



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Algoritmos y Estructuras de Datos III
Primer Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Aldasoro Agustina	86/13	agusaldasoro@gmail.com
Noriega Francisco	660/12	frannoriega.92@gmail.com
Zimenspitz Ezequiel	155/13	ezeqzim@gmail.com
Zuker Brian	441/13	brianzuker@gmail.com



Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

<http://www.fcen.uba.ar>

Poner resumen.

Índice

1. Dakkar	3
1.1. Descripción de la problemática	3
1.2. Resolución propuesta y justificación	4
1.3. Análisis de la complejidad	5
1.3.1. Complejidad Temporal	5
1.3.2. Complejidad Espacial	5
1.4. Código fuente	5
1.5. Experimentación	5
1.5.1. Contrastación Empírica de la complejidad	5
2. Zombieland II	6
2.1. Descripción de la problemática	6
2.2. Resolución propuesta y justificación	6
2.3. Análisis de la complejidad	6
2.4. Código fuente	6
2.5. Experimentación	6
2.5.1. Contrastación Empírica de la complejidad	6
3. Refinando petróleo	7
3.1. Descripción de la problemática	7
3.2. Resolución propuesta y justificación	8
3.3. Análisis de la complejidad	9
3.4. Código fuente	11
3.5. Experimentación	14
3.5.1. Contrastación Empírica de la complejidad	14

1. Dakar

1.1. Descripción de la problemática

La problemática trata de una travesía, la cual cuenta con n cantidad de etapas. Para cada una de las etapas, se puede elegir recorrerla en alguno de los tres vehículos disponibles: una BMX, una motocross o un buggy arenero. Cada uno de ellos permite concretar cada etapa en cantidades de tiempo diferentes. Además, la cantidad de veces que se pueden usar la motocross y el buggy arenero está acotada por k_m y k_b respectivamente.

Los *tiempos* que le llevan a los vehículos recorrer el trayecto varían por cada etapa y son datos conocidos pasados por parámetro.

Se pide recorrer la travesía, dentro de las restricciones, de modo que se utilice la menor cantidad de tiempo posible. Si existen dos (o más) maneras de atravesarla dentro del tiempo óptimo, se pide devolver sólo una.

Se exige resolver la problemática con una complejidad temporal de $O(n.k_m.k_b)$.

Dibujitos con ejemplos :)

1.2. Resolución propuesta y justificación

Para resolver esta problemática, optamos por implementar un algoritmo de *Programación Dinámica*.

Con el fin de encontrar el recorrido factible que emplee menos tiempo; debemos comparar, para cada etapa, cuál es el menor tiempo con el que puede recorrer el camino faltante eligiendo en la instancia actual uno de los tres vehículos disponibles. Dado que la formulación de este problema es muy extensa, se realizó una formulación recursiva de modo que para cada problema se le asigna un valor dependiendo de un subproblema menor.

Formulación Recursiva

Optamos por comenzar recorriendo desde la etapa n hasta la etapa 0; n va a indicar la etapa actual, k_m la cantidad de motos y k_b la cantidad de boogys restantes que se pueden utilizar.

- Cuando llegamos a la etapa $n=0$ es porque terminamos todo el recorrido, de modo que el tiempo devuelto va a ser 0.
- Cuando $k_m=0$ y $k_b=0$ es porque la etapa actual (n) y el recorrido restante (las $n-1$ etapas) lo vamos a tener que hacer sólo en bicicleta, sin importar el tiempo que conlleve ya que nos quedamos sin motos y boogys para usar.
- Cuando $k_m=0$ y $k_b \neq 0$ es porque utilizamos la mayor cantidad de motos posibles y las $n-1$ etapas restantes -conjunto a la actual(n)- las vamos a tener que recorrer con Bicicleta o Boogy. Por este motivo se elige la opción con tiempo menor usando Bicicleta o Boogy en la etapa n y llamando recursivamente a la función para $n-1$ considerando esta elección.
- De modo análogo, cuando $k_m \neq 0$ y $k_b=0$ sólo vamos a contar con Motos y Bicicletas para la etapa actual y las $n-1$ etapas faltantes.
- En cambio, en caso contrario, todavía tenemos disponible cantidad de los tres vehículos. Por este motivo, se comparan los tres casos: empleando la Bicicleta en la etapa n , la Moto o el Boogy llamando recursivamente a la función para $n-1$ de modo que va a devolver el menor tiempo posible considerando la elección llevada a cabo.

