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

Resumen

Habiéndonos sido dado una serie de tres problemáticas a resolver, se plantean sus respectivas soluciones acorde a los requisitos pedidos. Se adjunta una descripción de cada problema y su solución, conjunto a su análisis de correctitud y de complejidad sumado a su experimentación. El lenguaje elegido para llevar a cabo el trabajo es C++.

Dentro de cada *.cpp* está el comando para compilar cada ejercicio desde la carpeta donde se encuentran los mismos. A continuación se los adjunta. El flag *-std=c++11* debió ser añadido dado que utilizamos la librería <chrono>, la cual nos permitió medir tiempos de ejecución:

1. *g++ -o main Dakkar.cpp -std=c++11*
2. *g++ -o main Zombieland.cpp -std=c++11*
3. *g++ -o main RefinandoPetroleo.cpp -std=c++11*

Índice

1. Dakkar

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.

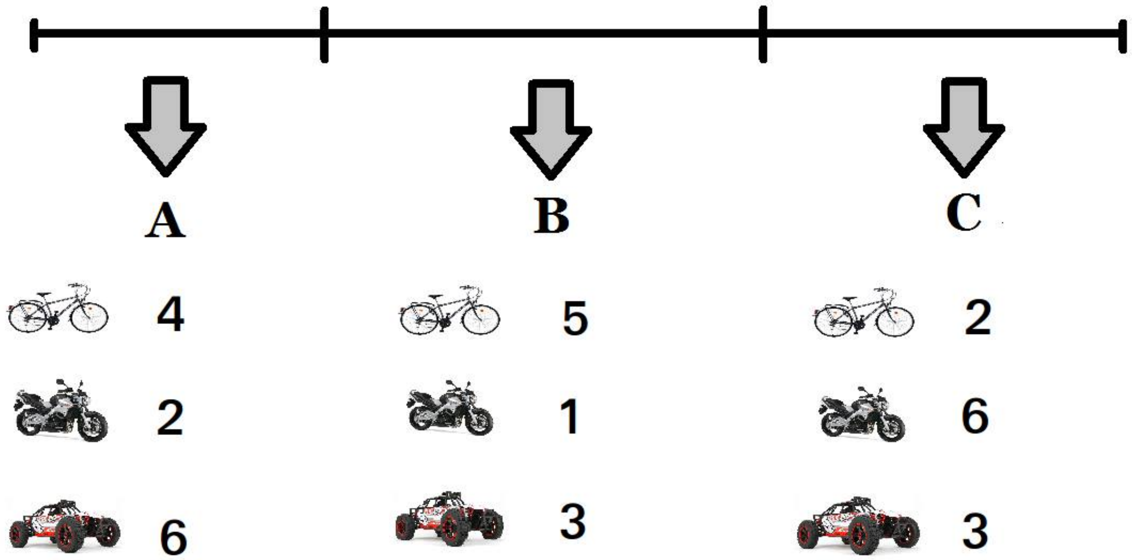
Los *tiempos* que le llevan a los vehículos recorrer el trayecto varían por cada etapa y son datos conocidos pasados por parámetro.

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 $O(n.k_m.k_b)$.

A continuación, se presenta un ejemplo sobre cómo resolver el problema, dado un caso específico.

Dada una travesía con 3 etapas (A, B y C), para las cuales se cuenta con 1 uso de la moto ($k_m = 1$) y 2 usos del buggy ($k_b = 2$), los tiempos que tarda cada vehículo por cada etapa son los siguientes:



En la siguiente tabla se muestran las diferentes elecciones que se pueden realizar, teniendo en cuenta la cantidad de usos disponibles, indicando cuál es el tiempo total en cada caso.

Casos	Vehículo utilizado por etapa			Tiempo total
	A	B	C	
Caso 1	BMX	BMX	BMX	11
Caso 2	BMX	BMX	MOTO	15
Caso 3	BMX	BMX	BUGGY	12
Caso 4	**BMX**	**MOTO**	**BMX**	**7**
Caso 5	BMX	MOTO	BUGGY	8
Caso 6	BMX	BUGGY	BMX	9
Caso 7	BMX	BUGGY	MOTO	13
Caso 8	BMX	BUGGY	BUGGY	10
Caso 9	MOTO	BMX	BMX	9
Caso 10	MOTO	BMX	BUGGY	10
Caso 11	**MOTO**	**BUGGY**	**BMX**	**7**
Caso 12	MOTO	BUGGY	BUGGY	8
Caso 13	BUGGY	BMX	BMX	13
Caso 14	BUGGY	BMX	MOTO	17
Caso 15	BUGGY	BMX	BUGGY	14
Caso 16	BUGGY	MOTO	BMX	9
Caso 17	BUGGY	MOTO	BUGGY	10
Caso 18	BUGGY	BUGGY	BMX	11
Caso 19	BUGGY	BUGGY	MOTO	15

En este ejemplo, la solución buscada, es decir, la óptima, puede ser tanto el caso 4 como el 11, ya que ambos toman el mismo tiempo y respetan la cantidad de usos de buggy y moto posibles. Por la formulación de nuestro algoritmo, nosotros devolvemos el caso 11.

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 buggys 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 buggys 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 buggy. Por este motivo se elige la opción con tiempo menor usando Bicicleta o buggy 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 Buggy llamando recursivamente a la función para *n-1* de modo que va a devolver el menor tiempo posible considerando la elección llevada a cabo.

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

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:

$$
\begin{array}{ccccc}
0 & \leq & n & \leq & n_{parametro} \\
0 & \leq & k_m & \leq & k_{m_{parametro}} \\
0 & \leq & k_b & \leq & k_{b_{parametro}}
\end{array}
$$

Teorema

La formulación recursiva planteada resuelve el problema.

Demostración

Para demostrar que el algoritmo resuelve, explotaremos la definición recursiva para aplicar inducción en la cantidad de etapas.
Entonces, queremos demostrar $P(n) = (\forall k_m, k_b) func(n, k_m, k_b)\ obtiene\ la\ longitud\ del\ camino\ minimo$.

Caso base:

$P(0) = 0$
Vale, puesto que si no hay etapas, el camino mínimo tiene longitud 0, independientemente de la cantidad de buggys o motos de los que se disponga.

Paso inductivo:

$P(n) \Rightarrow P(n+1)$
Primero vamos a mencionar que este problema cumple con el principio de optimalidad, es decir que, desde la etapa 0 hasta la i, la subsolución hallada, será óptima.

Esto se puede ver en la tabla de anterior:

En el caso 4 y 11, si acotamos el problema a 2 etapas, ambas soluciones nos indican que el tiempo total sería 5, que es óptimo.

Hasta la *n-ésima* etapa, sabemos que contamos con el tiempo mínimo para cada decisión posible en las n-1 etapas anteriores. Entonces por el principio de optimalidad, podemos agregar el tiempo mínimo de los posibles vehículos, obteniendo una elección óptima de *n+1* etapas.

Por lo tanto, vale $P(n) \Rightarrow P(n+1)$. □

Resulta pertinenete destacar que el algoritmo resuelve de la misma manera que el *Algoritmo de Bellman-Ford para caminos mínimos*, donde cada camino corresponde a las distintas decisiones posibles que se puedan tomar.

Diccionario a utilizar

El diccionario que vamos a utilizar consiste en una matriz de $k_{m_{inicial}}$x$k_{b_{inicial}}$, en la que por cada posición va a haber un contenedor de tamaño $n_{inicial}$ (lo cual forma un cubo) de modo que dentro de cada uno de ellos se va a poder almacenar el resultado de invocar a la función con estos tres parámetros.

Formulación Top Down

Si analizamos el comportamiento de nuestra función recursiva como si fuera un algoritmo recursivo, podemos notar que la primer posición del cubo que va a poder completar va a ser la de $(k_{m_{inicial}}, k_{b_{inicial}}, 0)$, dado que para todas las guardas el primer caso que compara es usar la bicicleta. Explicar con un poco mas de detalle.

Luego, seguirá completando la matriz variando en orden ascendente el tercer parámetro y completando para todos los k_m y k_b.

Formulación Bottom Up

Una vez comprendido el comportamiento de la función, podemos establecer una manera de completar nuestro diccionario cubo de modo iterativo.

Primero completamos para $n = 0$ todos los valores de k_m y k_b. Notar que dados los índices que utilizamos a la hora de la implementación, cuando hablamos de $n = 0$ nos referimos a la etapa 1 de la carrera (la última que vamos a recorrer), lo cual no es estrictamente el mismo resultado que el expuesto en la formulación recursiva. Sin embargo, podemos denotar esto como una diferencia que no afecta al funcionamiento del algoritmo.

Cuando $n = 0$ simplemente vamos a asignar el vehículo que presente menor tiempo para esta etapa, entre los disponibles basándonos en los k_m y k_b.

Luego para $n = 1$, debemos considerar el costo mínimo entre usar cada uno de los vehículos disponibles para esta etapa sumado a su correspondiente costo mínimo de la etapa anterior.

Continuamos con las iteraciones hasta llegar a $n = n_{parametro}$.

Habra que poner un dibujito que ejemplifique aca?

Cuando contamos con la matriz Diccionario completa, nuestro resultado va a estar ubicado en la posición $(k_{m_{parametro}}, k_{b_{parametro}}, n_{parametro})$.

Al ingresar los datos dentro del diccionario de esta manera, lo único que hacemos es invertir el orden de llenado. Es por este motivo que estamos cumpliendo con el mismo comportamiento de la función recursiva ya mencionada. Como ya pudimos asegurar que la formulación recursiva resolvía de manera correcta la problemática planteada, estamos en condiciones de afirmar que completar el diccionario en el orden inverso también resuelve el ejercicio.

Con el fin de poder calcular el camino una vez obtenida la cantidad de tiempo menor para realizar el trayecto, en cada posición del diccionario no sólo guardamos el tiempo empleado para llegar hasta ahí sino que también cuál fue su situación inmediatamente anterior.

De este modo, recorremos la matriz desde la posición $(k_{m_{parametro}}, k_{b_{parametro}}, n_{parametro})$ hacia la situación que indica la misma, y así sucesivamente. De este modo obtenemos las elecciones hechas desde la última etapa hasta la primera.

PseudoCódigo

input : int n, k_m, k_b , lista<BMX, Moto, Buggy> $datosPorEtapas$

for $etapa \leftarrow 0$ **to** n **do**
  for $cmoto \leftarrow 0$ **to** k_m **do**
    for $cbuggy \leftarrow 0$ **to** k_b **do**
      $dicc(cmoto, cbuggy, etapa) \leftarrow$*Elijo el mínimo entre los vehículos disponibles para esta etapa sumado a lo que tarde en la anterior considerando esta elección*;
      $dicc(cmoto, cbuggy, etapa) \leftarrow$*Elección tomada de vehículo*;
    end
  end
end

Recorrer el diccionario desde la posición final para armar el recorrido;

Algorithm 1: Dakkar

1.3. Análisis de la complejidad

1.3.1. Complejidad Temporal

Como parámetro vamos a recibir n, k_m, k_b y los tiempos necesarios para atravesar con cada vehículo las *n* etapas. Los tiempos los vamos a tener almacenados en un vector de tuplas, donde cada componente va a ser BMX, Moto y buggy.

Mediante tres `fors` anidados (primero por la cantidad de etapas, después por la cantidad de motos y por último la cantidad de buggys), vamos a recorrer posición a posición nuestra matriz de vectores (cubo). Comenzando en la posición $(0, 0, 0)$ y finalizando en (k_m, k_b, n).

Dentro de cada iteración, lo que vamos a hacer es completar la posición del diccionario correspondiente acorde indica el planteo recursivo, con la salvedad de que cada casillero representa cuánto nos cuesta llegar a la siguiente etapa, es decir, que la posición (k_m, k_b, n) tendrá lo que cueste ir de la etapa n a la etapa n+1, usando la cantidad de Motos y Buggys correspondiente.

En todas las iteraciones, escribiremos en el diccionario bajo la clave: (k_m, k_b, n). En cada posición vamos a poner un par: la primera posición corresponde al valor del tiempo y la segunda al indicador de la posición inmediatamente anterior.

