ese subconjunto. A cada uno de los subgrafos generados por esos subconjuntos y los ejes que tienen sus nodos por extremos se les llama **componente conexa**. Así, $CC = (V', E')$ es una componente conexa del grafo $G = (V, E)$ si CC es un subgrafo de G tal que $\forall i, j \in V',\ i\ C\ j,\ \forall k \in V \setminus V',\ i$ no $C\ j,$ y $e \in E' \iff e = \{i, j\} \in E \mid i, j \in V'$. Si todo par de nodos de un grafo no dirigido está conectado (o lo que es lo mismo, si el grafo solo tiene una componente conexa), al grafo se le dirá **conexo**. Un grafo dirigido se dirá conexo si su grafo subyacente es conexo.

Prop Si el grafo es conexo, $m \geq n - 1$.

Dem Procedo por inducción. Si $n = 1$ el grafo es conexo por definición y la relación se cumple. Supongo que la relación se cumple para todo $k < n$. Dado un grafo conexo de $n > 1$ nodos, elimino ejes del grafo hasta que este quede dividido en dos componentes conexas. Al tener cada una menos de n nodos la relación debe cumplirse para ambas. Si conectamos dos componentes conexas mediante uno de los ejes que habíamos quitado, el subgrafo que obtenemos tendrá al menos $(k_1 - 1) + (k_2 - 1) + 1 = (k_1 + k_2) - 1 = n - 1$ ejes, donde k_1 y k_2 son el número de nodos de las componentes conexas que estamos uniendo. Luego el nuevo subgrafo también cumple la relación. Como el grafo original debe tener al menos tantos ejes

como este subgrafo, debe cumplir también la relación.

En un grafo dirigido, decimos que un nodo j es **alcanzable** desde i si existe un camino desde el nodo i hasta j. Nótese que el que un grafo dirigido sea conexo no implica que todos sus nodos sean alcanzables desde cualquier otro. Como ejemplo, el grafo de la figura 2 es conexo pero el nodo 1 no es alcanzable desde 3.

Para terminar, consideremos el concepto de red. Una **red** R es una tripleta (V, E, l) donde (V, E) es un grafo y l es una función que asocia a cada eje de $i \in E$ un vector finito de valores $l(i) = (l_1(i), l_2(i), \ldots, l_k(i)$. Cada uno de los valores de este vector representa una característica del eje, como puede ser su capacidad o su coste. En las redes distinguimos dos nodos, el origen o y el destino s. El destino debe ser alcanzable desde el origen. Si una red tuviese originalmente más de un origen o destino, podemos transformarla en una red con un solo origen y un solo destino sin más que introducir un nuevo nodo ascendente de todos los origenes y un nuevo nodo descendiente de todos los destinos, y tratando estos nuevos nodos como el origen y el destino, respectivamente. A los ejes que salen o llegan de estos nuevos nodos les asignamos coste cero y capacidad infinita (a continuación explicaré lo que esto significa).

Formalmente, un **flujo** es una función que asigna a cada eje un valor, $f : E \to \mathbb{R}$. La capacidad de un eje, $u(e)$ es un valor positivo que indica el valor máximo que puede tomar un flujo factible para ese eje. Un flujo se dirá **factible** si nunca toma valores negativos, el flujo de cada eje nunca supera la capacidad de ese eje, y se satisface la conservación del flujo (el flujo que llega a un nodo es igual al flujo que sale de ese nodo) para todos los nodos excepto el origen o y el destino s.

$$f \text{ flujo} \iff \begin{cases} 0 \leq f(e) \leq u(e) \ \forall e \in E \\ \sum\limits_{i \in A(j)} f(\{i, j\}) = \sum\limits_{k \in D(j)} f(\{j, k\}) \forall E \setminus \{o, s\} \end{cases} \tag{1.2.2}$$

Para cada flujo f, llamamos **valor del flujo** a la cantidad de flujo que sale del origen:

$$val(f) = \sum_{j \in D(o)} f(\{o,j\}) - \sum_{i \in A(o)} f(\{i,o\}) \tag{1.2.3}$$