$$func(n, k_m, k_b) = \begin{cases} 0 & \text{si } n = 0 \\ tiempoBici(n) + f(n-1, 0, 0) & \text{si } k_m = 0 \wedge k_b = 0 \\ \min \left(\begin{array}{l} tiempoBici(n) + func(n-1, 0, k_b), \\ tiempoBoogy(n) + func(n-1, 0, k_b-1) \end{array} \right) & \text{si } k_m = 0 \wedge k_b \neq 0 \\ \min \left(\begin{array}{l} tiempoBici(n) + func(n-1, k_m, 0), \\ tiempoMoto(n) + func(n-1, k_m-1, 0) \end{array} \right) & \text{si } k_m \neq 0 \wedge k_b = 0 \\ \min \left(\begin{array}{l} tiempoBici(n) + func(n-1, k_m, k_b), \\ tiempoMoto(n) + func(n-1, k_m-1, k_b), \\ tiempoBoogy(n) + func(n-1, k_m, k_b-1) \end{array} \right) & \text{sino} \end{cases}$$

Dado que los n , k_m y k_b iniciales van a ser los dados por parámetro y en el planteo de nuestra ecuación en la llamada recursiva n siempre decrementa en 1 y los demás o bien quedan iguales o uno de ellos decrementa en uno, estos parámetros van a estar acotados por:

$$\begin{aligned} 0 &\leq n \leq n_{parametro} \\ 0 &\leq k_m \leq k_{m,parametro} \\ 0 &\leq k_b \leq k_{b,parametro} \end{aligned}$$

1.3. Análisis de la complejidad

1.3.1. Complejidad Temporal

1.3.2. Complejidad Espacial

Si bien, ya no piden ningún requisito, pongamos cuanta memoria usa :)

1.4. Código fuente

1.5. Experimentación

1.5.1. Constrastación Empírica de la complejidad

2. Zombieland II

2.1. Descripción de la problemática

2.2. Resolución propuesta y justificación

2.3. Análisis de la complejidad

2.4. Código fuente

2.5. Experimentación

2.5.1. Constrastación Empírica de la complejidad

3. Refinando petróleo

3.1. Descripción de la problemática

Dentro de una locación dada, se cuenta con n pozos de extracción de petróleo. Como el petróleo luego de ser extraído debe ser refinado, se nos pide diagramar la construcción de refinerías y tuberías para hacerlo posible.

Para armar esta red de refinamiento se podrán construir refinerías, colocadas en cada pozo, y tuberías que conectan los distintos pozos entre sí. Como todos los pozos deben tener un camino factible, para llegar a alguna planta procesadora, se les deberá colocar una planta de refinamiento en el mismo, o bien, armar un camino de tuberías que lo conecte con una.

Las refinerías tienen un costo fijo, independiente del pozo donde se ubiquen, mientras que el costo de las tuberías depende de los pozos que se quiera conectar. Además, contamos con la restricción de que sólo se pueden construir tuberías entre los pozos que nos son indicados. Todos estos datos nos son dados como parámetros de la función.

Se desea escribir un algoritmo que de una distribución de refinerías y tuberías tal que todo pozo tenga acceso a una refinería y minimice el costo de la inversión. Se debe indicar cuántas refinerías construir y en qué pozos, así como también cuántas tuberías y entre que pozos.

El algoritmo debe tener complejidad mejor que $O(n^3)$ siendo n la cantidad de pozos de la zona.