Para escribir la *primer posición* vamos a seguir las siguientes reglas:

- Si $n = 0 \wedge k_m = 0 \wedge k_b = 0$ escribimos el tiempo que le lleva a la BMX para ir de 0 a 1
- Si $n = 0 \wedge k_m = 0 \wedge k_b \neq 0$ escribimos el tiempo que sea menor para ir de 0 a 1 entre BMX y Buggy
- Si $n = 0 \wedge k_m \neq 0 \wedge k_b = 0$ escribimos el tiempo que sea menor para ir de 0 a 1 entre BMX y Moto
- Si $n = 0 \wedge k_m \neq 0 \wedge k_b \neq 0$ escribimos el tiempo que sea menor para ir de 0 a 1 entre BMX, Buggy y Moto
- Si $n \neq 0 \wedge k_m = 0 \wedge k_b = 0$ escribimos el tiempo que le lleva a la BMX para ir de n a $n + 1$ + $dicc(m, b, n - 1)$
- Si $n \neq 0 \wedge k_m = 0 \wedge k_b \neq 0$ escribimos el tiempo que sea menor para ir de n a $n + 1$ entre $tiempo(BMX) + dicc(m, b, n - 1)$ y $tiempo(Buggy) + dicc(m, b - 1, n - 1)$
- Si $n \neq 0 \wedge k_m \neq 0 \wedge k_b = 0$ escribimos el tiempo que sea menor para ir de n a $n + 1$ entre $tiempo(BMX) + dicc(m, b, n - 1)$ y $tiempo(Moto) + dicc(m - 1, b, n - 1)$
- Si $n \neq 0 \wedge k_m \neq 0 \wedge k_b \neq 0$ escribimos el tiempo que sea menor para ir de n a $n + 1$ entre $tiempo(BMX) + dicc(m, b, n-1)$, $tiempo(Buggy) + dicc(m, b-1, n-1)$ y $tiempo(Moto) + dicc(m-1, b, n-1)$

Debido a la forma de completar el diccionario, todas estas asignaciones son operaciones con un costo de $\mathbf{O(1)}$.

Esto se debe a que acceder al vector con todos los costos por etapa, sumar y buscar el mínimo entre enteros (min() [1]) son operaciones de costo: $O(1)$. Además, al buscar en el diccionario siempre lo hacemos en posiciones ya definidas lo cual también presenta un costo de $O(1)$ debido a que es una matriz de vectores.

[1]http://www.cplusplus.com/reference/algorithm/min/

Para la *segunda posición*, que representa la escritura para devolver el camino, lo que realizamos es dejar asentado cuántos Buggys y cuántas Motos quedan disponibles después de esta iteración.

El costo de estas operaciones también pertenece a $\mathbf{O(1)}$. Las mismas son operaciones elementales (asignación y suma de enteros) y un llamado a make_pair() [2] que toma tiempo constante.

Finalmente retornamos el valor de $dicc(k_{m_{parametro}}, k_{b_{parametro}}, n_{parametro})$

La complejidad resulta entonces: por cada etapa, por cada posible uso de moto, por cada posible uso de buggy, realizar operaciones en tiempo constante. De esto deducimos que la complejidad es de $\mathbf{O(k_m * k_b * n)}$, como se pedía en el enunciado.

1.3.2. Complejidad Espacial

Dado que la estructura utilizada es una matriz (`vector<vector>`) de m_kxm_b, donde en cada posición contamos con un arreglo de n posiciones y una tupla ($O(1)$); su complejidad espacial es de $\mathbf{O(k_m * k_b * n)}$.

[2] http://www.cplusplus.com/reference/utility/make_pair/

1.4. Código fuente

```
struct datosPorEtapas{
    unsigned int bmx;
    unsigned int moto;
    unsigned int buggy;
};
```

```
int main(int argc, char const *argv[]){
    unsigned int etapas, cmoto, cbuggy;
    cin >> etapas >> cmoto >> cbuggy;
    deque<datosPorEtapas> datos;
    for (int i = 0; i < etapas; ++i){
        unsigned int bmx, moto, buggy;
        datosPorEtapas actual;
        cin >> actual.bmx >> actual.moto >> actual.buggy;
        datos.push_back(actual);
    }
    Matriz cubo;
//iniciliazamos el cubo
    for (int i = 0; i <= cmoto; ++i){
        Filas fila;
        for (int j = 0; j <= cbuggy; ++j){
            Etapas etapa;
            for (int k = 0; k < etapas; ++k){
                etapa.push_back(make_pair(0, make_pair(0, 0)));
            }
            fila.push_back(etapa);
        }
        cubo.push_back(fila);
    }
//cout pedido
    cout << dakkar(etapas, cmoto, cbuggy, datos, cubo);
//para devolver del comienzo hasta el final
    deque<int> usados;
    pair<int, int> recorrido;
    int cantMoto = cmoto, cantBuggy = cbuggy;
    for (int cantE = etapas-1; cantE >= 0; --cantE) {
        recorrido = cubo[cantMoto][cantBuggy][cantE].second;
        if(cantMoto > recorrido.first){
            cantMoto--;
            usados.push_front(2);
        }
        else if(cantBuggy > recorrido.second){
            cantBuggy--;
            usados.push_front(3);
        }
        else{
            usados.push_front(1);
        }
    }
    for (int i = 0; i < etapas; ++i) {
        cout << " " << usados[i];
    }
    cout << endl;
    return 0;}
```

```
unsigned int dakkar(unsigned int etapas, unsigned int cmoto, unsigned int cbuggy,
 deque<datosPorEtapas>& datos, Matriz& cubo){
    int bici, moto, buggy;
    for (int n = 0; n < etapas; ++n){
        for (int m = 0; m <= cmoto; ++m){
            for (int b = 0; b <= cbuggy; ++b){
            //guardamos un link hacia el lugar de donde salimos
                pair<int, int> par;
                par.first = m;
                par.second = b;
            //llenamos espacios triviales y luego sabemos cuales estan calculados,
            // para ahorrar rehacer las cuentas
                if (n==0){
                    if(m == 0){
                        if(b == 0){
                            bici = datos[n].bmx;
                            cubo[m][b][n] = make_pair(bici, par);
                        }
                        else{
                            bici = datos[n].bmx;
                            buggy = datos[n].buggy;
                        //si elegimos el buggy guardamos esa decision
                            if(bici > buggy)
                                par.second--;
                            cubo[m][b][n] = make_pair(min(bici, buggy), par);
                        }
                    }
                    else{
                        if(b == 0){
                            bici = datos[n].bmx;
                            moto = datos[n].moto;
                        //si elegimos la moto guardamos esa decision
                            if(bici > moto)
                                par.first--;
                            cubo[m][b][n] = make_pair(min(bici, moto), par);
                        }
                        else{
                            bici = datos[n].bmx;
                            moto = datos[n].moto;
                            buggy = datos[n].buggy;
                        //guardamos lo que elegimos
                            if(bici > buggy || bici > moto)
                                if(buggy > moto)
                                    par.first--;
                                else
                                    par.second--;
                            cubo[m][b][n] = make_pair(min(min(bici, moto), buggy), par);
                        }
                    }
                }
```

```
                else{
                    if(m == 0){
                        if(b == 0){
                            bici = cubo[m][b][n-1].first + datos[n].bmx;
                            cubo[m][b][n] = make_pair(bici, par);
                        }
                        else{
                            bici = cubo[m][b][n-1].first + datos[n].bmx;
                            buggy = cubo[m][b-1][n-1].first + datos[n].buggy;
                        //si elegimos el buggy guardamos esa decision
                            if(bici > buggy)
                                par.second--;
                            cubo[m][b][n] = make_pair(min(bici, buggy), par);
                        }
                    }
                    else{
                        if(b == 0){
                            bici = cubo[m][b][n-1].first + datos[n].bmx;
                            moto = cubo[m-1][b][n-1].first + datos[n].moto;
                        //si elegimos la moto guardamos esa decision
                            if(bici > moto)
                                par.first--;
                            cubo[m][b][n] = make_pair(min(bici, moto), par);
                        }
                        else{
                            bici = cubo[m][b][n-1].first + datos[n].bmx;
                            moto = cubo[m-1][b][n-1].first + datos[n].moto;
                            buggy = cubo[m][b-1][n-1].first + datos[n].buggy;
                        //guardamos lo que elegimos
                            if(bici > buggy || bici > moto)
                                if(buggy > moto)
                                    par.first--;
                                else
                                    par.second--;
                            cubo[m][b][n] = make_pair(min(min(bici, moto), buggy), par);
                        }
                    }
                }
            }
        }
    }
    return cubo[cmoto][cbuggy][etapas-1].first;
}
```

1.5. Experimentación

1.5.1. Constrastación Empírica de la complejidad

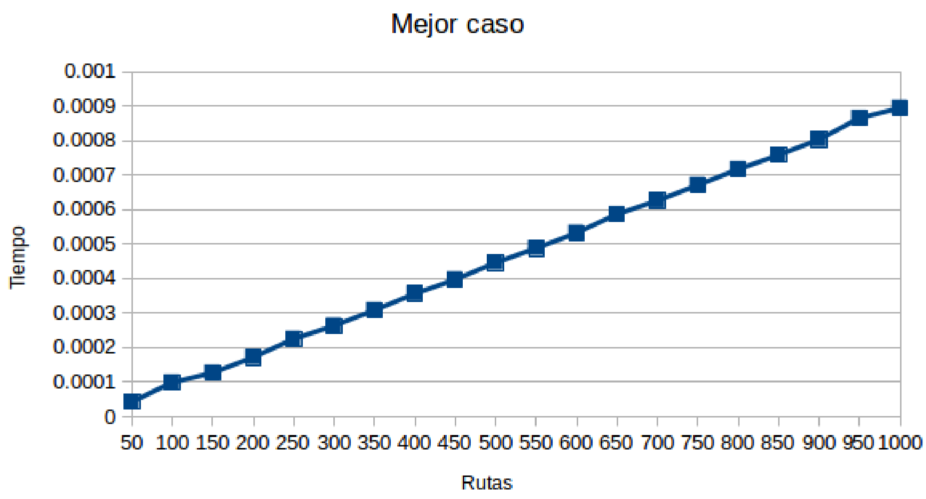
Para realizar la experimentación, generamos una serie de tests en los cuales fuimos aumentando progresivamente la cantidad de etapas (n). Los tiempos que le toma a cada vehículo recorrer cada etapa fueron elegidos aleatoriamente, con números entre 1 y 10000, ya que no influyen en la complejidad del algoritmo. Esto se debe a que las operaciones entre ellos son de comparación, es decir que tienen una complejidad de $O(1)$.

Luego, comenzamos mirando el caso más sencillo (*mejor caso*), que se da cuando tanto k_m como k_b son igual a 0. Es decir, en todos los casos debemos usar la BMX.

La complejidad teórica planteada es de $O(n * k_m * k_b)$, por lo que se intuye que en este caso debería ser $O(n)$.

Los tiempos de ejecución para cada etapa n fueron:

n	Tiempo en segundos
50	0.0000428063
100	0.0000990471
150	0.0001270313
200	0.0001718291
250	0.0002251666
300	0.0002632707
350	0.0003082768
400	0.0003563523
450	0.0003971006
500	0.0004461637
550	0.0004880608
600	0.0005325091
650	0.0005865319
700	0.0006258109
750	0.0006709416
800	0.0007168112
850	0.0007580348
900	0.000803331
950	0.0008650634
1000	0.0008941003

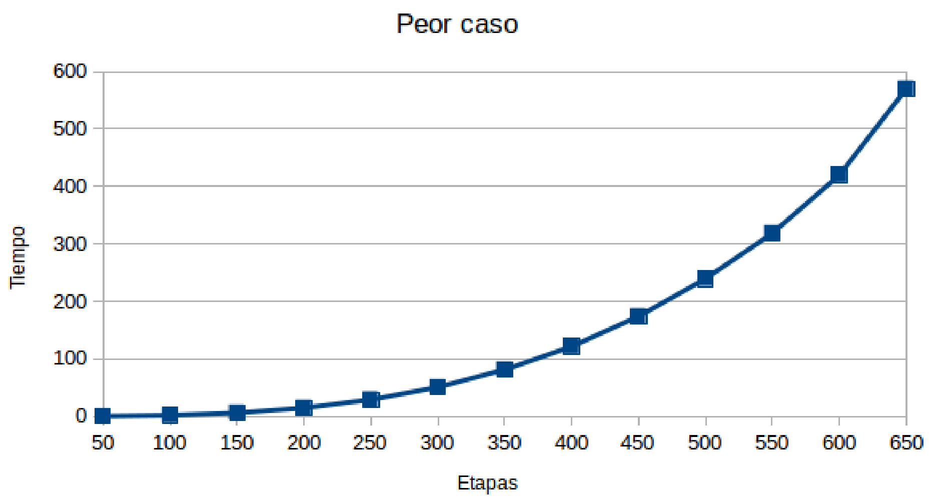


Como preveíamos, el gráfico tiene comportamiento lineal, lo cual es consistente con la cota teórica planteada.

