Ficha 10

Algoritmos de Base Aleatoria

1.) Algoritmos de Base Aleatoria.

Las estrategias para planteo de algoritmos que hemos enumerado en fichas anteriores (fuerza bruta, divide y vencerás, algoritmo ávidos, etc.) no son las únicas posibles, pero son quizás las más conocidas. Todas ellas se basan en el intento de arribar a un algoritmo (más o menos eficiente) que resuelva un problema, y en todas la idea es que ese algoritmo sea *determinista*.

La característica de un algoritmo determinista es que ante una misma entrada, produce siempre la misma salida. Cada paso que el algoritmo aplica es una consecuencia lógica del estado en que se encontraba en el paso anterior y por lo tanto, un algoritmo determinista correctamente planteado siempre produce la solución correcta al problema analizado.

Está claro que siempre queremos que un algoritmo sea correcto, pero también esperaríamos que un algoritmo sea eficiente al menos en su tiempo de ejecución: ante un gran volumen de datos quisiéramos un tiempo de ejecución de orden polinomial o menor, evitando entrar en el campo de la intratabilidad (problemas cuyas únicas soluciones conocidas son de orden exponencial).

Sin embargo, muchos problemas no son tan simples. Existen problemas que recurrentemente aparecen y requieren soluciones prácticas, pero no siempre esas soluciones prácticas son eficientes. Si la cantidad de posibles soluciones fuese exponencial (por ejemplo), un algoritmo determinista de Fuerza Bruta tendría un tiempo de ejecución exponencial ($O(2^n)$) lo cual es inaplicable incluso para valores de n pequeños. Otras estrategias aplicadas sobre el mismo problema podrían lograr una mejora sustancial en cuanto al tiempo de ejecución, pero podrían ser muy complejas de implementar; o que el tiempo de ejecución no sea exponencial, pero tampoco polinomial; o que el grado de ese tiempo polinomial sea muy alto.

Frente a estos casos existe la alternativa de plantear una *Estrategia de Randomización (*o de *Aleatorización)*. La idea es intentar plantear algoritmos que ya no sean deterministas, sino de *base aleatoria*: ahora, ante la misma entrada, el algoritmo podría producir salidas diferentes ya que el siguiente paso a aplicar surge de algún tipo de selección aleatoria. Y está claro que si interviene el azar, entonces es posible que el algoritmo no llegue eventualmente a una solución correcta.

Lo anterior implica que si se piensa seriamente en aplicar un algoritmo randomizado para un problema dado, ese algoritmo debe ser cuidadosamente analizado desde varios aspectos:

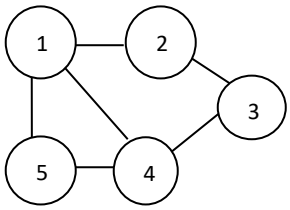
- Por lo pronto, debe quedar claro que el algoritmo valga la pena en cuanto a tiempo de ejecución esperado (por ejemplo, si obtiene un tiempo de ejecución polinómico o que combine una expresión polinómica con otra logarítmica). Si se aplica un algoritmo randomizado que de todos modos tendrá un tiempo de ejecución exponencial o super polinomial, no habrá ningún beneficio y todavía se tendrá la desventaja de eventualmente no obtener la solución correcta.
- Si bien se sabe que un algoritmo randomizado podría no obtener la solución correcta, se espera poder conocer con precisión cuál es la probabilidad de que el algoritmo falle, y diseñarlo de tal modo que esa probabilidad sea realmente baja o muy baja.

- Y finalmente, en la medida de lo posible, se esperaría que el algoritmo planteado sea relativamente simple de implementar (cosa que suele ser el caso de los algoritmos randomizados).

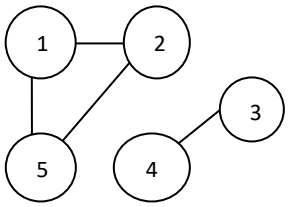
2.) Caso de Análisis: El Problema del Corte Mínimo en un Grafo No Dirigido.

A modo de primera experiencia práctica, intentaremos mostrar la forma de aplicar un algoritmo randomizado sobre un conocido problema de ámbito de los grafos: el problema del *Corte Mínimo de un Grafo*[1].