Insertar ejemplito con dibujito :)

3.2. Resolución propuesta y justificación

Dada la estructura del problema presentado, se optó por modelar la situación mediante un grafo no dirigido. En este mismo, cada nodo presenta un pozo petrolero y los ejes indican la posibilidad de la construcción de tuberías. Es decir, dados dos nodos e y f del grafo, va a existir un eje $w = (e, f)$ si y solo si se puede construir una tubería entre e y f . Además, cada eje va a contar con un peso que indica el costo de construcción de su tubería.

Este grafo inicial se va a armar con los datos pasados por parámetro. En primera instancia, se guardan los ejes que se obtienen de la *entrada standard* como una lista de adyacencia.

A partir de aquí vamos a trabajar con un algoritmo particular sobre *árboles generadores mínimos*: **algoritmo de Kruskal**.

El grafo obtenido hasta ahora puede tener desde 1 hasta n componentes conexas, para cada una de ellas vamos a formar su árbol generador mínimo. Esto nos asegurará formar un árbol por componente tal que su peso total sea el mínimo de todos los árboles factibles. Al saber que los pesos de los ejes son todos positivos, contamos con que el único camino resultante entre dos nodos va a ser el camino de menor costo de los caminos originales.

El algoritmo de Kruskal es un algoritmo goloso que está implementado para grafos. Lo primero que realizamos fue ordenar los ejes que tenemos del grafo, por su peso en orden creciente. A continuación, se recorren los ejes secuencialmente y para cada uno se decide si debe permanecer o se debe quitar, de modo que si el eje actual no genera un ciclo (con los elegidos hasta esta instancia) permanece, en caso contrario se elimina.

Además, cuenta con una restricción adicional. Cada eje que revisa va a permanecer sólo si su peso es menor al costo de construir una refinería. Esta nueva condición, nos asegura que para cada pozo su camino para llegar a una refinería va a ser el mínimo. **Poner un ejemplito**

Una vez que obtuvimos los ejes definitivos, nos queda simplemente contar cuántas refinerías hay que colocar (una, por cada componente conexa -árbol- que formen los ejes seleccionados) y ubicarla en cualquier nodo de la componente.

Llevar la cuenta de cuánto se lleva gastado es sumar los costos de cada tubería colocada más la cantidad de refinerías por su costo, y así obtenemos el costo total.

3.3. Análisis de la complejidad

Dados los datos ingresados como parámetros, contamos con el costo de construir una refinería y un grafo no dirigido de forma tal que la cantidad de nodos están numerados (arbitrariamente) de 0 a $n - 1$ y posee m ejes. Tomamos como notación $n = \text{cantNodos}$ y $m = \text{cantEjes}$.

Es importante destacar que la cantidad de ejes está acotada por $O(n^2)$; ya que cada nodo, en el peor caso, puede conectarse con todos los nodos del grafo, excepto consigo mismo, es decir que para cada nodo, existen a lo sumo $n-1$ ejes que entran y salen del mismo.

El grafo que construimos con los datos pasados como parámetro va a estar representado mediante una lista de adyacencia, la cual va a ser un vector de ejes. Esto presenta un costo de $O(m)$, como $m \leq n^2$ pertenece a $O(n^2)$.

Luego, vamos a utilizar la estructura *Union-Find*^{1 2}. Esta estructura (adjuntada en el archivo *Union-Find.h*) consiste de un vector capaz de almacenar diversos conjuntos disjuntos -cada uno con un *Elemento Representante*-, la cual cuenta con los siguientes tres métodos: `find_set(x)`, `union_set(x, y)` y `is_in(x, y)`.

`find_set(x)` devuelve el elemento representante del conjunto al que pertenece x .

`union_set(x, y)` une al conjunto que contiene a x con el conjunto al que pertenece y .

`is_in(x, y)` devuelve true si x está en el mismo conjunto que y , false en caso contrario.

El algoritmo de Kruskal se apoya en esta estructura. Nuestra función `generarArbolesMinimos` recibe como parámetros de entrada al vector de ejes totales, el costo de construir una refinería y la cantidad de pozos (n).