A los ejes, además de una capacidad, les asignaremos también un **coste**, que será el coste por unidad de flujo que pasa por ese arco. Lo denotaremos c_e p c_ij. El coste total de un flujo será:

$$c_G = \sum_{e \in E} f(e)c_e \tag{1.2.4}$$

1.3. Problemas de la teoría de grafos

En esta sección voy a presentar tres problemas muy importantes de la teoría de grafos. En realidad, los dos primeros son casos particulares del último, el problema de flujo de coste mínimo, pero algunos algoritmos resuelven esos casos particulares como paso previo para resolver los más generales, así que es importante entenderlos.

El primer problema que voy a tratar es uno de los problemas más básicos y más útiles de la teoría de grafos, la **búsqueda del camino más corto**. En él partimos de un particular nodo de la red y queremos llegar a otro. El origen será el nodo del que partimos, y el destino, el nodo al que queremos llegar. Cruzar cada eje lleva asociado un coste, y queremos minimizar el coste total que incurrimos al ir desde el origen al destino. Si todos los costes son unitarios, estamos buscando el camino que pasa por el menor número de ejes para ir del origen al destino. Cuando los costes no son unitarios, diremos que estamos haciendo la búsqueda del camino más corto en sentido de costes.

Este problema tiene multitud de aplicaciones directas, pero más interesante para este caso es el hecho de que resolver este problema es un requisito para resolver otros muchos. El algoritmo de sucesivos caminos más cortos que veremos en la sección 2.1 requerirá resolver un gran número de veces la búsqueda del camino más corto, por lo que es importante poder hacer esto de forma eficiente.

Existe una gran cantidad de algoritmos para resolver este problemas, muchos de ellos variaciones de otros, pero los básicos fueron desarrollados en las décadas

de los cincuenta y sesenta. No voy a decir mucho de ellos, ni voy a explicar ninguno con detalle, pero sí que voy a mencionar brevemente el algoritmo que he implementado y que voy a usar a lo largo de este trabajo. El código que he creado puede verse en el anexo.

El algoritmo a usar debe hacer la búsqueda del camino más corto en sentido de costes, admitiendo costes negativos para sus arcos. Esta condición descarta los algoritmos más eficientes para resolver el sistema. Nótese que si hubiese un ciclo que pudiesemos recorrer con coste total negativo no habría camino más corto, ya que sea cual sea el coste que tengamos, podemos reducirlo más dando suficientes vueltas a ese ciclo. Por ello, asumiré que no hay tales ciclos.

El algoritmo escogido se llama algoritmo de corrección de etiquetas. La versión que estoy usando es una versión modificada usando estructuras FIFO. La idea es asignarle a cada nodo j una etiqueta d que nos diga cuál es el coste mínimo con el que se puede llegar a ese nodo desde el origen. Es fácil ver que este valor debe ser $d(j) = \text{mín}\, i \in A(i)\{d(i) + c_{ij}\}$. Inicialmente asignamos a todos los nodos salvo el origen una etiqueta $d(j) = \text{ínf}$ (al origen le corresponde la etiqueta $d(o) = 0$). Acto seguido recorremos el grafo examinando todos los ejes salen de o y actualizando las etiquetas de sus nodos finales siempre que que $d(j) > d(i) + c_{ij}$. Mantenemos una lista de todos los nodos de los cuales no sabemos si sus etiquetas son finales (hemos alcanzado el mínimo). Esta lista sigue una estructura FIFO. En cada iteración tomamos un nodo de la lista y comprobamos las etiquetas de todos los nodos que son descendientes suyos. Cuando hemos acabado, ese nodo sale de la lista, y todos los nodos cuyas etiquetas hemos cambiado entran en la lista. El algoritmo termina cuando no queda ningún nodo en la lista.

Mi implementación del algoritmo encuentra el camino más corto en $O(nm)$. Véase la siguiente sección para una explicación de lo que esto significa.

