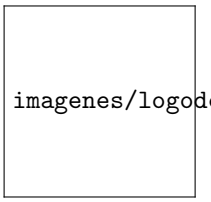

Trabajo Práctico II

Algoritmos y Estructuras de Datos III
Segundo Cuatrimestre de 2016

Integrante	LU	Correo electrónico
Nicolás Bukovits	546/14	nicobuk@gmail.com
Kevin Frachtenberg	247/14	kevinfra94@gmail.com
Julián Len	467/14	julianlen@gmail.com
Nicolás Len	819/11	nicolaslen@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

Índice

1. Introducción

En este trabajo se tienen tres problemas que se resolvieron aplicando las técnicas algorítmicas estudiadas en la materia. Los mismos presentaron cada uno un desafío distinto, teniendo que aplicar métodos diferentes para resolverlos y cumplir con los requisitos exigidos.

Además de la resolución de los mismos, se procedió a demostrar la correctitud de cada implementación. Esto fue acompañado a su vez de una justificación de la complejidad temporal.

Cada ejercicio contó con su respectiva experimentación para corroborar que la complejidad temporal teórica se cumpliera y en los casos donde el algoritmo podía comportarse mejor, verlo reflejado de alguna manera.

Los experimentos contaron con diversas medidas para asegurar su efectividad de las cuales las siguientes fueron iguales para los tres problemas:

- Sólo se midió el costo temporal de generar la solución, no de lectura y escritura del problema.
- Para la medición del tiempo se utilizó la biblioteca `chronos` con unidad de tiempo en microsegundos.

2. Ejercicio 1: Laberinto

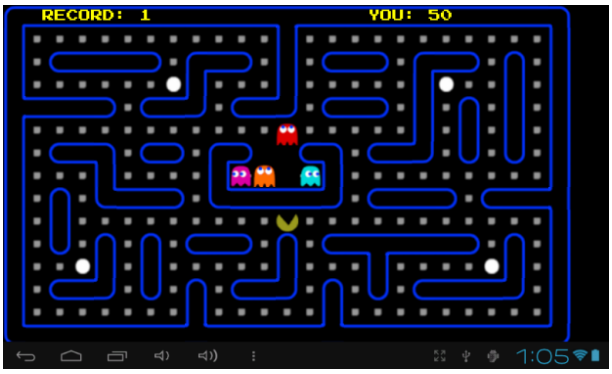

Figura 1: Perdidos y con poca fuerza

2.1. Descripción del problema

Indiana Jones continua en su expedición en la fortaleza de alguna civilización antigua. Mientras llenaban las mochilas con los tesoros que más les convenian encontraron un mapa peculiar. El mapa se parece mucho a un laberinto salvo por el hecho de que no estan conectados todos los puntos. En el mapa hay un punto que parece indicar el lugar en donde se encuentran juntando tesoros, y hay un lugar que esta indicado con una cruz. Intrigados por saber lo que se encuentra en ese lugar, se proponen como objetivo ir hacia ahi. Como nuestro equipo vino equipado con pico y pala, pueden derribar algunas paredes. Por otro lado el equipo ya se encuentra cansado y no desea recorrer mucha distancia ni esforzarse rompiendo paredes. Entonces, nos pidieron ayuda con lo siguiente: Teniendo un mapa indicando un punto de origen (identificado con una o) y un punto de destino (identificado con una x), quieren caminar lo menos posible desde el origen al destino rompiendo a lo sumo una cierta cantidad de paredes.

La resolución del problema consiste en encontrar el camino más corto que derribe a los sumo P paredes, las cuales se reconocen de un mapa dibujado con . y # que indican los lugares por donde se puede pasar y las paredes respectivamente. Cada paso cuenta como 1 avance desde el origen y cuando una pared es derribada, se toma en cuenta como un lugar por el que se puede pasar, por lo que suma 1 también como parte del camino. El mapa tiene tamaño F filas y C columnas, los cuales se pasan por parametro al programa junto con la cantidad de paredes y el mapa. En caso de que este camino no exista, entonces la salida será solamente -1

Por ejemplo, si el programa recibe lo siguiente como entrada:

```
5 9 2
# # # # # # # # #
# o # . # . # x #
# . # . # . # . #
# . # . # . . . #
# # # # # # # # #
```