Como primer paso, ordena los ejes mediante el algoritmo `sort()`³ de la librería Standard de C, cuya complejidad es $O(m \cdot \log(m))$. Lo cual es equivalente a $O(n^2 \cdot \log(n^2))$, por propiedades de logaritmo es $O(n^2 \cdot 2 \cdot \log(n))$ y eliminando constantes es $O(n^2 \cdot \log(n))$.

Luego, se recorren todos los ejes del vector obtenido como parámetro (ejes), ya ordenados. De modo que, primero nos ubicamos en el eje de menor costo, terminando el recorrido con el de mayor. Contamos con un nuevo vector `res`, el cual va almacenar sólo los ejes que esten en el resultado, el cual comienza vacío. Para cada eje recorrido en `ejes`, se consulta si al insertarlo en `res` formaría un ciclo con los que ya se encuentran en él. Si esta consulta es positiva, se saltea este eje; en caso contrario se inserta al vector `res` sólo si su costo es menor al costo de construir una refinería.

Además vamos a contar con una estructura *Union-Find* (`bosqueMinimo`), la cual va a representar los conjuntos disjuntos que forman las componentes conexas del grafo. De modo que, al momento de insertar un nuevo eje a `res` dentro del ciclo también se uniran los dos nodos que une este eje dentro de su conjunto correspondiente en (`bosqueMinimo`).

Este paso aprovecha la estructura de conjuntos disjuntos *Union-Find*, para ver si dos conjuntos son disjuntos y eventualmente unirlos eficientemente.

¹Introduction to Algorithms (Third Edition) - Cormen, Leiserson, Rivest, Stein [2009], Unidad 21.

²Gregory C. Harfst and Edward M. Reingold. A potential-based amortized analysis of the union-find data structure, 2000.

³<http://www.cplusplus.com/reference/algorithm/sort/>

Dado que lo implementamos con las heurísticas de *path compression* y *union by rank*, la complejidad de m operaciones `find_set` y `union_set` más una llamada a `make_set` de complejidad $O(n)$, ejecutan en tiempo $O(m\alpha(n))$, donde $\alpha(n)$ es la inversa de *función de Ackerman*²:

$$\begin{aligned} A(m, n) &= (2 \uparrow^{m-2} (n+3)) - 3 \\ \alpha(n) &= \min\{k \in N_0 : A(k, 1) \geq n\} \end{aligned}$$

Algunos valores de la *función de Ackerman* se presentan a continuación:

$$\begin{aligned} A(0, 1) &= 2 \\ A(1, 1) &= 3 \\ A(2, 1) &= 7 \\ A(3, 1) &= 2047 \\ A(4, 1) &= 10^{80} \end{aligned}$$

Como A crece excesivamente rápido, α crece excesivamente lento:

$$\alpha(n) = \begin{cases} 0 & \text{si } 0 \leq n \leq 2 \\ 1 & \text{si } n = 3 \\ 2 & \text{si } 4 \leq n \leq 7 \\ 3 & \text{si } 8 \leq n \leq 2047 \\ 4 & \text{si } 2048 \leq n \leq A(4, 1) \end{cases}$$

Esto nos indica que para casi cualquier caso práctico que empleemos (hasta 10^{80} nodos), $\alpha(n)$ será a lo sumo 4. Por lo tanto, la complejidad de estas operaciones es $O(m)$ o lo que es lo mismo $O(n^2)$.

Tener en cuenta que, dentro del ciclo también se ejecutan asignaciones y `push_back()`⁴ en un vector pero estas operaciones cuentan con un costo $O(1)$ (amortizado en el caso de `push_back`), lo cual es despreciable.

Finalmente devuelve el vector resultante por copia, lo que suma, en el peor caso, $O(n)$ dado que un árbol generador tiene a lo sumo $n - 1$ ejes.

Hasta este punto, la complejidad es la suma de las mencionadas: $O(n^2) + O(n^2 \cdot \log(n)) + O(n^2) = O(n^2 \cdot \log(n))$.

Resta hacer un recorrido lineal en los ejes del árbol generador mínimo que nos devolvió el algoritmo de *Kruskal*, armando un nuevo *Union-Find* para distinguir cuáles son las componentes conexas resultantes. De esta manera, vamos a determinar qué componentes necesitan una refinera (determinamos que el elemento representante de cada conjunto).