Luego, consideramos el *peor caso* posible, que es el que iguala tanto k_m como k_b con n. Es decir, el caso en el que para cada etapa, tenemos la posiblidad de elegir cualquiera de las 3 opciones, BMX, motocross o buggy.

Los tiempos de ejecución para cada etapa n fueron:

n	Tiempo en segundos
50	0.1744401
100	1.733896
150	6.082229
200	14.68914
250	29.19128
300	50.62509
350	81.18956
400	121.5241
450	173.8635
500	239.492
550	318.615
600	420.213
650	570.048

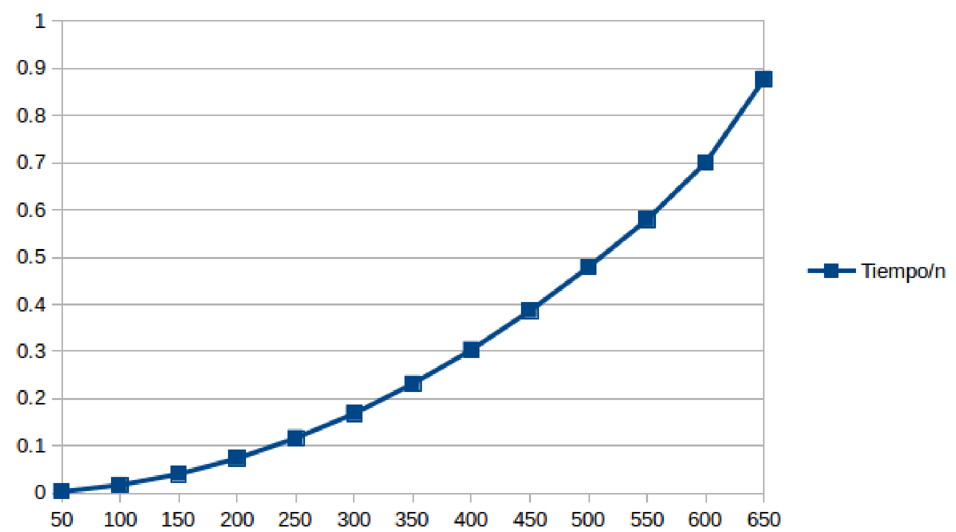


Como en este caso tanto k_m como k_b son iguales a n, la Cota de Complejidad planteada teóricamente ($O(n * k_m * k_b)$) es igual a $O(n^3)$.

En el gráfico se aprecia una parábola creciente, pero no podemos asegurar que ésta sea la que nosotros planteamos, ya que las curvas son similares a simple vista.

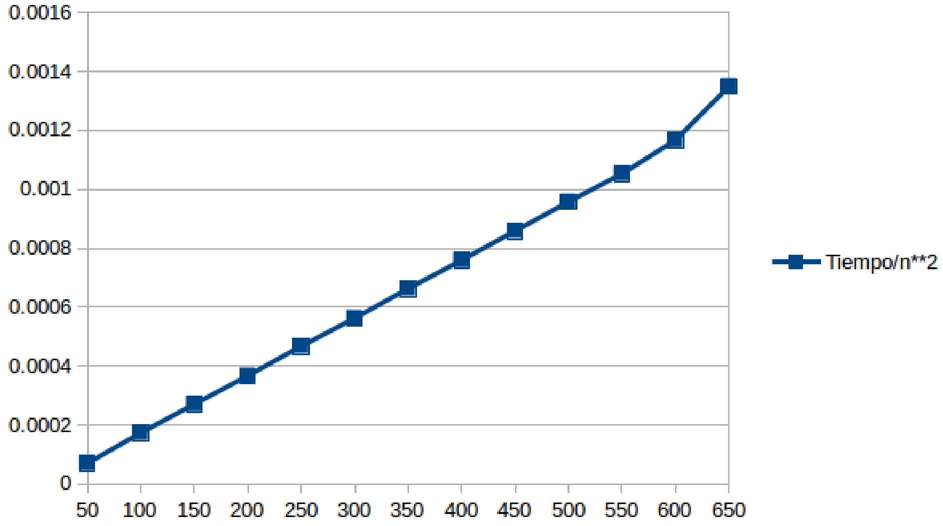
Por lo explicado anteriormente, linealizamos los tiempos, dividiendo cada instancia por n.

El gráfico resultante es el siguiente:



Este gráfico es similar al anterior, con una pendiende menos pronunciada.

De todas formas, para asegurarnos que nuestra experimentación se condice con la Cota Teórica planteada, volvemos a dividir cada instancia por n linealizando así la muestra obtenida:

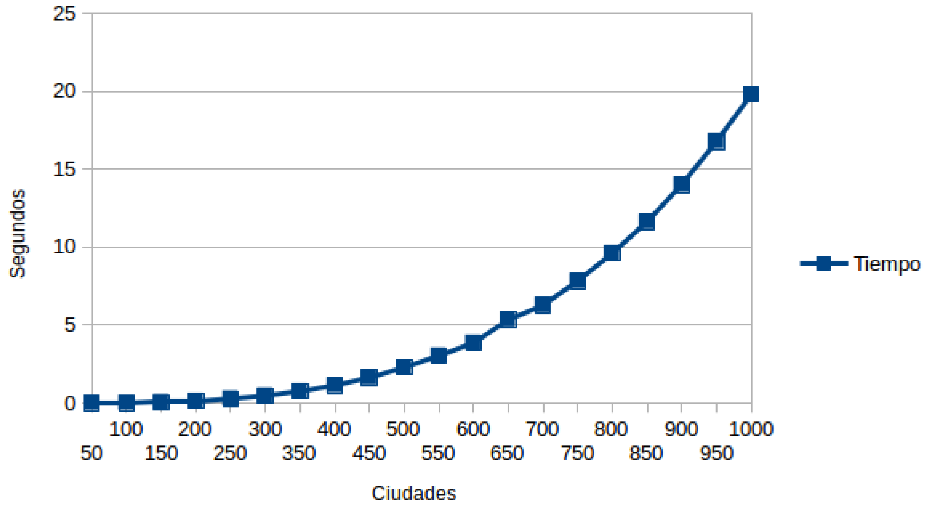


En efecto, este gráfico tiene comportamiento lineal, lo cual confirma nuestra afirmación y prueba empíricamente nuestra cota de Complejidad Temporal Teórica.

Luego, con el fin de comparar los tiempos de ejecución, se realizó una última experimentación, en la cual se tomaron los mismos valores para k_m y k_b, proporcionales al n dado.

Como resultado se obtuvieron los siguientes valores:

n	Tiempo en segundos
50	0.0015956707
100	0.0152238567
150	0.055239755
200	0.132059613
250	0.264843302
300	0.464658156
350	0.748833425
400	1.13247454
450	1.63054066
500	2.28274084
550	3.03236404
600	3.84826384
650	5.3375904
700	6.2688524
750	7.82798085
800	9.62078432
850	11.6278678182
900	14.0280025641
950	16.79448
1000	19.80024



Como se ve, el gráfico tiene un comportamiento similar al del peor caso, pero los tiempos de ejecución son mucho menores, por lo que es evidente la influencia de k_m y k_b en la Cota Teórica planteada.

2. Zombieland II

2.1. Descripción de la problemática

El escenario con el que vamos a trabajar es una ciudad con n calles horizontales y m calles verticales. En esta ciudad se encuentran dispersos zombies, los mismos están dispuestos en las calles (tanto en las verticales como en las horizontales) pero ninguno está ubicado exactamente en alguna esquina.

Contamos con s soldados, los cuales se quieren movilizar desde una posición inicial (X_i) hasta una posición final (X_f) donde se encuentra el búnker.

Todos estos datos van a ser ingresados como parámetros de entrada.

Efectivamente, al encontrarse soldados con zombies se va a producir un enfrentamiento. Si la cantidad de soldados es mayor o igual a la cantidad de zombies, se exterminan los zombies de esa cuadra sin dejar ningún soldado muerto.

En cambio, si la cantidad de zombies es mayor a la cantidad de soldados, se va a sufrir una baja de $(z - s)$ soldados. De este modo, si $z - s \geq s$ toda la cuadrilla va a quedar eliminada impidiendo el avance con el recorrido. En caso contrario, se eliminan los zombies de esa cuadra perdiendo los $z - s$ soldados.

Esto implica el impedimento de recorrer un camino cuando $s = 0$ (incluso cuando así sea el parámetro de entrada) sin tener en cuenta la existencia o no de zombies en el trayecto restante.

Se busca hallar la manera de llegar desde X_i hasta X_f con la mayor cantidad de soldados sobrevivientes, si es posible. De haber más de un camino que cumpla estas condiciones, se debe devolver sólo uno.

Se exige una cota de complejidad temporal de $\mathbf{O}(\mathbf{s.n.m})$.

Un caso específico de la problemática a tratar es el que detallamos a continuación.

Consideremos una ciudad con $n = 3$ y $m = 3$, de modo que se forma una ciudad cuadrada de cuatro manzanas en total. La cantidad de soldados inicial(s) es 10 y se deben mover desde un extremo de la ciudad (X_i) hasta el otro (X_f) tal como indica la figura. En color verde se indica el camino óptimo que debe ser devuelto al ejecutar el algoritmo.

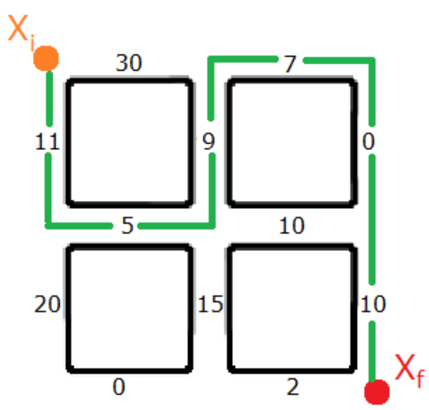


Figura 1: Ciudad de 3x3

Es claro observar que la cuadra donde la cantidad de zombies es 30 no va a pertenecer al camino solución ya que al pasar por allí con la cantidad máxima de soldados (la cantidad inicial) no habrá sobrevivientes. A continuación, el primer movimiento que pueden hacer es pasar por la cuadra con 11 zombies, sufriendo la baja de un soldado.

Luego de este paso, contamos con una cantidad de 9 soldados sobrevivientes por lo que resulta también trivial esquivar la calle con 20 zombies ya que no se podría salvar a ningún soldado de la cuadrilla. Por este motivo, se prosigue el camino por la cuadra con 5 zombies, sin sufrir ninguna baja.

Al llegar a este momento, mantenemos la cantidad de soldados sobrevivientes en 9. Si bien un camino más corto en longitud sería pasar por la cuadra con 10 zombies, no es el camino solución ya que al enfrentarse 9 soldados con 10 zombies se sufre una baja más dejando al ejército con 8 soldados. En cambio, se pasa por las cuadras de 9, 7 y 0 zombies sin perder ningun hombre.

Al finalizar, sólo queda una cuadra posible que contiene a 10 zombies también. Luego de cruzar esta cuadra, la cantidad de soldados será 8 y llegaremos a destino.

2.2. Resolución propuesta y justificación

Para modelar la problemática planteada, utilizamos dos grafos.