Sea *G* un grafo *G(V, A)* no dirigido y no ponderado, en el cual *V* es el conjunto de *n* vértices o nodos del grafo, y *A* es el conjunto de *m* arcos o aristas del grafo. En general, un grafo *G* se dice *conexo* si para cada par de vértices de este, existe un camino que los vincula (en otras palabras, no hay subconjuntos de vértices disjuntos o separados del resto de los vértices):



a.) un grafo conexo



b.) un grafo no conexo

A su vez, se llama *componente conexa* de un grafo *G* a cada subgrafo conexo que no esté vinculado con otros subgrafos conexos. La figura *b* de los ejemplos anteriores, tiene dos componentes conexas: la primera está conformada por el conjunto de tres vértices *{1, 2, 5}* vinculados entre sí, y la segunda por el conjunto *{3, 4}* con los otros dos. Está claro que si un grafo es conexo, entonces tiene una y sólo una componente conexa que coincide con el propio grafo (figura *a*)

Un conjunto de arcos tal que si se elimina produce que el grafo deje de ser conexo se designa como un *corte* del grafo. En la figura *a*, un posible corte es el conjunto que contiene a los arcos {(1,4), (5,4), (2,3)} ya que si eliminan esos tres arcos, el grafo pasa a tener dos componentes conexas: {1, 2, 5} y {3, 4}. Pero está claro que un grafo puede tener varios cortes posibles válidos: en el mismo grafo de la figura *a*, otro corte posible es el conjunto de arcos {(1,2), (4,3)} que lleva al grafo a tener las dos componentes conexas {1, 4, 5} y {2, 3}.

En algunas situaciones prácticas, se requiere poder identificar un *corte mínimo* para un grafo *G*: esto es, un corte de entre los muchos válidos, que tenga la menor cantidad posible de arcos (en otras palabras, la mínima cantidad de arcos tal que si se eliminan, provocan que el grafo deje de ser conexo). El grafo de la figura *a* admite diversos cortes mínimos de tamaño 2.

Este requerimiento surge, por ejemplo, en problemas en los que el grafo modela una red de algún tipo y se quieren determinar posibles cuellos de botella en esa red: el conjunto de arcos más pequeño por el que fluyen elementos de una parte del grafo hacia otra.

El problema de determinar un corte mínimo podría intentar resolverse por Fuerza Bruta, identificando todos y cada uno de los posibles cortes (mínimos o no) que el grafo tenga, y quedándose con aquel que

[1] La explicación que sigue sobre el problema del Corte Mínimo está inspirada en el material del curso "*Design and Analysis of Algorithms I*" – Stanford University (a cargo de *Tim Roughgarden*, Associate Professor): https://www.coursera.org/courses. Algunas demostraciones matemáticas y de cálculo de probabilidades se omiten aquí, pero pueden consultarse en la fuente original citada.

tenga la menor cantidad de arcos. Sin embargo, esta idea puede ser muy mala ya que un rápido análisis muestra que en un grafo conexo de *n* vértices, la cantidad total de cortes diferentes puede llegar a 2^n si el grafo es denso (cantidad de arcos *m* en el orden de n^2).

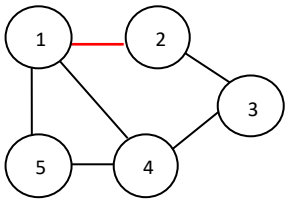
Existen algoritmos deterministas que resuelven este problema, basándose en que puede establecerse una relación con el problema del *Flujo de Fluidos* en una red. En ese sentido, a partir del algoritmo de *Edmonds-Karp*[2] puede obtenerse una solución cuyo tiempo de ejecución es $O(n*m^2)$, mientras que la solución de *Goldberg-Tarjan*[3] tiene un tiempo de $O(n*m*log(n^2/m))$, siendo *n* el número de vértices y *m* el número de arcos.

Una posible mejora para esos tiempos de ejecución fue propuesta en 1993 por *David Karger*. El *algoritmo de Karger* es esencialmente un algoritmo randomizado, que permite encontrar un corte mínimo para un grafo en base a repetir la idea de *contracción de arcos*.

Básicamente hablando, una *contracción* es la operación por la cual un *arco (u, v)* que une los vértices *u* y *v* de un grafo, *fusiona* los vértices *u* y *v* en uno solo *uv*, eliminando el arco y reduciendo el número total de vértices en uno. Todos los arcos que incidían en *u* o en *v*, se suponen redirigidos hacia el nodo *uv* resultante de la contracción.