Esto tarda $O(n)$ ya que, como se indicó anteriormente, a lo sumo son $n - 1$ ejes.

La cuenta final resulta $O(n) + O(n^2 \cdot \log(n)) = O(n^2 \cdot \log(n))$, la cual es estrictamente mejor que $O(n^3)$ como se pedía en el enunciado.

⁴http://www.cplusplus.com/reference/vector/vector/push_back/

3.4. Código fuente

```
class UnionFind {
public:
    UnionFind(int tamano);
    ~UnionFind();
    int find_set(int x);
    void union_set(int x, int y);
    bool is_in(int x, int y);

private:
    vector<int> parent;
    vector<int> rank;
};
```

```
UnionFind::UnionFind(int tamano){
    parent = vector<int>(tamano);
    rank = vector<int>(tamano);
    //cada indice es su propio representante, su ranking es 0
    for (int i = 0; i < tamano; ++i) {
        parent[i] = i;
        rank[i] = 0;
    }
}
```

```
int UnionFind::find_set(int x) {
    //si es representante lo devuelvo, si no lo hago apuntar directamente al representante
    //para que llamadas consecutivas cuesten tiempo constante
    if(parent[x] != x)
        parent[x] = find_set(parent[x]);
    return parent[x];
}
```

```
void UnionFind::union_set(int x, int y) {
    //buscamos representantes de ambos elementos
    //requiere que no esten en el mismo conjunto
    int rx = find_set(x);
    int ry = find_set(y);
    //incluyo el de menor ranking en el de mayor para mantener balanceada la estructura
    if(rank[rx] < rank[ry]){
        parent[rx] = ry;
    }
    else{
        parent[ry] = rx;
        if(rank[ry] == rank[rx])
            rank[rx]++;
    }
}
```

```
bool UnionFind::is_in(int x, int y) {
    return find_set(x) == find_set(y);
}
```

```
struct eje {
    unsigned int pozoA;
    unsigned int pozoB;
    unsigned int costoTuberia;
    bool operator< (const eje& otro) const{
        return costoTuberia < otro.costoTuberia;
    }
};

int main(int argc, char const *argv[]){
    unsigned int pozos, cantConexiones, costoRefineria;
    unsigned int pozoA, pozoB, costoTuberia;
    cin >> pozos >> cantConexiones >> costoRefineria;
    vector<eje> ejes;
    //leemos la entrada, la almacenamos en ejes
    for (int i = 0; i < cantConexiones; ++i){
        cin >> pozoA >> pozoB >> costoTuberia;
        pozoA--;
        pozoB--;
        eje conex;
        conex.pozoA = pozoA;
        conex.pozoB = pozoB;
        conex.costoTuberia = costoTuberia;
        ejes.push_back(conex);
    }
    //aplicamos el algoritmo
    refinandoPetroleo(ejes, pozos, costoRefineria);
    return 0;
}
```

```

int refinandoPetroleo(vector<eje>& ejes, int cantPozos, int costoRefineria){
//generamos los arboles minimos para cada componente conexas,
//en realidad solo los ejes que cuesten menos que poner refinarias
    vector<eje> conexionesMinimas = generarArbolesMinimos(ejes, costoRefineria, cantPozos);
    UnionFind conexos(cantPozos);
    int costoTotal = 0, cantRef = 0;
//armamos un union-find para identificar componentes triviales, en las demas solo hara falta
//poner una refineria en el representante de la componente pues los tubos son mas baratos
//para unir los distintos pozos
    for (int i = 0; i < conexionesMinimas.size(); ++i) {
        conexos.union_set(conexionesMinimas[i].pozoA, conexionesMinimas[i].pozoB);
        costoTotal += conexionesMinimas[i].costoTuberia;
    }
//en las componentes triviales van refinarias
    vector<bool> refinarias(cantPozos);
    for (int i = 0; i < cantPozos; ++i) {
        if(conexos.find_set(i) == i){
            refinarias[i] = true;
            costoTotal += costoRefineria;
            cantRef += 1;
        }
    }
//cout pedido
    cout << costoTotal << " " << cantRef << " " << conexionesMinimas.size() << endl;
    for (int i = 0; i < refinarias.size(); ++i) {
        if(refinarias[i])
            cout << i+1 << " ";
    }
    cout << endl;
    for (int i = 0; i < conexionesMinimas.size(); ++i) {
        cout << conexionesMinimas[i].pozoA+1 << " " << conexionesMinimas[i].pozoB+1 << endl;
    }
    return costoTotal;
}