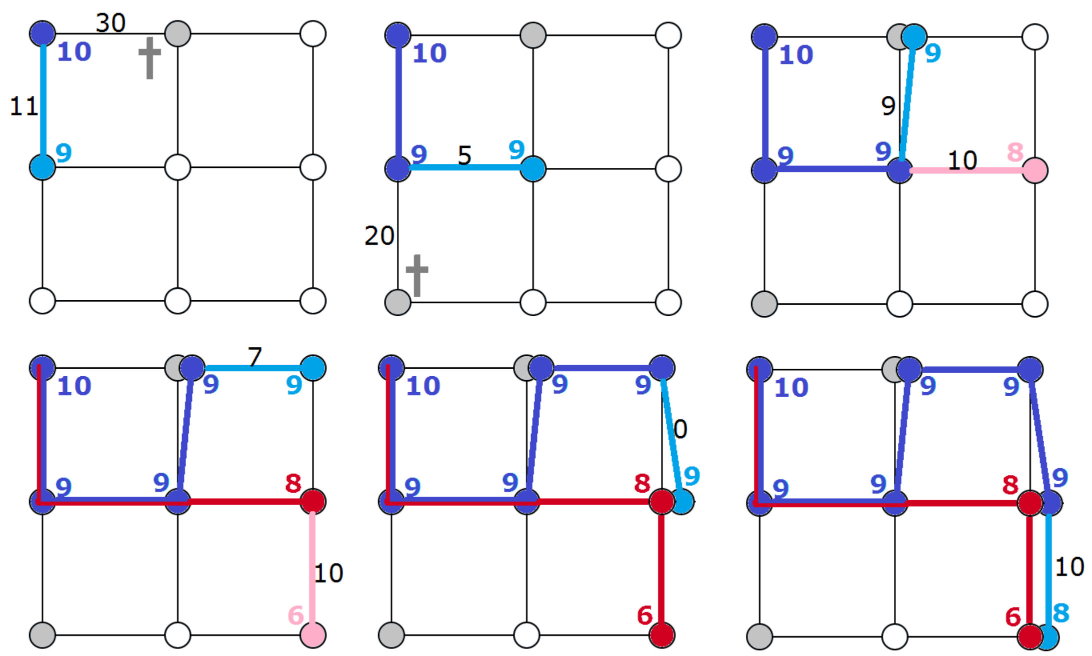
El *primer grafo* (no dirigido) con el que vamos a tratar es el que almacena los datos de entrada -`ciudadInfestada`- (cantidad de zombies por cuadra), de modo que cada nodo es una esquina de la ciudad y cada eje tiene en su peso la cantidad de zombies que están presentes en esa locación.

El *segundo grafo* (dirigido) -`grafo`- lo vamos a ir construyendo, de modo que al finalizarlo tengamos calculada la solución.

Cada nodo de `grafo` va a contener la siguiente información: cantidad de soldados vivos con la que se llegó a esa posición (`soldadosVivos`) y las coordenadas de su posición: `i` y `j`.

Los ejes de nuestro `grafo` no van a contener ninguna información extra, sólo indicarán los movimientos posibles desde un nodo a otro. Es decir, sólo va a haber una conexión del nodo *e* al nodo *f* si en ese trayecto no se produjo la baja total de los soldados.

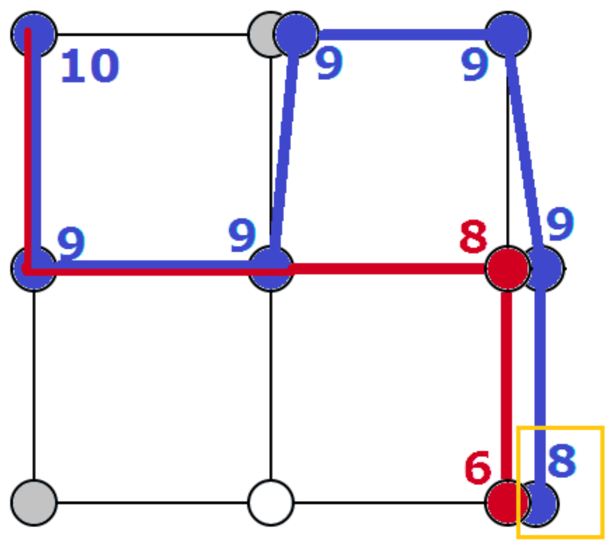
Por ejemplo, si consideramos el caso visto en la Sección 1, vamos a construir un grafo de la siguiente manera:



Donde los números en negro que figuran sobre los ejes es la cantidad de zombies por esa cuadra, consultada en `ciudadInfestada`. Los nodos que permanecieron grises, o ni fueron consultados, no van a formar parte de nuestro grafo.

Así mismo, se puede apreciar que para una misma esquina hay dos nodos distintos, ya que cambia la cantidad de soldados con la que se arribó.

En este caso, encontramos dos caminos para llegar al búnker con vida. De todos modos, queremos el que lo haga con mayor cantidad de sobrevivientes, que es el camino azul.



Lo primero que vamos a hacer es situarnos en la posición inicial ($X_i = (inicioH, inicioV)$) y crear este nodo en `grafo`, de modo que `soldadosVivos` va a ser la cantidad de soldados inicial y las coordenadas van a ser `i=inicioH,j=inicioV`.

Para continuar la creación de este `grafo`, lo vamos a recorrer mediante BFS. Es decir, vamos a conectar al nodo inicial con sus cuatro nodos vecinos (si los tiene) e indicar en ellos con cuántos soldados se arriba (en caso de ser menor o igual a 0, ese nodo no se agrega) y desde qué posición. De la misma manera, vamos a repetir el proceso para todos los nodos que sean agregados. De este modo, armamos todos los caminos posibles.

Al momento de avanzar de nodo, nos fijamos en el grafo `ciudadInfestada` la cantidad de zombies en esa cuadra.

Cabe destacar que podría formarse un ciclo, para evitar esto, marcamos a los nodos ya visitados para no volverlos a agregar. Además no nos interesa un camíno que repita cuadras. Marcar los nodos visitados salva el caso en que podría pasar por cuadras repetidas con más de 0 soldados, en caso contrario, se alcanzan los 0 soldados y decimos que esa no es una solución posible.

Para realizar este procedimiento, utilizamos una `cola` la cual nos va a indicar qué nodos nos falta recorrer.

Comenzamos con el primer nodo, lo encolamos. Al desencolarlo, lo marcamos como visitado y no sólo agregamos los nodos vecinos correspondientes al `grafo`, sino que los insertamos en la cola. A continuación, desencolamos el elemento siguiente, lo marcamos como visitado y agregamos los nodos vecinos correspondientes que no hayan sido visitados.

Repetimos este proceso hasta que la cola quede vacía. Es importante destacar que no es lo mismo visitar al nodo de la posición `(x,y)` con `soldadosVivos=`s_1 que el de la misma posición con `soldadosVivos=`s_2 (con $s_1 \neq s_2$). En cambio, si $s_1 = s_2$, ya teníamos un camino que llegaba con la misma cantidad de soldados, por lo tanto, este lo desestimamos.

Una vez completo todo el `grafo`, basta verificar que exista al menos un nodo en la posición $X_f = (bunkerH, bunkerV)$ para establecer que existe un camino donde sobrevive algún soldado.

Para encontrar la cantidad de soldados máxima con la que se puede llegar a destino, lo que debemos hacer es recorrer todos los nodos de la posicón X_f y localizar al que tenga la cantidad de `SoldadosVivos` mayor. Esta va a ser la solución óptima.

Por último, para hallar el camino debemos hacer el proceso inverso. Como en cada nodo existente de `grafo` vamos a tener la coordenada anterior en la que se estaba para llegar allí, vamos accediendo secuencialmente. De esta manera, tenemos todas las posiciones (esquinas) por donde pasó la guardilla hasta llegar a destino.

Esta solución propuesta resuelve el problema porque considera todos los caminos posibles desde el punto inicial X_i. Es decir, recorre el grafo situándose en un primer momento en este nodo por todos los caminos existentes. Sólo se detiene cuando la cantidad de zombies es suficiente para destruir a todo el escuadrón. Sin embargo, todos los recorridos deben finalizar cuando se alcanza la posición del bunker X_f.

De este modo, diagramamos todos los caminos posibles que conducen desde X_i hasta X_f dejando (al menos) un soldado con vida. Los caminos que se cortan antes de alcanzar la posición del búnker no nos son de interés.

PseudoCódigo

```
input : int n, m, s, Ih, Iv, Bh, Bv, grafo ciudadInfestada

cola.push(Ih, Iv, s);
while !cola.vacia() do
    nodoActual ← cola.pop();
    nodoActual.marcar();
    for nodoVecino in nodoActual.vecinos() do
        if Cantidad de Zombies en grafo desde nodoActual hasta nodoVecino no mata a todos los
        nodoActual.soldadosVivos then
            nodoVecino.previo ← nodoActual;
            cola.push(nodoVecino);
        end
    end
    nodoActual ← nodoFinal;
end
return cantidad máxima de soldadosVivos para la posición de nodoFinal;
```

Algorithm 2: Zombieland II

2.3. Análisis de la complejidad

Los valores pasados por parámetro están almacenados en el grafo `ciudadInfestada`, el cual está representado mediante una matriz. Cada posición de la matriz es una esquina de la ciudad y en la misma se indican la cantidad de zombies que hay en el trayecto hacia la esquina de la derecha y la cantidad que hay en el inferior. Esta información es suficiente, porque si se desea saber la cantidad de zombies existentes hacia la izquierda, basta con posicionarse en el valor de la matriz de esa posición y mirar hacia la derecha. En el caso de los bordes, como no poseen cuadras a su derecha y/o hacia abajo, se les asignó el valor de -1.

Armamos una función (`zombiesCuadra`) la cual nos devuelve en $O(1)$ la cantidad de zombies que hay desde una esquina hacia alguna de sus cuatro direcciones. La complejidad Temporal de esta función es constante porque sólo basta con acceder al índice de la matriz correspondiente y buscar la posición de la tupla deseada.

Para almacenar el grafo que va a contener la solución (`grafo`), utilizamos una matriz de vectores (`cubo`). Esta matriz tiene un tamanño de $O(s.n.m)$ ya que para cada coordenada pueden existir s cantidad de `soldadosVivos` distintas para llegar hasta allí.

Si bien, no necesariamente vayamos a completar la matriz completa, debemos considerar el peor caso que es cuando esto sí sucede.

Por lo detallado anteriormente, nuestro `grafo` tiene, a lo sumo, $s.n.m$ nodos. Sabiendo esto, podemos justificar que la guarda del while va a ejecutar $\mathbf{O(s.n.m)}$ ya que cada nodo visitado se va a marcar y no se va a recorrer dos veces el mismo nodo.

La complejidad del ciclo del While podemos afirmar que es $\mathbf{O(1)}$. Las operaciones que se ejecutan son: de acceso a una matriz (ya sea `ciudadInfestada` o `grafo`) asignando o leyendo, en ambos casos con complejidad $O(1)$; operaciones elementales ($O(1)$) y operaciones de cola en un queue (push() [3], pop() [4] y front() [5]) las cuales poseen todas complejidad constante.

Por lo tanto, la complejidad Temporal del algoritmo es de $O(s.n.m) * O(1)$, lo que por propiedas de O equivale a $\mathbf{O(s.n.m)}$ tal que concuerda con la complejidad exigida.

[3] http://www.cplusplus.com/reference/queue/queue/push/
[4] http://www.cplusplus.com/reference/queue/queue/pop/
[5] http://www.cplusplus.com/reference/queue/queue/front/