Esta explicación es bastante superficial pero, dado que este problema no es el objetivo de este trabajo, debería ser suficiente para continuar. Si se quiere tener una comprensión más profunda del algoritmo, véase su sección correspondiente en

el anexo.

Para terminar, decir que para ver este problema como caso particular del problema de flujo de coste mínimo, basta poner todas las capacidades infinitas.

El problema de flujo máximo es otro caso particular de flujo de coste mínimo, que se obtiene cuando todos los costes son cero. Consiste en hallar el flujo de una red que hace máximo el valor del flujo. Aunque hablo del problema de flujo de coste mínimo, en realidad lo que voy a resolver es el problema de flujo máximo de coste mínimo, en el que encontraré un flujo que, en particular, será máximo. Si quisieramos encontrar un problema de flujo de coste mínimo cuyo valor no fuera el máximo de los posibles sino otro inferior, bastaría añadir un nodo ascendente al origen y darle a su eje coste cero y capacidad el valor del flujo que queremos hallar.

Los algoritmos que voy a utilizar para resolver el problema de coste mínimo partiran de un flujo factible y lo irán incrementando mediante un proceso determinado hasta que alcance el valor máximo. Otros algoritmos parten de un flujo máximo y lo van modificando hasta que sea de coste mínimo. Estos algoritmos requieren resolver previamente el problema de flujo máximo, pero como yo no los voy a utilizar, no voy a dar ningún algoritmo para hacerlo. Si alguien está interesado, puede consultar la bibliografía.

Vamos ahora con el problema que realmente nos interesa: el problema de flujo de coste mínimo.

Partimos de una red $G = (V, E, l)$ en la que cada eje $e = \{i, j\}$ tiene asociado un coste por unidad de flujo, $l_1(\{i, j\}) = c_{ij}$, y una capacidad $l_2(\{i, j\}) = u_{ij}$ que indica el flujo f_{ij} que puede pasar como máximo por ese eje. Distinguimos dos nodos especiales, el de origen y el destino. Indicaremos el flujo que estos nodos i producen o absorben por $b(i)$, que será un valor positivo para el origen y negativo para el destino. Para el resto de nodos, $b(i) = 0$. Queremos enviar el flujo máximo posible desde el nodo de origen al de destino, resolviendo el problema de optimización:

$$\text{Min} \quad \sum_{\{i,j\}\in E} c_{ij} f_{ij} \tag{1.3.1a}$$

$$\text{Sujeto a} \sum_{\{j:\{i,j\}\in E\}} f_{ij} - \sum_{\{j:\{j,i\}\in E\}} f_{ji} = b(i) \qquad \forall\, i \in V \tag{1.3.1b}$$

$$0 \leq f_{ij} \leq u_{ij} \qquad \forall \{i,j\} \in E \tag{1.3.1c}$$

La restricción 1.3.1b se conoce como **restricción del equilibrio de masas**, y establece que la diferencia entre el flujo que sale y el que entra en un nodo i ha de ser $b(i)$. La restricción 1.3.1c se llama **restricción de la cota del flujo** y da los valores válidos para el flujo en cada eje.

Para la mayor parte de los algoritmos presentados voy a suponer que las capacidades y los costes son números enteros. Esto no es muy restrictivo en la práctica ya que nuestro objetivo es implementar estos algoritmos en ordenadores que sólo pueden hacer operaciones con racionales. Multiplicando todos los costes y capacidades por un número suficientemente alto (obteniendo una red equivalente en términos de optimización) no tenemos problema en convertir todos los racionales en enteros. De todas formas, también presentaré al final algún algoritmo que funciona con costes y capacidades irracionales.

Antes de cerrar esta sección, veamos algunas aplicaciones de estos problemas.