La salida correcta sería:

```
10
```

2.2. Solución propuesta

Este problema resulta más simple de entender cuando se piensa el mapa como un grafo dirigido en tres dimensiones, donde la cantidad de nodos será de $FC(P+1)$ y en cada nivel habrá FC nodos. De esta manera, podemos ver a cada piso del grafo (comenzando en el piso 0) como la cantidad de parades que se rompieron hasta ese punto. Los nodos estarán conectados cada uno con sus vecinos, aunque si uno de ellos es una pared, entonces el nodo se conectará con el vecino que representa a la pared pero un nivel

más arriba. Hay que tener en cuenta que los nodos se modelan, para cada nivel, simplemente como el número de nodo en el grafo base (el del nivel 0) más la cantidad de nodos de cada nivel, y sus conexiónes (o aristas) se almacenan en listas de adyacencias.

Teniendo el mapa representado de esta manera, sabemos en todo momento que la cantidad de paredes derribadas es menor o igual al límite establecido, por lo que si se puede llegar a la x destruyendo una cantidad menor o igual de paredes que P, entonces existe un camino entre el origen y el destino.

Para encontrar el camino mínimo entre o y x utilizamos el algoritmo conocido como *Breadth-first Search* o *BFS*, el cual conciste en recorrer el grafo analizando una sola vez cada nodo y desde éllos, observar sus vecinos. Si los vecinos de ese nodo aún no fueron analizados, se agregan a una cola para ser revisados después. Cada vez que se observa un vecino, chequeamos que no haya sido visto con anterioridad y si ese es el caso, se marca como visto cargando su distancia al origen en una *lista de distancias* y agregándolo a la cola. Este algoritmo funciona en grafos donde las aristas no tienen peso o sus pesos son todos iguales, como en este ejercicio que como se dijo anteriormente, avanzar en el mapa implica sumar 1 a la cantidad de pasos, o bien aumentar en 1 la distancia al origen. Luego, los pesos de las aristas serán todos de valor 1 y puede utilizarse BFS.

La *lista de distancias* tiene el tamaño de la cantidad de nodos totales y cada posición i representa al nodo i del grafo. Esta lista solamente contiene la distancia al origen de cada nodo, inicializándose con -1 para todos y a medida que avanza el algoritmo, si el vecino k del nodo i, que es el que se está analizando tiene distancia menor a 0, entonces quiere decir que aún no ha sido observado.

2.2.1. Detalles implementativos

Para poder manejar correctamente el grafo, se creó la clase ListaAdy con namespace Grafos, la cual modela un grafo representado con listas de adyacencias. Las únicas operaciones que tiene la clase son el constructor, que recibe la cantidad de nodos totales, agregarArista, que recibe dos enteros u, v y agrega v a la lista de adyacencias de u, y BFS que recibe el número de nodo origen s, el número de origen de destino t, y la cantidad f de filas y c de columnas. La forma en que se almacenan las listas de adyacencias es con un vector de vectores de enteros, tipo que se provee de la librería estándar de C++.

Normalmente, BFS solo necesitaría el nodo de origen y de destino, y almacenaría todos las posibles distancias de caminos que hay entre ambos nodos. Sin embargo, agregamos estos dos valores para que el algoritmo termine la primera vez que encuentre el nodo destino. Esto es por cómo funciona BFS: Al recorrer todos los nodos según su distancia al origen, siempre se analizan los nodos que solo están a distancia 1 más que el nodo vecino "padre"(o sea, el nodo que tenía como vecino al que se está analizando y por el cual se agregó a la cola de revisión). Ergo, si existe un camino entre s y t, BFS encontrará el más corto antes que cualquier otro y por eso es que podemos terminar su ejecución cuando eso pase. En peor caso, no existirá un camino entre s y t, y BFS recorrerá todos los nodos.