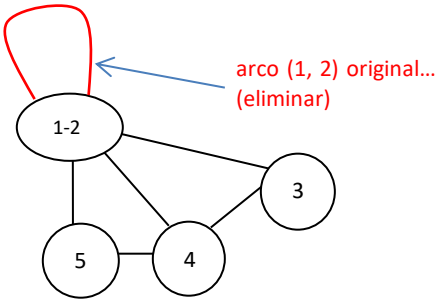
A modo de ejemplo, volvamos al grafo de la figura *a*. Si aplicamos una contracción sobre el arco *(1, 2)*, entonces los nodos *1* y *2* serían fusionados en uno solo que designaremos como *1-2*. Los arcos que incidían en el nodo *1* o en el nodo *2*, ahora inciden en el nodo *1-2*.

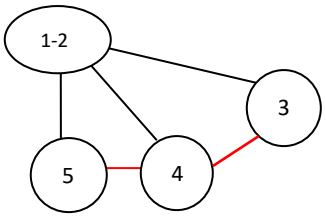
El propio arco *(1, 2)* que se contrajo, inicialmente forma un *auto ciclo* sobre el nodo fusionado *1-2* (un arco que parte del nodo *1-2* y regresa al mismo nodo *1-2*). El proceso de contracción debe eliminar el auto ciclo, y finalmente el grafo resultante se vería como en la figura *c* que sigue a continuación:



a. un grafo conexo…

contraer (1, 2) ⇨

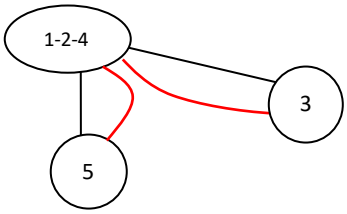




b. grafo resultante

Está claro que si se aplica esta idea, una contracción podría producir lo que se designa como *arcos paralelos*: arcos diferentes (pero válidos) que conectan los mismos nodos. En el caso de la figura *b*, si se decide contraer el arco *(1-2, 4)*, el grafo resultante quedará como sigue:

[2] Fuente: http://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp_algorithm.
[3] Fuente: http://en.wikipedia.org/wiki/Push%E2%80%93relabel_maximum_flow_algorithm.



c. arcos paralelos…

El arco original (5, 4) se convierte ahora en *otro* arco (1-2-4, 5), lo cual es válido. Y el arco original (4, 3) pasa a ser un segundo arco (1-2-4, 3), también válido.

Con estas definiciones e ideas ya expresadas, mostramos un pseudocódigo del algoritmo de Karger para obtener el corte mínimo de un grafo:

Entrada: un grafo *G(V, A)*, no dirigido, no ponderado, conexo y posiblemente denso, tal que *G* admite arcos paralelos (*V* es el conjunto de *n* vértices, *A* es el conjunto de *m* arcos).

Salida: un *corte mínimo* para *G*.

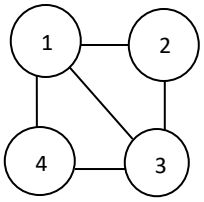
Pseudocódigo:

min_cut(G):
- mientras G contenga más de dos vértices:
  - seleccione aleatoriamente un arco a(u, v) de G.
  - contraiga los nodos u y v en un nuevo vértice uv.
  - elimine los auto ciclos.

- retorne el corte representado por los dos vértices finales.

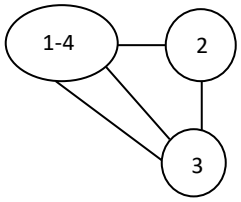
Lo que el algoritmo retorna es el grafo final reducido a dos vértices, con todos los arcos que quedaron luego de todas las contracciones. Dijimos que un corte es un conjunto de arcos, y el algoritmo está retornando un grafo, por lo cual hacemos notar especialmente la palabra *representado* en la última línea del pseudocódigo: el corte obtenido, es el conjunto de arcos que quedaron en el grafo retornado.

El algoritmo así expresado es asombrosamente simple, pero tiene un pequeño problema: no garantiza que el resultado obtenido sea siempre correcto. De acuerdo a los arcos que se decida contraer, podría llegar a ocurrir que al final se retorne un corte que no sea mínimo. Por ejemplo, si el grafo fuese el siguiente:

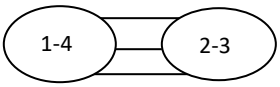


Este grafo tiene al menos un *corte mínimo de tamaño 2*, que contiene a los arcos *(1, 4)* y *(4, 3)*: si esos dos arcos se eliminan, se llega a las dos componentes conexas *{4}* y *{1, 2, 3}*. Pero si se aplica el algoritmo

de Karger y se comienza contrayendo los arcos *(1, 4)* y luego *(2, 3)*, quedaría un grafo de dos vértices (con lo cual el algoritmo se detiene), pero con *tres* arcos de cruce (en lugar de los *dos* que se esperarían en el corte mínimo):



luego de contraer el arco (1, 4)…



luego de contraer el arco (2, 3)…

Si el algoritmo puede fallar, entonces… ¿cuál es la probabilidad de éxito? y ¿qué podemos hacer para aumentar esa probabilidad? Si se realiza un estudio de probabilidades detallado (cosa que no haremos aquí…) puede deducirse que la probabilidad *P(C)* de obtener un corte mínimo *C* en una sola aplicación del algoritmo de Karger, es:

$$P(\text{éxito en obtener un corte mínimo}) = P(C) >= 1/n^2$$

siendo *n* el número original de vértices del grafo. Esta probabilidad es lamentablemente baja… si el grafo tuviese *n = 100* vértices (un número relativamente pequeño de vértices), entonces $P(C) >= 1 / 100^2 = 1 / 10000 = 0.0001$, con lo cual no parece que valiese la pena aplicar el algoritmo.

Pero la idea final detrás de un algoritmo randomizado es aumentar la probabilidad de éxito aplicando un gran número de veces *T* en forma repetida el algoritmo sobre el mismo conjunto de datos (en nuestro caso, sobre el mismo grafo inicial). Otra vez, diversos cálculos de probabilidades muestran que si el algoritmo se aplica un número *T* de veces, entonces la probabilidad *P(FT)* de que *todos* los intentos *fallen*, es:

$$P(\text{falla en } todos \text{ los T intentos}) = P(FT) <= (1 - 1/n^2)^T$$

Y finalmente, entonces, puede verse que:

si $T = n^2$ entonces $P(FT(T=n^2)) <= e\ (-(1/n^2))^{n2} = 1 / e$
si $T = n^2 * \ln(n)$ entonces $P(FT(T=n^2*\ln(n))) <= e\ (1/e)^{\ln(n)} = 1 / n$

Si el número de repeticiones a aplicar es $T = n^2 * ln(n)$, entonces la probabilidad final de fallar se reduce a medida que *n* es mayor… para *n = 100* vértices, el programa debería aplicar el algoritmo de Karger una cantidad $T = 10000 * ln(100) = 10000 * 4.6 = 46000$ para reducir la probabilidad de falla a $P(FT) <= 1/100 = 0.01$, y si *n = 1000*, entonces deberá aplicar el algoritmo una cantidad de veces $T = 1000000 * 7 = 7000000$ (siete millones…) para reducir la probabilidad de falla a $P(FT) <= 0.001$.

El pseudocódigo final, podría entonces quedar como sigue:

Entrada: un grafo *G(V, A)*, no dirigido, no ponderado, conexo y posiblemente denso, tal que *G* admite arcos paralelos (*V* es el conjunto de *n* vértices, *A* es el conjunto de *m* arcos).

Salida: un *corte mínimo* para *G*.
Pseudocódigo: (Suponiendo que se elige $T = n^2 * ln(n)$)

min_cut(G):

- sea T = n^2 * ln(n) (con n el número de vértices en G).
- repetir T veces:
  - ✓ Obtenga una copia clonada G' de G.
  - ✓ mientras G' contenga más de dos vértices:
    - tome aleatoriamente un arco a(u, v) de G'.
    - contraiga los vértices u y v en un nuevo vértice uv.
    - elimine los auto ciclos.
  - ✓ quédese con el corte obtenido que tenga el menor número de arcos
- retorne el corte menor que haya obtenido.

La pregunta final es: ¿vale la pena implementar el *algoritmo de* Karger para hacer *T* repeticiones, si el número de repeticiones será tan alto? La respuesta es que eso depende del tiempo de ejecución de cada aplicación del algoritmo básico. Si el grafo se implementa con estructura de listas de adyacencia, entonces una corrida del algoritmo básico se ejecuta en un tiempo de $O(n^2)$, siendo *n* el número de vértices del grafo original. Por lo tanto, si se hacen $T = n^2 * ln(n)$ repeticiones, puede probarse que el tiempo total de ejecución será $O(n^2 * m * log(n))$. Este resultado no mejora el rendimiento asintótico del algoritmo de *Goldberg-Tarjan*, pero en contrapartida, el *algoritmo de Karger* es más simple de implementar… y de todos modos, ¡sirve como modelo para un estudio inicial de la estrategia de randomización!

Se deja para los estudiantes la implementación del algoritmo.