2.4. Código fuente

```
struct posYsold {
    int soldadosVivos;
    int i;
    int j;
};
```

```
Matriz ciudadInfestada;
unsigned int n, m;

int main(int argc, char const *argv[]){
    unsigned int s;
    cin >> n >> m >> s;
    unsigned int inicioH, inicioV, bunkerH, bunkerV;
    cin >> inicioH >> inicioV >> bunkerH >> bunkerV;
    inicioV--;
    inicioH--;
    bunkerV--;
    bunkerH--;
//la matriz de la ciudad guarda cuantos zombies tiene el eje para moverse
// a la derecha y hacia abajo (en ese orden)
//para la izquierda es ir al de la izquierda y preguntar por el derecho
//para arriba es ir al de arriba y preguntar por el de abajo
    ciudadInfestada = Matriz(n, vector<pair<int, int> >(m));
    for (int i = 0; i < n-1; ++i) {
        for (int j = 0; j < m-1; ++j) {
            cin >> ciudadInfestada[i][j].first;
        }
//no hay camino a la derecha
        ciudadInfestada[i][m-1].first = -1;
        for (int j = 0; j < m; ++j) {
            cin >> ciudadInfestada[i][j].second;
        }
    }
    for (int j = 0; j < m-1; ++j) {
        cin >> ciudadInfestada[n-1][j].first;
//no hay camino hacia abajo
        ciudadInfestada[n-1][j].second = -1;
    }
//no hay camino a la derecha ni abajo
    ciudadInfestada[n-1][m-1].first = -1;
    ciudadInfestada[n-1][m-1].second = -1;
//creamos el cubo a completar
    Cubo grafo(n, vector<vector<pair<posYsold, bool> > >(m));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            grafo[i][j] = vector<pair<posYsold, bool> >(s+1);
        }
    }
//aplicamos el algoritmo
    int soldadosVivos = zombieland(grafo, inicioH, inicioV, bunkerH, bunkerV, s);
```

```
//cout pedido
    cout << soldadosVivos << endl;
    deque<posYsold> recorrido;
//para armarlo desde el principio hasta el final
    if(soldadosVivos != 0){
        posYsold posActual;
        posActual.soldadosVivos = soldadosVivos;
        posActual.i = bunkerH;
        posActual.j = bunkerV;
        while(posActual.i != inicioH || posActual.j != inicioV){
            recorrido.push_front(posActual);
            posActual = grafo[posActual.i][posActual.j][posActual.soldadosVivos].first;
        }
        posActual.soldadosVivos = s;
        posActual.i = inicioH;
        posActual.j = inicioV;
        recorrido.push_front(posActual);
    }
    for (int i = 0; i < recorrido.size(); ++i) {
        cout << recorrido[i].i+1 << " " << recorrido[i].j+1 << endl;
    }
    return 0;
}
```

```
int resulBatalla(int sold, int zomb){
//devuelve si es posible pasar por una cuadra
    if(sold>=zomb)
        return sold;
    return sold-(zomb-sold);
}
```

```
int zombiesCuadra(int i, int j, movimiento mov){
//devuelve los zombies que en la cuadra pedida
    switch(mov){
        case ARRIBA:
            if(i == 0)
                return -1;
            return ciudadInfestada[i-1][j].second;
        case ABAJO:
            return ciudadInfestada[i][j].second;
        case IZQ:
            if(j == 0)
                return -1;
            return ciudadInfestada[i][j-1].first;
        case DER:
            return ciudadInfestada[i][j].first;
    }
}
```

```
int zombieland(Cubo& grafo, int inicioH, int inicioV, int bunkerH, int bunkerV, int soldados){
    int maxSoldados = 0;
//cola para el BFS arranca con la posicion de donde salimos
    queue<posYsold> cola;
    posYsold actual;
    actual.soldadosVivos = soldados;
    actual.i = inicioH;
    actual.j = inicioV;
    cola.push(actual);
//se marca esta posicion como visitada
    grafo[inicioH][inicioV][soldados].second = true;
    int zombies;
    int resultadoBatalla;
//mientras tengamos algo para visitar (no sabemos si llegaremos al final)
    while(cola.size() > 0){
//actual es el primero de la cola
        actual = cola.front();
    //si no estamos el final vemos si podemos ir a cada una de las cuatro direcciones
    //si es valido, no se mueren todos los soldados y no visite ese nodo,
    //agregare el siguiente nodo a visitar
    //si ya lo visite, existe un camino que me lleva hasta ahi,
    //ya fue encolado el camino que le sigue
        if(!(actual.i == bunkerH && actual.j == bunkerV)){
            zombies = zombiesCuadra(actual.i, actual.j, ARRIBA);
            resultadoBatalla = resulBatalla(actual.soldadosVivos, zombies);
            if(zombies != -1 && resultadoBatalla > 0 && !grafo[actual.i-1][actual.j]
            [resultadoBatalla].second){
                posYsold arriba;
                arriba.soldadosVivos = resultadoBatalla;
                arriba.i = actual.i-1;
                arriba.j = actual.j;
                cola.push(arriba);
                grafo[actual.i-1][actual.j][resultadoBatalla].second = true;
                grafo[actual.i-1][actual.j][resultadoBatalla].first = actual;
            }
            zombies = zombiesCuadra(actual.i, actual.j, DER);
            resultadoBatalla = resulBatalla(actual.soldadosVivos, zombies);
            if(zombies != -1 && resultadoBatalla > 0 && !grafo[actual.i][actual.j+1]
            [resultadoBatalla].second){
                posYsold der;
                der.soldadosVivos = resultadoBatalla;
                der.i = actual.i;
                der.j = actual.j+1;
                cola.push(der);
                grafo[actual.i][actual.j+1][resultadoBatalla].second = true;
                grafo[actual.i][actual.j+1][resultadoBatalla].first = actual;
            }
```

```
            zombies = zombiesCuadra(actual.i, actual.j, ABAJO);
            resultadoBatalla = resulBatalla(actual.soldadosVivos, zombies);
            if(zombies != -1 && resultadoBatalla > 0 && !grafo[actual.i+1][actual.j]
            [resultadoBatalla].second){
                posYsold abajo;
                abajo.soldadosVivos = resultadoBatalla;
                abajo.i = actual.i+1;
                abajo.j = actual.j;
                cola.push(abajo);
                grafo[actual.i+1][actual.j][resultadoBatalla].second = true;
                grafo[actual.i+1][actual.j][resultadoBatalla].first = actual;
            }
            zombies = zombiesCuadra(actual.i, actual.j, IZQ);
            resultadoBatalla = resulBatalla(actual.soldadosVivos, zombies);
            if(zombies != -1 && resultadoBatalla > 0 && !grafo[actual.i][actual.j-1]
            [resultadoBatalla].second){
                posYsold izq;
                izq.soldadosVivos = resultadoBatalla;
                izq.i = actual.i;
                izq.j = actual.j-1;
                cola.push(izq);
                grafo[actual.i][actual.j-1][resultadoBatalla].second = true;
                grafo[actual.i][actual.j-1][resultadoBatalla].first = actual;
            }
        }
    //si llegamos al final, actualizamos los soldados que quedaron vivos,
    //si es mayor a haber llegado por otro camino
        else
            if(maxSoldados < actual.soldadosVivos)
                maxSoldados = actual.soldadosVivos;
    //desencolo el nodo que estaba analizando
        cola.pop();
    }
    return maxSoldados;
}
```

2.5. Experimentación

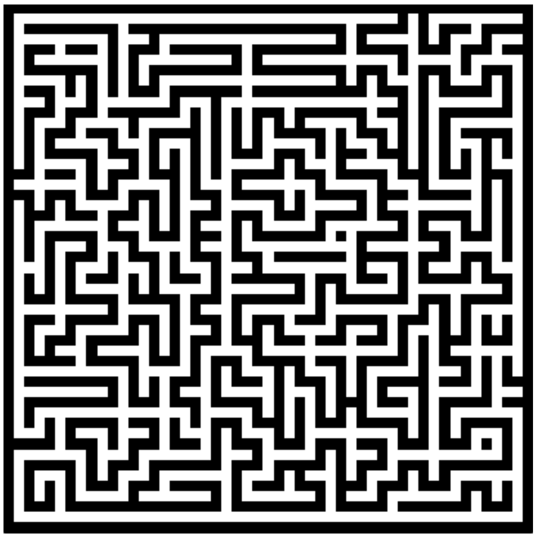
2.5.1. Constrastación Empírica de la complejidad

Al momento de llevar a cabo la experimentación, se consideró como *peor caso*, aquel en el que se debiera recorrer cada nodo de la ciudad (`grafo`), tantas veces como cantidad de soldados se tuviese a disposición en el momento inicial. Es decir, pasar por cada esquina de la ciudad s veces, cada vez con una cantidad diferente de `soldadosVivos`. Siendo ese caso, en el que se recorren $s.n.m$ elementos, coincidiendo así con la complejidad teórica $\mathbf{O}(\mathbf{s}.\mathbf{n}.\mathbf{m})$.

Aunque este sería el caso deseable, generar dicha instancia para los distintos valores de s, n y m, resultó tan dificultoso como resolver el problema en cuestión. Por este motivo, se decidió generar instancias con una cantidad aleatoria de zombies por cuadra.

Primero analizaremos con detalle estas instancias, y luego expondremos porque decidimos usarlas.

Asumiendo que la posición inicial es la de la esquina superior izquierda y la final la de la esquina inferior derecha, al ejecutar el generador de instancias aleatorias se obtiene un diagrama similar al siguiente:



Donde los pasajes del laberinto representan caminos donde hay entre 0 y $S+1$ zombies por cuadra, y las paredes, caminos donde hay entre 0 y $2.S$ zombies por cuadra.

Vale aclarar, que las paredes no necesariamente constituyen un medio por el cual los soldados no puedan pasar, sino un medio donde lograrlo es poco probable. Los pasajes, de modo similar, no necesariamente constituyen un medio por el cual los soldados siempre pueden pasar, sino un medio donde tener bajas, es poco probable.

Estos valores aleatorios fueron tomados adrede, por un lado, para que incluso el mejor camino (alguno de los pasajes que conducen a la salida), tuviera pérdida de soldados, pero intentando que éstas sean mínimas, para aumentar las probabilidades de que puedan efectivamente llegar al búnker.

Aún así, el s tomado en el algoritmo aleatorio es el que se recibe en la entrada. Esto significa, que se configura al principio, y no se actualiza con la cantidad de soldados después de una batalla.

Por ende, no podemos determinar el porcentaje de soldados totales que llegarán, pero sí podemos saber el porcentaje que sobrevivirá a la primer batalla, en el peor de los casos.

Así, la cantidad de soldados que sobrevivirán a la primer batalla, en el *peor caso*, es:

- Si toma un pasaje, en el peor de los casos, sobrevivirá el 50 % de los soldados (casos en el que hay 1 soldado y 2 zombies). No obstante, ese porcentaje tiende a 100 % a medida que la cantidad de soldados aumenta.
- Si toma una pared, solo sobrevivirá un 50 %, sin importar la cantidad de soldados.

Estos rangos aleatorios se obtuvieron experimentando con distintas distribuciones de zombies, hasta lograr valores que ocasionaran bajas pero que no impidieran a los soldados llegar al búnker en la mayoría de los casos. Dichos experimentos exceden el interés de esta sección y no serán discutidos

Al proceder bajo este método fue posible efectuar casos de prueba de gran tamaño. Sin embargo, debido a la dimensión de los casos de tests, hay ciertos aspectos de la solución del problema que no podemos determinar por nuestra cuenta para contrastar con la solución propuesta por el algoritmo:

1. No se puede determinar si existe un camino desde el punto de inicio (X_i) hasta el búnker (X_f).
2. No se puede determinar, si al existir un camino, este será único.
3. No se puede determinar cuántos soldados van a llegar al búnker.
4. No se puede determinar cuántos caminos adulteran la cantidad de soldados, ni cuántos soldados mueren en cada uno de ellos.

Finalmente, y pese a las dificultades expuestas, se decidió proceder con dichas instancias, tomándose sólo aquellas donde se pudo asegurar que existía un camino al búnker, y que existía un camino donde hubiera al menos una baja (no necesariamente, estos dos sean el mismo camino), ya que lograban generar una fluctuación en la cantidad de soldados, y esto, si bien lejos del peor caso propuesto, resultó ser una aproximación que consideramos suficiente.
Para obtenerlas, se generaron instancias hasta obtener una que tuviera dichas características.

Consecuentemente, se realizaron experimentos sobre los siguientes dos casos:

- **Cero zombies**: En esta instancia, la cantidad de zombies en cada cuadra es cero, con lo cual, el algoritmo recorre toda la ciudad, y se queda con cualquier camino.
- **Zombies aleatorios**: Es la instancia explicada anteriormente.