```

```

vector<eje> generarArbolesMinimos(vector<eje>& ejes, int costoRefineria, int cantPozos){
    UnionFind bosqueMinimo(cantPozos);
    vector<eje> res;
//ordenamos los ejes segun su costo para obtener en tiempo constante, los menores
    sort(ejes.begin(), ejes.end());
    for (int i = 0; i < ejes.size(); ++i) {
//si agregar el eje no forma ciclo
        if(!bosqueMinimo.is_in(ejes[i].pozoA,ejes[i].pozoB)){
//lo uno y si cuesta menos que poner una refineria, lo agrego a res
            bosqueMinimo.union_set(ejes[i].pozoA, ejes[i].pozoB);
            if(ejes[i].costoTuberia < costoRefineria){
                eje conex;
                conex.pozoA = ejes[i].pozoA;
                conex.pozoB = ejes[i].pozoB;
                conex.costoTuberia = ejes[i].costoTuberia;
                res.push_back(conex);
            }
        }
    }
    return res;
}

```

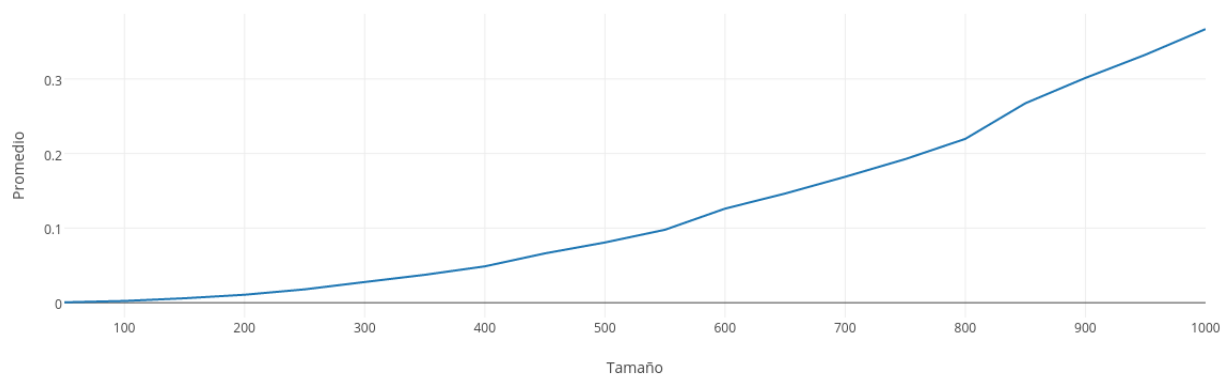
3.5. Experimentación

3.5.1. Contrastación Empírica de la complejidad

Para llevar a cabo esta experimentación, consideramos el peor caso posible de cantidad de ejes del grafo, es decir que habrá exactamente $n \cdot (n - 1)$ ejes (cada nodo puede conectarse con cualquier nodo del grafo, excepto consigo mismo), variando la cantidad de nodos.