1.4. Algoritmos: estudio e implementación

Encontrar un algoritmo que resuelva cualquiera de los problemas de la sección anterior no es difícil. Por ejemplo, podemos encontrar el camino más corto recorriendo todos los caminos posibles y calculando el coste de cada uno. Sin embargo, no estamos tan interesados en la existencia de la solución como en hallarla. Para encontrar la solución este método requeriría recorrer una cantidad ingente de caminos, cosa que para muchas redes un ordenador no podría llegar a hacer antes de que se acabase el universo. Es obvio que tenemos que encontrar algoritmos me-

jores, como el que ya he presentado, pero ¿cómo evaluamos qué algoritmo es mejor?

Para el estudio teórico de los algoritmos, la medida más usada es la notación O-grande, que nos da una cota superior del número máximo de operaciones que el algoritmo tendrá que llevar a cabo en el peor de los casos, en función del tamaño de los argumentos de entrada, a partir de un cierto tamaño. Está claro que cuanto más grande sea el problema, más tiempo requerirá, pero no todos los algoritmos evolucionan igual al aumentar ese tamaño. No nos interesará, por tanto, saber qué algoritmo es más rápido en un problema concreto, sino saber qué algoritmo es más rápido asintóticamente, cuando el tamaño del problema tiende a infinito.

Si cuantificamos el tamaño del problema con un número $n \in \mathbb{N}$, podemos dar el tiempo de ejecución de nuestro algoritmo mediante una función $f : \mathbb{N} \to \mathbb{R}$. La mejor forma de hacer esto es tomar n igual al número de bits necesarios para representar el problema, pero en la práctica usaremos otros datos como el número de nodos y ejes de un grafo. Dada una función $g : \mathbb{N} \to \mathbb{R}$, diremos que $f(n)$ es O-grande de $g(n)$ o que $f(n)$ es $O(g(n))$ si existen constantes c y n_0 tales que $|f(n)| \leq c\,|g(n)|\,\forall n > n_0$.

Una caracterización de las $f(n)$ que son $O(g(n))$ vendría dada por el hecho de que $f(n)$ es $O(g(n)) \iff \lim_{n\to\text{ínf}} \frac{f(n)}{g(n)} = c$.

Esta definición nos permitirá distinguir algoritmos según la $g(n)$ de la que es O-grande. Por ejemplo, si $f(n)$ es $O(log(n))$, la función se dirá que tiene complejidad logarítmica y será preferible a una función similar que sea $O(n^p)$, polinomial.

Los algoritmos que voy a utilizar parecen todos polinomiales, con el tamaño del problema representado por el número de nodos n, el número de ejes m, la capacidad del eje con mayor capacidad del grafo, U, y el coste del eje con mayor coste del grafo, C. Por ejemplo, el algoritmo de caminos sucesivos más cortos es $O(nU)$, mientras que el de ajuste repetido de capacidad es $O(m^2 \log n)(m + n \log n))$, que parece, a primera vista, peor. En realidad, esto no es así: el primero, a pesar de su apariencia, no es polinomial.

Un algoritmo es **pseudo polinomial** si es polinomial como función de los

parámetros de entrada, pero no como función de los bits necesarios para almacenar esos parámetros. Escribir la capacidad U requiere $O(log(U))$ bits. Lo mismo pasa con el coste C. Así, como función de los bits necesarios para representar el grafo, el algoritmo que era $O(nU)$ es $O(2^x)$, exponencial, lo cual es mucho peor.

Los algoritmos que son realmente polinomiales, a su vez, pueden dividirse en algoritmos débilmente polinomiales y fuertemente polinomiales. Hasta ahora estábamos considerando que las operaciones eran llevadas a cabo por una máquina de Turing, es decir, que una operación consistía en leer un bit, añadir un bit, cambiar un bit, moverse al bit de la derecha o la izquierda o parar. Por fortuna, los ordenadores actuales más que como máquinas de Turing actúan siguiendo el modelo aritmético, cuyas operaciones básicas son las operaciones aritméticas. Se considera entonces que estas operaciones se realizan en una unidad de tiempo.

Un algoritmo es **fuertemente polinomial** si su número de operaciones (en el modelo aritmético) está acotado superiormente por una función polinomial $g(n)$ donde n es el número de enteros usados como parámetros del problema, y el espacio está acotado superiormente por una función polinomial $h(x)$ donde x es el número de bits usados por el algoritmo.