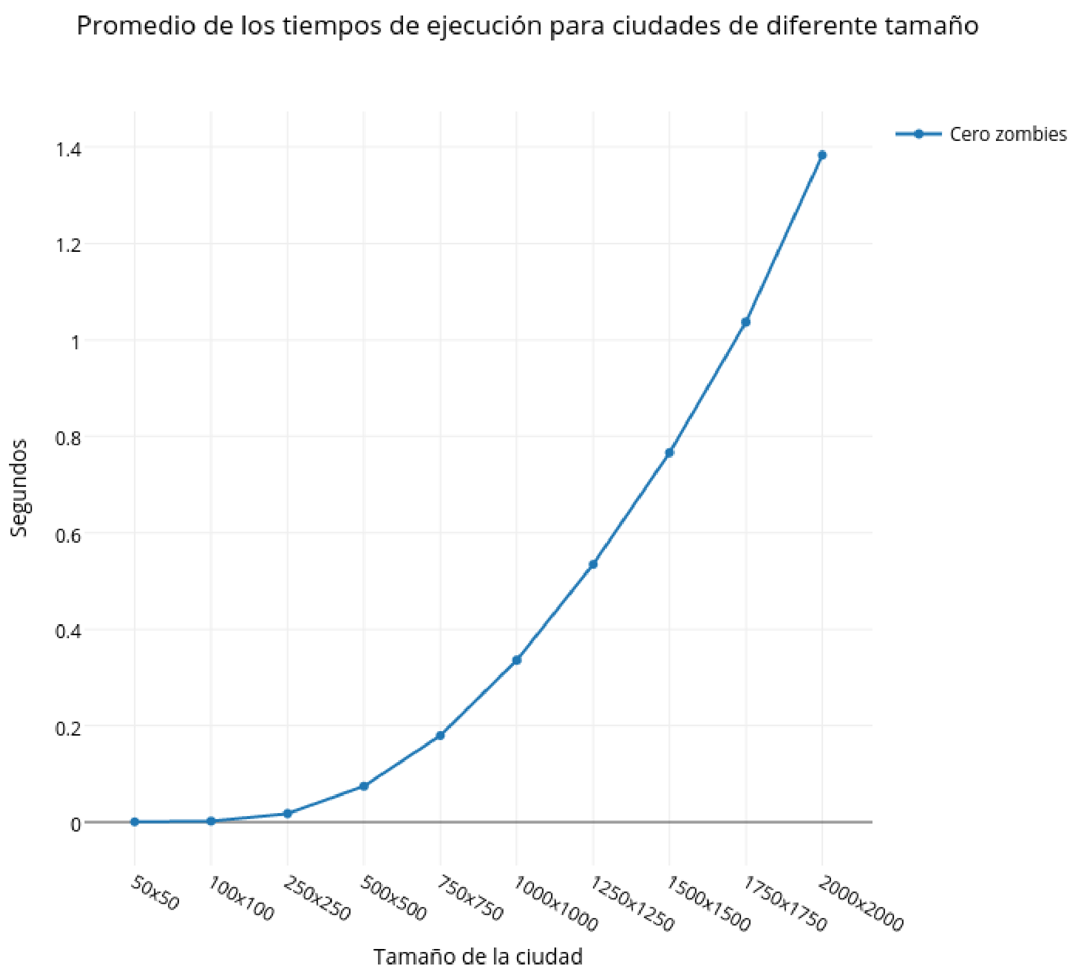
También es necesario aclarar, que los experimentos se realizaron sobre ciudades cuadradas, con puntos de inicio y llegada en los extremos opuestos de la ciudad, y con cantidad de soldados iniciales igual a 20.
Tomamos esta decisión porque los casos de ciudades cuadradas y de ciudades rectangulares, son análogos, dado que lo importante es la cantidad de cuadras en total. En lo que respecta a los puntos de partida y llegada, fueron elegidos para aprovechar al maximo la dimensión de la ciudad y obtener el más largo de los caminos posibles.

A continuación se detallan los experimentos y sus resultados. Debido al tamaño de las instancias de prueba, los inputs de dichos experimentos no fueron adjuntados.

Experimento	**Cero zombies**	**Zombies aleatorios**
Tamaño de la ciudad	Soldados al final	
50x50	20	6
100x100	20	19
250x250	20	17
500x500	20	17
750x750	20	16
1000x1000	20	12
1250x1250	20	20
1500x1500	20	7
1750x1750	20	14
2000x2000	20	20

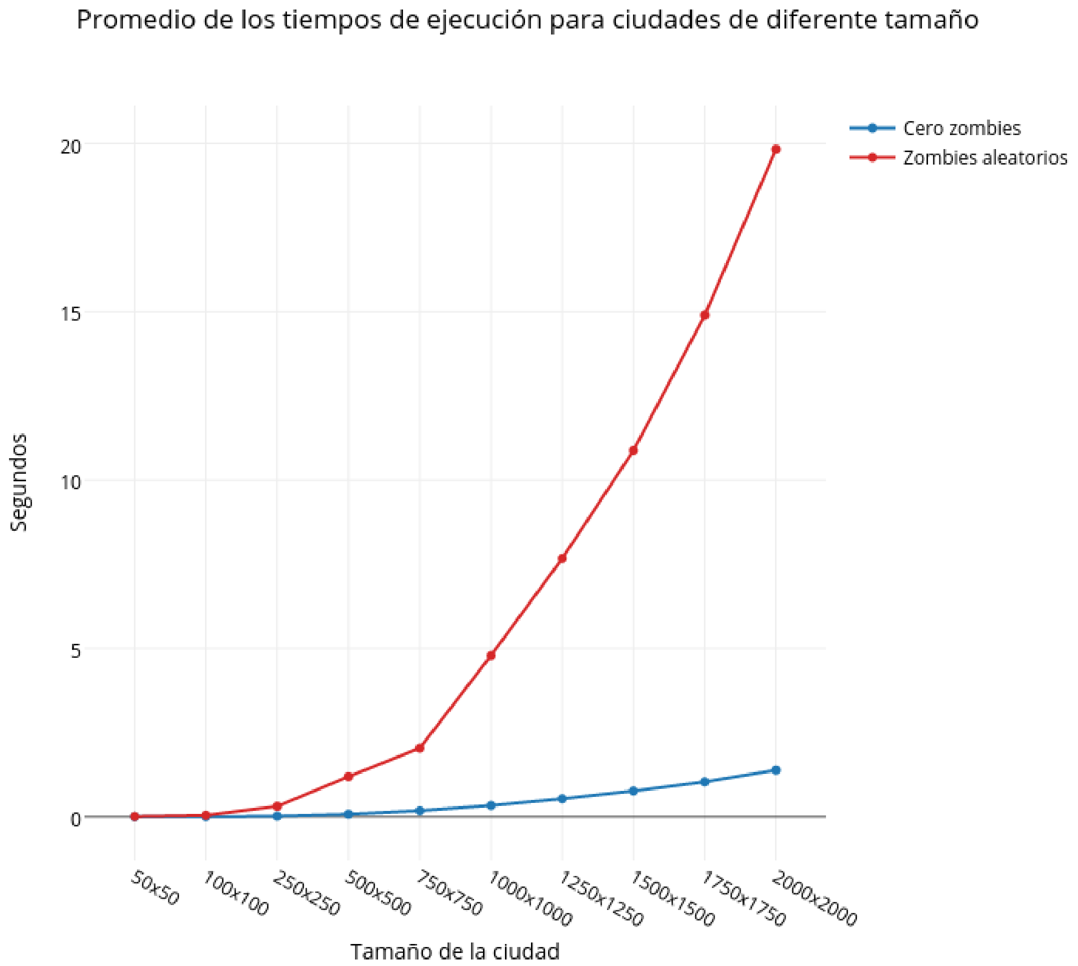
Cero Zombies

Dado que los tiempos de ejecución para ambos experimentos varían ampliamente, primero analizaremos el experimento de **Cero zombies**.



Como se puede apreciar, al no haber zombies, no existe ningun camino en el cual mueran soldados, por lo que los tiempos se incrementan acorde a la dimension de la ciudad, y al ser cuadradas, posee un crecimiento cuadrático. Más adelante justificaremos por qué pensamos que el gráfico es una función cuadrática y no de un grado mayor.

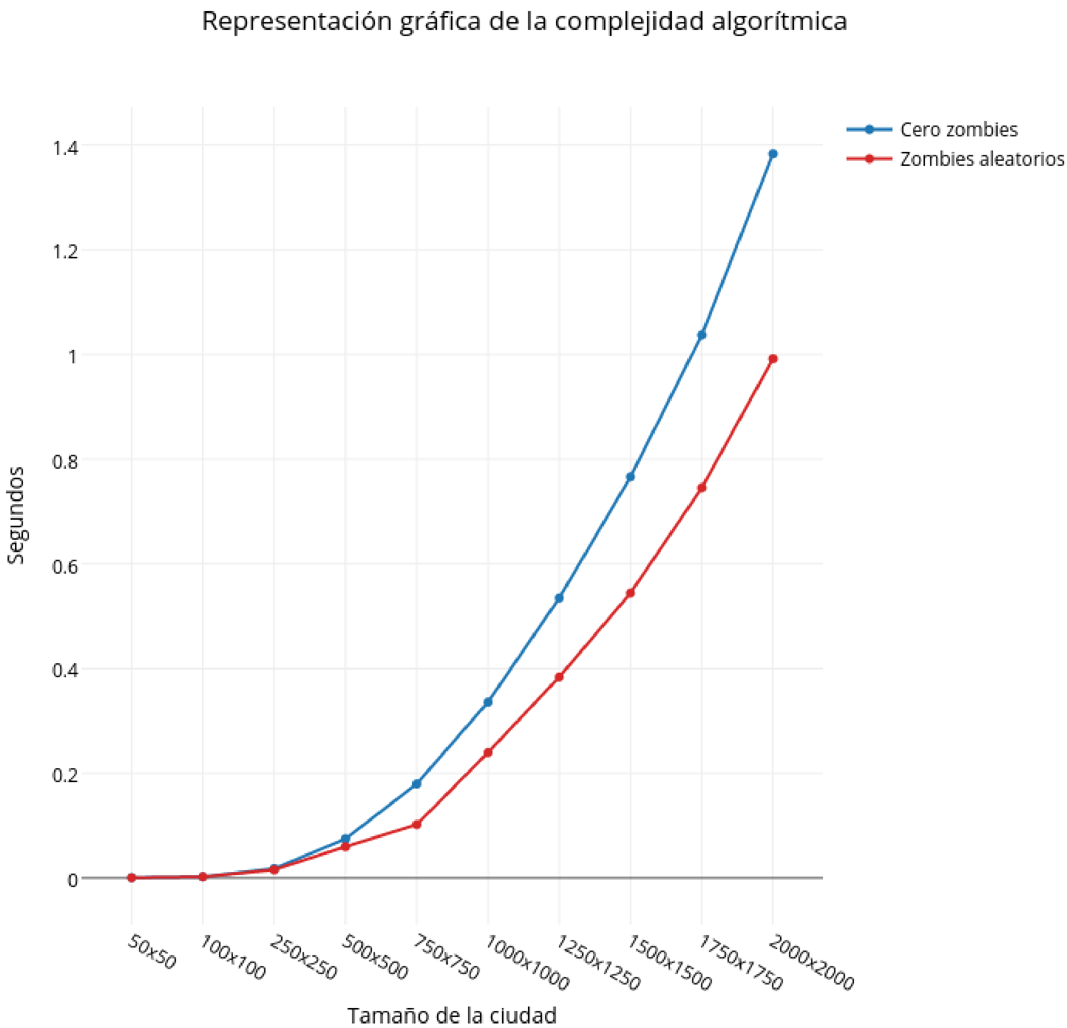
Cero Zombies vs Zombies Aleatorios



Aquí se pueden apreciar las diferencias de tiempos. Ésta radica en el hecho de que **Zombies aleatorios** posee caminos en los cuales los soldados mueren. Por ello, el algoritmo deberá recorrer, en peor caso, s veces la ciudad entera. En cambio, en el caso de **Cero Zombies** sólo arma una "capa" de la ciudad, es decir todos los nodos para las esquinas de la ciudad pero siempre `soldadosVivos`$= 0$.

Se procedió, entonces, a dividir los resultados de **Zombies aleatorios** por la cantidad de soldados iniciales con los que se ejecutó el algoritmo a fin de que quede reflejado, que en ese caso, los tiempos serán muy similares a los de **Cero zombies**. Esto se debe a que ya no son los tiempos de recorrer s veces la ciudad, sino de recorrerla una sola vez y por ende, debe ser una función cuadrática.

Sin embargo, es necesario aclarar que la similitud que se intenta mostrar, es aproximada, ya que no podemos asegurar que efectivamente, el algoritmo recorre s veces la ciudad. Tal es el peor caso, y como hemos expuesto en un principio, no podemos asegurar que sea el que realmente ocurre.