El algoritmo que resuelve el problema puede separarse en dos partes: Una se encarga de leer el mapa de entrada y construir el grafo dirigido tridimensional y la otra, es el BFS que busca verdaderamente la solución. En la primera parte recorremos el mapa que se encuentra guardado en una matriz de **char** $P + 1$ veces, para así poder construir los $p + 1$ niveles del grafo y poder conectar cada nodo con sus respectivos vecinos, y para poder verificar que el mapa pasado sea válido y buscar los nodos marcados con la o y la x. La segunda parte se recorre el grafo una sola vez dado el funcionamiento de *BFS*, pero como tiene $FC(P + 1)$ nodos, entonces esa será la cantidad de nodos que pasarán por la cola de revisión como máximo. La función BFS está implementada de la siguiente manera:

```
BFS(enteros : s, t, f, c)
  res = -1
  cola = cola vacía
  distancias[nodosTotales]
  para i entre 0 y nodosTotales-1
    distancias[i] = -1
  fin para

  cola.encolar(s)
  distancias[s] = 0
  mientras (cola no esté vacía)
    tope = cola.tope
    cola.desencolar //al ser una cola, desencola el tope de la misma
    si (tope % (f*c)) == t
      devolver distancias[tope]
    fin si

    para i entre 0 y largo(vecinos(tope))
      si distancias[vecinos(tope)[i]] < 0
        distancias[vecinos(tope)[i]] = distancias[tope] + 1
        cola.encolar(vecinos(tope)[i])
      fin si
    fin para
  fin mientras

  devolver res
```

El único cambio que tiene esta implementación de BFS respecto de la original, es que chequea si el resultado se encuentra antes de terminar, lo cual puede hacerse por lo antes dicho. Como complemento, podemos decir que en el código puede observarse que *distancias*[*tope*] siempre existe y es mayor a 0 porque antes de agregar un nodo a la cola, se carga en *distancias* su valor correspondiente. Además, la razón por la cual se chequea el resto de dividir *tope* por $f * c$ es que t es el número del nodo con la x en el nivel 0, mientras que al ser un grafo tridimensional donde cada nivel representa la cantidad de paredes rotas desde el origen, t va a estar en todos los niveles con una diferencia de $f * c * nivelActual$. Ergo, al dividir tope por $f * c$, el resto debería ser el número de nodo que se va a analizar como si fuera uno del nivel 0.

2.3. Complejidad teórica

Para este análisis, nuevamente separaremos el algoritmo en dos partes (construcción del grafo y BFS). En la primera parte, se generan $P + 1$ niveles de FC nodos que como se explica en el punto anterior, se representan mediante listas de adyacencias. Agregar una arista entre dos nodos cuesta $O(1)$ ya que es simplemente agregar el número de nodo destino de la arista a la lista de adyacencia del nodo origen. Hacer esto por cada nodo costaría $O(|E|)$ por cada nivel con $|E|$ siendo el número de aristas. Sin embargo, al ser un digrafo tipo *grid*, la cantidad máxima de aristas por nodo es de 4 porque cada nodo se conecta con, a lo sumo, un nodo a cada lado (izquierda, derecha, arriba y abajo). Además, si bien hay FC nodos en cada nivel, los nodos que sean paredes no tendrán aristas de entrada que provengan de nodos del mismo piso. Luego, $|E| \leq 4FC \in O(FC)$. Como esto ocurre $P + 1$ veces, la complejidad de la primera parte es

$$O((P + 1)FC) \in O(FCP)$$

La segunda parte, el algoritmo conocido como *BFS*, tiene complejidad $O(|V| + |E|)$ siendo $|V|$ la cantidad de nodos. Ya probamos que $|E| \in O(FCP)$ y sabemos que $|V| = FC$, entonces la complejidad de la segunda parte es

$$O(FC + FCP) \in O(FCP)$$

Considerando las dos partes juntas, nos queda que la complejidad temporal es

$$O(FCP + FCP) \in O(FCP)$$

En cuanto a la complejidad espacial, el tamaño del grafo sobre listas de adyacencias es de $O(FCP)$ ya que es la cantidad de nodos totales sumado a lo que, por lo probado anteriormente, acota la cantidad de aristas totales.

2.4. Experimentación

Para poder mostrar que la cota propuesta en la complejidad temporal funciona para el algoritmo que resuelve este problema, realizamos experimentos con diferentes matrices y diferentes cantidades de paredes que se pueden destruir.