Por el contrario, un algoritmo es **débilmente polinomial** si es polinomial (según la definición dada al inicio) pero no es fuertemente polinomial.

En nuestro caso, es fácil ver cuándo un algoritmo está en cada una de estas tres categorías. Nuestros algoritmos son $O(g(n, m, U, C)$. Asumiendo que n y m aparecen de forma polinomial (que siempre nos va a pasar), cuando U o C aparezcan de forma lineal, como en $O(nU)$, el algoritmo será solo pseudo polinomial. Cuando U o C aparezcan de forma logarítmica, como en $O((m \log U)(m + n \log n))$, el algoritmo será débilmente polinomial. Cuando no aparezca ni U ni C, el algoritmo será fuertemente polinomial, como en $O((m^2 \log n)(m + n \log n))$. Estos algoritmos suelen ser preferidos porque permiten solucionar los problemas sin que el tiempo de ejecución dependa de los costes y capacidades. Los algoritmos fuertemente polinomiales además resuelven problemas con datos irracionales.

Existen otras formas de comparar algoritmos, como mirar la media del tiempo de ejecución del algoritmo en el peor y el mejor caso, pero son mucho menos usadas. El principal problema que tiene usar el criterio de ser $O(g(n))$ es que solo nos fijamos en el peor caso posible. En muchos escenarios prácticos, este caso puede no alcanzarse nunca. En la práctica, la forma más útil de evaluar si un algoritmo es mejor que otro para resolver un cierto tipo de problema es coger suficientes problemas distintos de ese tipo y ver cuál los resuelve más rápido. Esto es complicado porque tenemos que aislar e ir variando cada uno de los parámetros en cada uno de los problemas e intentar extrapolar cómo se comportaría el algoritmo al variar todo en problemas de ese tipo más grandes.

Una de las motivaciones de este trabajo es ver la relación entre la complejidad teórica y el tiempo de ejecución real. Me parece interesante ver si los algoritmos que mejor se comportan según el criterio explicado en esta sección (el usado siempre en la literatura) son los que más rápido encuentran la solución a los problemas en la práctica. No parece haber motivo por el que esto debiera ser necesariamente así porque los problemas reales no se parecen mucho al peor escenario posible que analizamos teóricamente.

Al calcular la complejidad de un algoritmo hay que tener en cuenta la complejidad de todas sus subrutinas. Por ejemplo, si como paso intermedio un algoritmo utiliza otro, la complejidad del primero será como poco la complejidad del segundo. Ese es el motivo por el que nos interesaba mejorar todo lo posible el algoritmo de búsqueda de caminos más cortos. También tenemos que tener en cuenta las estructuras de datos que usamos, ya que según las que elijamos y cómo estén implementadas, variará el tiempo de ejecución.

2. Algoritmos para el problema de flujo de coste mínimo

Empezaré con cuatro algoritmos para crear el esqueleto del trabajo. Una vez hecho esto, seleccionaré y añadiré tantos algoritmos como pueda sin sobrepasar el límite de páginas.

2.1. Sucesivos caminos más cortos

2.2. Algoritmo de relajación

2.3. Ajuste de capacidad

2.4. Ajuste repetido de capacidad

3. Comparación de algoritmos

3.1. Metodología

¿Grafos generados aleatoriamente? ¿NETGEN? ¿En qué nos vamos a fijar?

Como ya dije en la sección 1.4, el tiempo de ejecución está afectado por el lenguaje usado y decisiones como las estructuras de datos usadas.

En cuanto al lenguaje, todos los algoritmos y análisis de este trabajo están hechos en octave. La elección del lenguaje no fue por ninguna ventaja particular de este. La elección suele basarse en los programas y estructuras ya existentes, pero como yo voy a crear todo lo que necesite esto no es un factor. Hay cosas en las que es peor, como el uso de recursión y listas vinculadas, que evitaré usar cuando sea posible, pero en cualquier caso no es suficiente como para afectar a las conclusiones del trabajo, y por otro lado me permite mostrar los conocimientos adquiridos en este lenguaje a lo largo de la carrera.