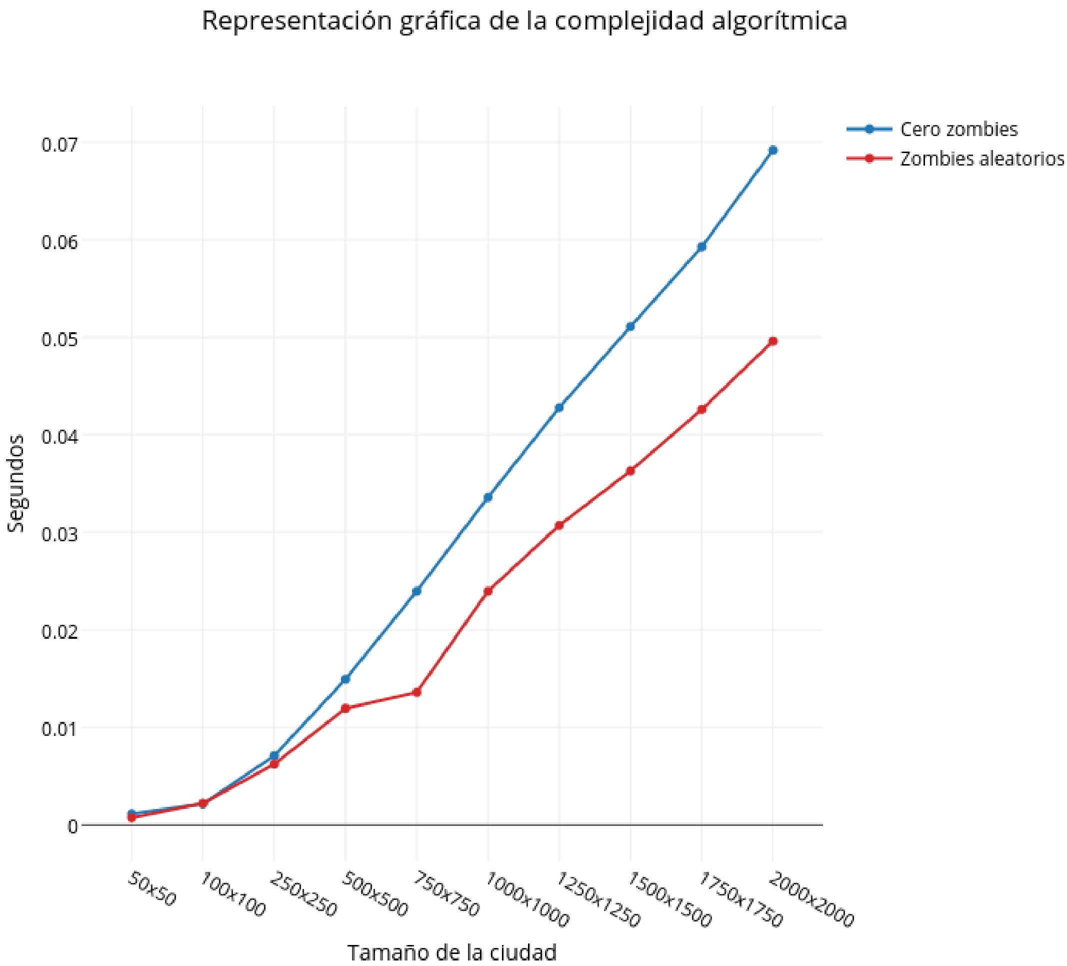
Tal como era esperado, en este grafico se muestran dos funciones cuadráticas, que como se explico anteriormente, solo difieren en la constante que las multiplica.

Manteniendo los resultados de **Zombies aleatorios** divididos por s, finalmente dividimos los resultados tanto de **Cero zombies** como de **Zombies aleatorios**, por n, y en una instancia aparte, por $n.m$.

De esta manera, dado que en ambos tienen s igual a 1, solo queda ver que al dividir por n, los resultados se aproximan a una función lineal, y que al dividirlos por $n.m$, se aproximan a una constante.

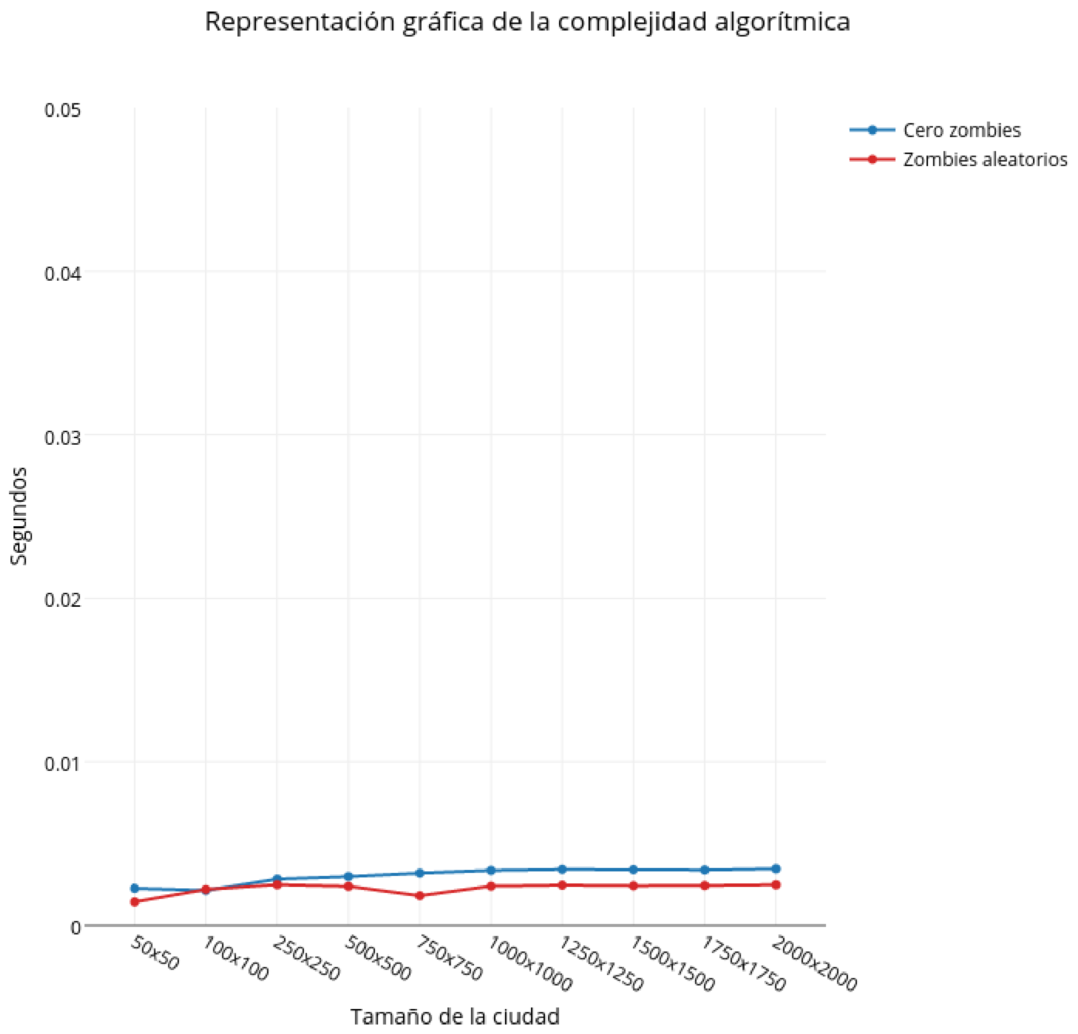
Como sólo nos interesa la relación, y no la función exacta ni la constante, multiplicamos los resultados de dichas divisiones por 100, para que los valores sean más claros y visibles.

El gráfico siguiente corresponde a los tiempos de ejecución divididos por $s.n$ en el caso de **Zombies aleatorios** y por n en el caso de **Cero zombies**.



Como se puede apreciar, el gráfico ahora muestra una función lineal, con lo cuál se inclina a demostrar que en el primer gráfico, la función representada era una función cuadrática. Sin embargo, volveremos a realizar la división para asegurarnos que ésta función, es realmente una función lineal.

El gráfico siguiente corresponde a los tiempos de ejecución divididos por $s.n.m$ en el caso de **Zombies aleatorios** y por $n.m$ en el caso de **Cero zombies**.



Efectivamente, el resultado de la división es una constante, y con esto se muestra que la primer función graficada, correspondía a una función cuadrática.

Se puede ver que la experimentación se corresponde con la teoría, y que la complejidad, en peor caso, es $\mathbf{O(s.n.m)}$.

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.

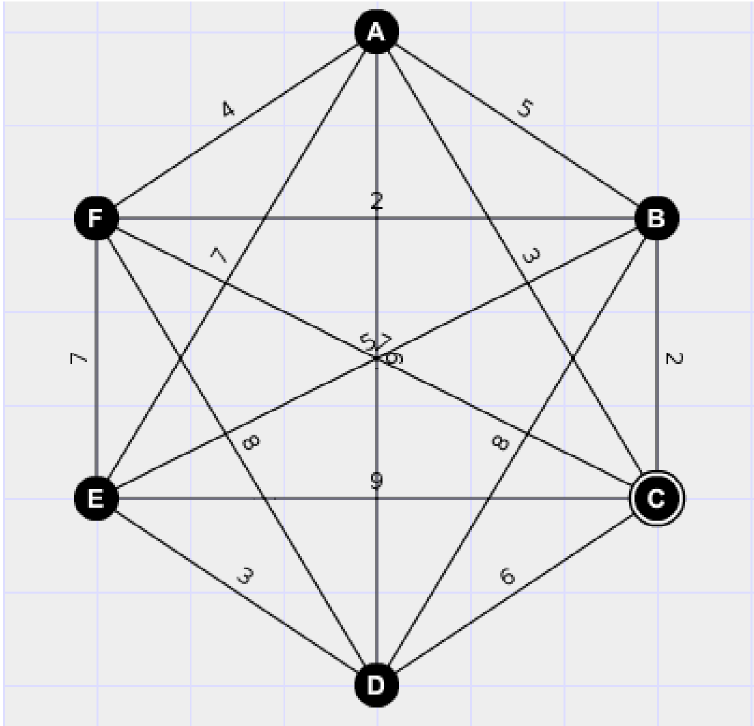
Las refinerías tienen un costo fijo, independiente del pozo donde se ubiquen, mientras que el costo de las tuberías depende de los pozos que se quiera conectar. Además, contamos con la restricción de que sólo se pueden construir tuberías entre los pozos que nos son indicados. Todos estos datos nos son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

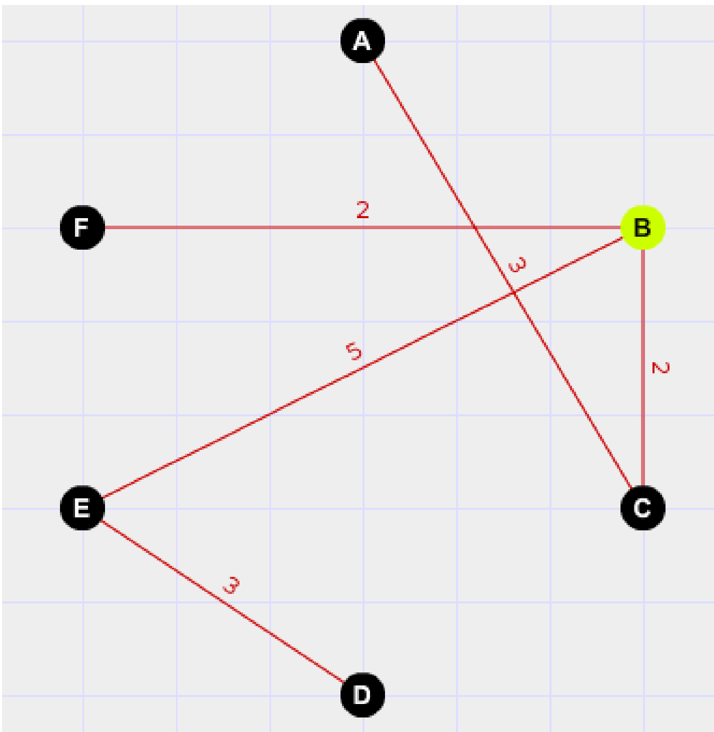
El algoritmo debe tener complejidad mejor que $O(n^3)$ siendo n la cantidad de pozos de la zona.

A continuación citaremos una serie de ejemplos donde los modelamos como grafos, tal que los nodos son los pozos existentes y los ejes iniciales son las posibles conexiones entre ellos con un peso igual al costo de construir una tubería allí. En las soluciones, se preservan sólo los ejes donde construiremos una tubería y los nodos que sean amarillos representan pozos que contendrán una refinería en ellos.