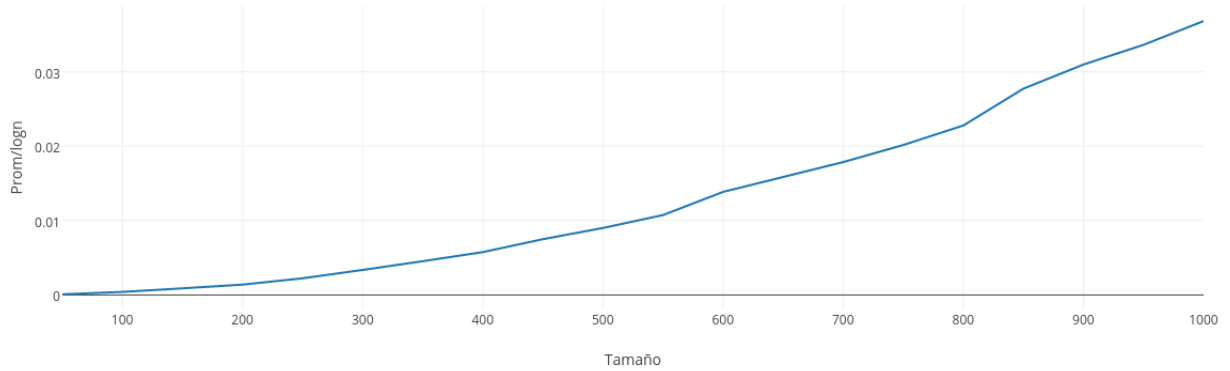
Los tiempos de ejecución para cada n (cantidad de nodos) fueron los siguientes:

n	Tiempo en segundos
50	0.0005254864
100	0.002332719
150	0.0059328642
200	0.01065945
250	0.0178728144
300	0.0277215188
350	0.0373807766
400	0.0487278266
450	0.0661190683
500	0.0807527896
550	0.0978193478
600	0.126140013
650	0.146378511
700	0.168847877
750	0.192539493
800	0.219705288
850	0.267457097
900	0.301437881
950	0.332602623
1000	0.366929358

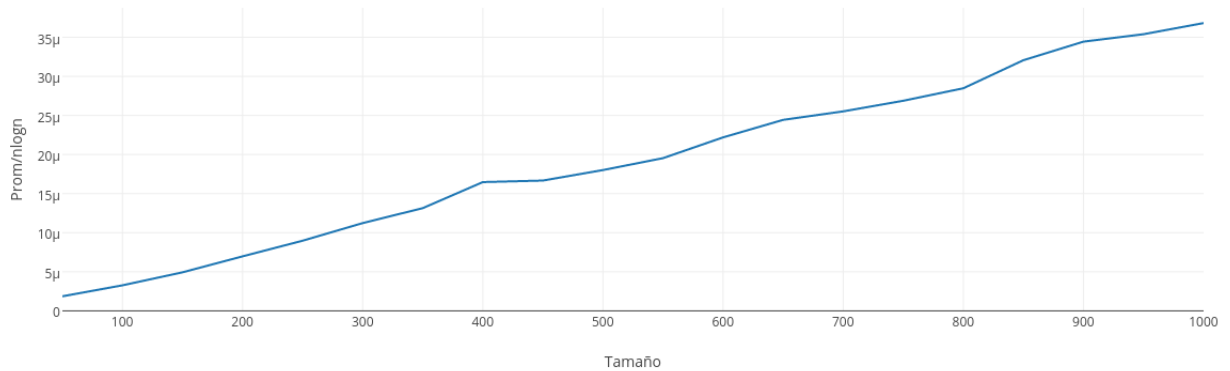


Dado que la Cota de Complejidad planteada teóricamente es de $O(n^2 \cdot \log(n))$, era esperable que la curva sea una parábola creciente.

A simple vista, no se puede apreciar si la relación que tienen respecto de tamaño/tiempo es efectivamente la que buscamos (pues casi todas las curvas polinomiales tienen gráficos similares). Por este motivo, como siguiente paso decidimos comenzar a linealizar los tiempos, dividiendo a cada uno por $\log(n)$.



La morfología de este gráfico es similar a la anterior, sigue siendo una parábola creciente, por lo tanto terminaremos de linealizar, dividiendo a cada uno por n , para ver si se trata de la parábola cuadrática.



Efectivamente puede observarse que el comportamiento es lineal. Por lo tanto, podemos afirmar que nuestra experimentación condice a la Cota Teórica planteada de $O(n^2 \cdot \log(n))$.