En cuanto a estructuras de datos, las redes las almacenaré mediante representaciones en estrella hacia delante y hacia atrás. Esta representación requiere muy poca memoria y realiza todas las operaciones básicas que vamos a usar en grafos en $O(1)$, pero crearla inicialmente es más costoso. Añadir un nuevo nodo o arco tiene complejidad $O(m)$, pero como esas acciones solo se van a realizar antes de empezar a resolver el problema, esto no va a afectar en absoluto al estudio.

En la representación en estrella hacia delante, ordenamos los ejes según el nodo del que emanen (todos los que salen del nodo 1 primero, todos los que salen del nodo 2 después, etc), y les asignamos el número de su orden. Para cada eje tenemos cuatro vectores: *inicial*, $final$, *coste* y *capacidad*, que guardan los extremos y los atributos de cada arco. Así, el eje que ocupa la posición 137 tendrá por nodo inicial $inicial(137)$, por nodo final $final(137)$, por coste $coste(137)$ y por capacidad $capacidad(137)$. Tendremos además un vector *pointer* que señala para cada nodo i el lugar que ocupa el primer eje que sale de i. Así, si $pointer(7) = 34$ y $pointer(8) = 41$, los ejes que salen del nodo 7 ocupan las posiciones 34 a 40. Por

consistencia, si ningún eje sale del nodo i, $pointer(i) = pointer(i+1)$. Además, si el grafo tiene n nodos y m ejes, $pointer(n+1) = m+1$.

La representación en estrella hacia delante nos permite acceder a mucha información muy rápido, pero con ella es difícil encontrar los ejes que llegan a un determinado nodo. Para facilitar esto se le añade la representación en estrella hacia atrás. Lo que hacemos es ordenar los ejes según los nodos a los que llegan y crear un vector *rpointer* que señale para cada nodo i la posición según ese orden del primer eje que llega a i. El resto es igual que para la representación en estrella hacia delante. Para no tener que volver a crear todos los vectores, en su lugar uso un vector *trace* que señala según el orden de la representación hacia atrás la posición que ocupa cada arco en la representación hacia delante.

Para terminar, utilizamos un vector *inverso* que señala para cada eje la posición que ocupa el eje inverso. De este modo, si $e_1 = \{i, j\}$ ocupa la posición 4 en la representación hacia delante y $e_2 = \{j, i\}$ ocupa la posición 19, entonces $inverso(4) = 19$ y $inverso(19) = 4$.

Los grafos que utilizo en este trabajo han sido transformados en grafos equivalentes con los que es más fácil trabajar. En particular, cuando un grafo tiene varios orígenes o salidas, lo transformo en un grafo con un solo origen y una sola salida como vimos en la sección 1.2. Además, si un grafo contiene un eje $e_1 = \{i, j\}$ pero no el eje $e_2 = \{j, i\}$, introduzco este último con capacidad cero y coste $c_{e_2} = -c_{e_1}$. Esto no cambia el grafo (un eje de capacidad cero es como si no existiese en términos de flujo) y me simplifica las operaciones de los algoritmos que voy a aplicar.

3.2. Estudio comparativo

Todos los datos significativos y gráficas obtenidas. Análisis básico.

3.3. Conclusiones

¿Cuál es el mejor algoritmo? ¿Será el mejor en la práctica?

Referencias

[1] Blas Pelegrín, Lázaro Cánovas, Pascual Fernández, *Algoritmos en Grafos y Redes*, PPU, S.A., Barcelona, España, 1992.

[2] Jonathan L. Gross, Jay Jellen, *Graph Theory and its Applications*, Chapman & Hall/CRC, Taylor & Francis Group, Boca Raton, Florida, EEUU, 2006.

[3] Ravindra K. Ahuja, Thomas L. Magnanti, James B.Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, Inc, Upper Saddle River, Nueva Jersey, EEUU, 1993.