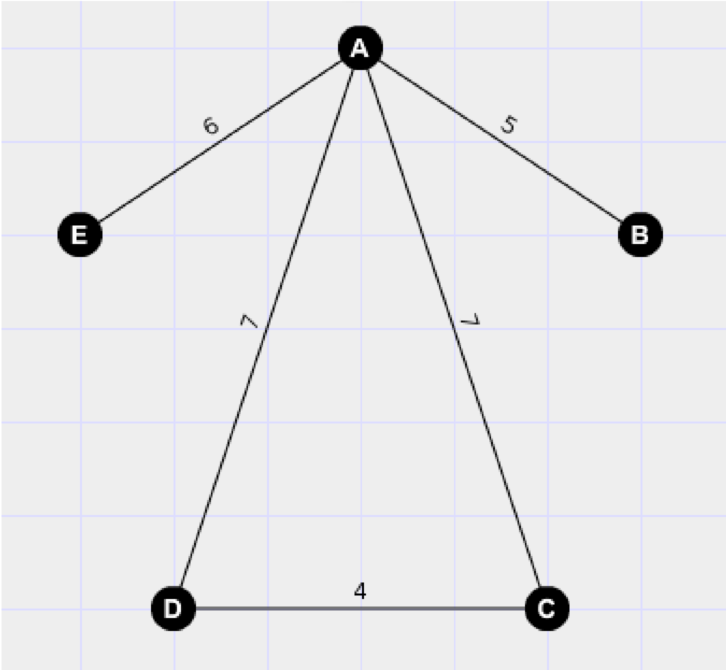
Ejemplo 1: considerando un costo de construir una refinería igual a 7, contamos con la siguiente distribución de pozos y sus posibles conexiones:



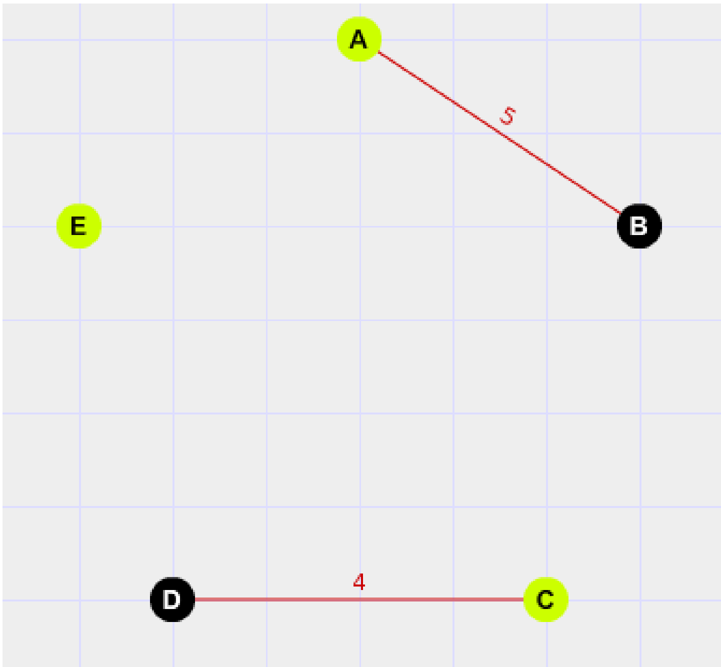
La solución debe tener un costo total de 22 y es correcto devolver una distribución como la siguiente:



Ejemplo 2: Consideramos otra distribución con un costo de construir una refinería igual a 6:



La solución tendrá un costo total de 27, con la siguiente como una distribución válida:



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.

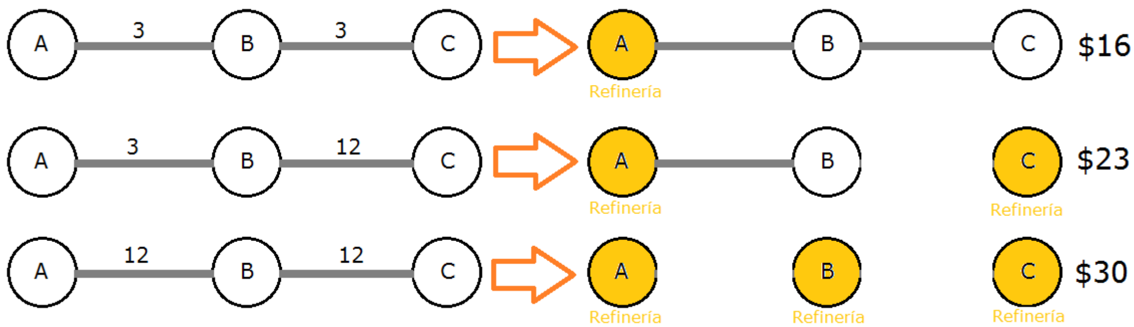
A partir de aquí vamos a trabajar con un algoritmo particular sobre *árboles generadores mínimos*: **algoritmo de Kruskal**.

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.

Consideremos el caso de $n = 3$, donde contamos con los nodos: A, B y C con un costo de construir una refinería de \$10. A continuación, adjuntamos tres ejemplos donde se puede ver que la elección (entre insertar una tubería entre dos nodos o una refinería en cada uno) se condice con insertar una tubería sólo si su costo es menor al de construir una refinería.



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.

PseudoCódigo

input : int m, int n, int C, listaDeEjes $ejes$

Ordenar $ejes$ *en orden creciente acorde a su costo*;

$conexionesMinimas$ ←*Aplicar Kruskal sobre* $ejes$*, eliminando también los ejes que tengan un costo mayor a construir una refinería*;

for $i \leftarrow 0$ **to** $conexionesMinimas.size()$ **do**
  costoTotal$+=$*conexionesMinimas[i]*;
end

for *cada componente conexa* $\in$ *conexionesMinimas* **do**
  Poner una refinería en un nodo de la componente;
end

Algorithm 3: Refinando Petróleo

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 = cantNodos* y *m = 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 construímos 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 $\mathbf{O(n^2)}$.

Luego, vamos a utilizar la estructura *Union-Find* [6] [7]. 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() [8] de la librería Standard de *C*, cuya complejidad es $O(m.log(m))$. Lo cual es equivalente a $O(n^2.log(n^2))$, por propiedades de logaritmo es $O(n^2 2.log(n))$ y eliminando constantes es $\mathbf{O(n^2.log(n))}$.

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. Manejaremos un vector `res` y una estructura *Union-Find* (`bosqueMinimo`), que nos permitirán almacenar los ejes que sean candidatos válidos para nuestra solución.

Al comienzo de la iteración, `res` estará vacío y (`bosqueMinimo`) contendrá a todos los nodos, sin ninguna conexión entre sí. Vamos a recorrer el vector `ejes` y por cada uno de ellos vamos a fijarnos si va a estar en la solución o no.

Esto implica verificar que, al agregarlos, no se forme un ciclo con los ejes ya iterados. Es decir, que sólo se unirán los dos nodos que une este eje dentro de su conjunto correspondiente en (`bosqueMinimo`), si pertenecen a conjuntos distintos. Si es un eje válido y su costo es menor al costo de construir una refinería, entonces lo agregamos a res.

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

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*[7]:

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

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

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

[8]http://www.cplusplus.com/reference/algorithm/sort/

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 $\mathbf{O(n^2)}$.

Tener en cuenta que, dentro del ciclo también se ejecutan asignaciones y push_back() [9] 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, $\mathbf{O(n)}$ dado que un árbol generador tiene a lo sumo $n-1$ ejes.

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

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 refinería (determinamos que el elemento representante de cada conjunto).

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

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

[9] http://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 conexa,
//en realidad solo los ejes que cuesten menos que poner refinerias
    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 refinerias
    vector<bool> refinerias(cantPozos);
    for (int i = 0; i < cantPozos; ++i) {
        if(conexos.find_set(i) == i){
            refinerias[i] = true;
            costoTotal += costoRefineria;
            cantRef += 1;
        }
    }
//cout pedido
    cout << costoTotal << " " << cantRef << " " << conexionesMinimas.size() << endl;
    for (int i = 0; i < refinerias.size(); ++i) {
        if(refinerias[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

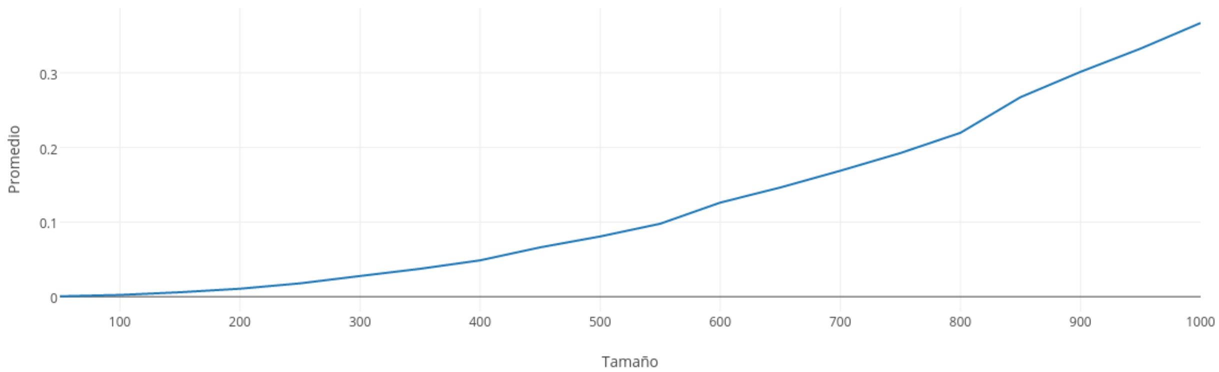
3.5.1. Constrastación Empírica de la complejidad

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

Armamos un script que generó estas instancias, toma como parámetro la cantidad de pozos, y el costo de la refinería, las conexiones no, porque queríamos un grafo completo, este valor será $n.(n-1)$. El mismo escribe un archivo con el formato de la entrada standard, donde a cada eje le asigna un peso random entre 0 y 2∗costoRefinería.

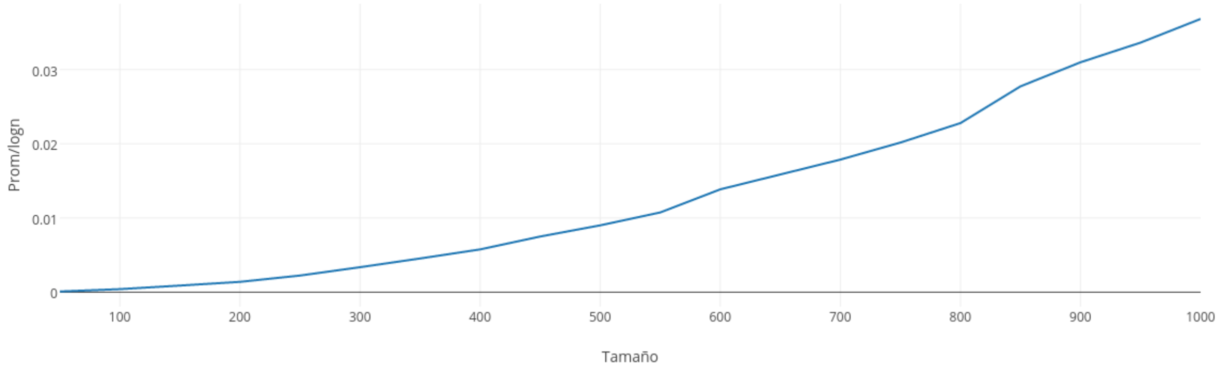
Los tiempos de ejecución para cada n (cantidad de nodos) fueron los siguientes:

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

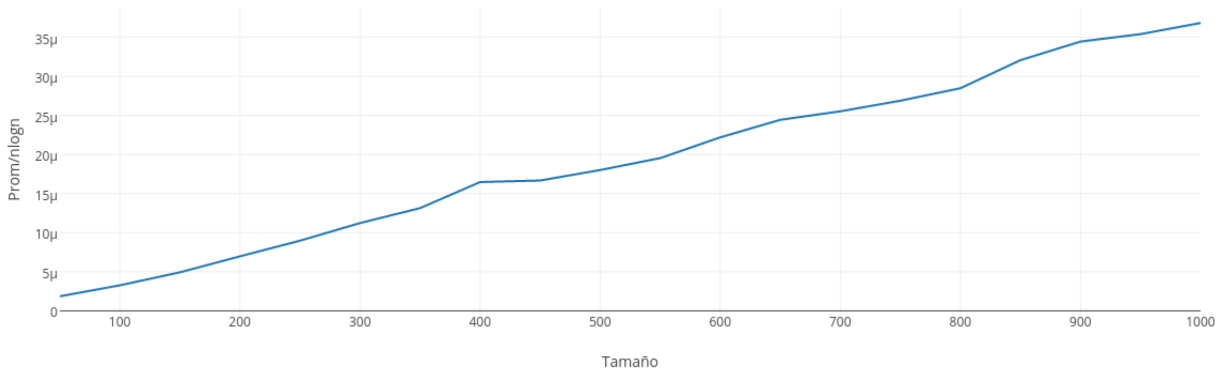


Dado que la Cota de Complejidad planteada teóricamente es de $O(n^2.log(n))$, era esperable que la curva sea una parábola creciente.

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 $\mathbf{O(n^2.log(n))}$.