Primero se realizó un experimento con 5 matrices de tamaño 10x10. Una matriz estaba compuesta por todas paredes excepto el origen y el destino, otra tenia la mitad de paredes horizontales respecto del total de nodos, otra la mitad de paredes verticales, otra la mitad de paredes diagonales y la última no contenía ninguna pared y el origen y el destino estaban uno al lado del otro. Se optó por realizarlo de esta forma ya que se pueden probar casos límite, que son cuando se tienen que romper todas paredes, mientras que en el lado opuesto no hay que romper ninguna y el destino está a un sólo nodo de distancia del origen. Por como esta implementado el algoritmo se esperó que en los casos en donde hay que romper mucha más cantidad de paredes tarden más en ejecutarse que los que no hay que romper ninguna y además está más cerca el nodo destino del origen. Los esquemas de las matrices que se utilizaron para realizar esta parte del experimento son los siguientes:

```
o # # # # # # # # #
# # # # # # # # # #
# # # # # # # # # #
# # # # # # # # # #
# # # # # # # # # #
# # # # # # # # # #
# # # # # # # # # #
# # # # # # # # # #
# # # # # # # # # #
# # # # # # # # # x

o # . # . # . # . #
. # . # . # . # . #
. # . # . # . # . #
. # . # . # . # . #
. # . # . # . # . #
. # . # . # . # . #
. # . # . # . # . #
. # . # . # . # . #
. # . # . # . # . #
. # . # . # . # . x

o # # # # # # # # #
. . . . . . . . . .
# # # # # # # # # #
. . . . . . . . . .
# # # # # # # # # #
. . . . . . . . . .
# # # # # # # # # #
. . . . . . . . . .
# # # # # # # # # #
. . . . . . . . . x

o # . # . # . # . #
# . # . # . # . # .
. # . # . # . # . #
# . # . # . # . # .
. # . # . # . # . #
# . # . # . # . # .
. # . # . # . # . #
# . # . # . # . # .
. # . # . # . # . #
# . # . # . # . # x

o x . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
```

El expermiento se corrío para cada matriz teniendo en cuenta que se pueden romper desde 0 hasta 99 paredes. Es decir, en total se examinaron 500 casos (100 para cada una de las cinco matrices). Además

para cada caso con una determinada cantidad de paredes, se repitió el mismo experimento 100 veces para luego tomar el promedio y obtener un resultado más representativo del tiempo que tarda cada caso. Los primeros casos en los que se podían romper pocas paredes fueron los que menos tiempo de ejecución demandaron. Los resultados obtenidos fueron plasmados en el siguiente gráfico. El mismo es la representación del tiempo en funcion de la cantidad de paredes que se pueden romper para cada una de las cinco matrices consideredas. También se muestra la funcion propuesta como cota de complejidad temporal.

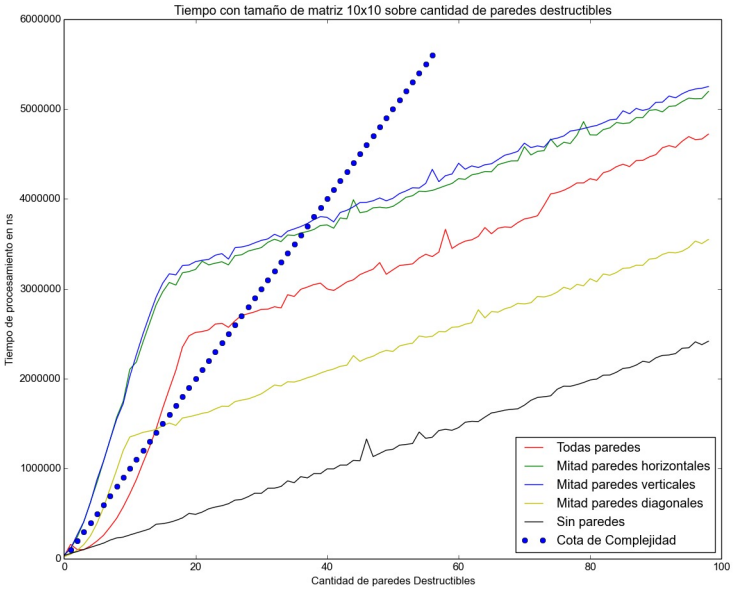

Figura 2

Para cada matriz puede observarse que a medida que crece la cantidad de paredes que se pueden derribar, también crece el tiempo que toma cada ejecución del programa, que era el comportamiento que se esperaba. Si bien el algoritmo de BFS corta la ejecución en cuanto encuentra el destino (en caso de poder encontrarse), no se observa una mejoría cuando se utiliza una cantidad mayor de paredes que se pueden destruir a la cantidad de nodos, debido a que en la construcción del grafo se toman en cuenta todos los niveles (cantidad de paredes), por lo que esa parte del algoritmo ya toma $O(FCP)$. Por esta razón es que el tiempo de ejecución no se mantiene constante una vez que se toma una cantidad de paredes mayor al total de nodos. El algortimo como se puede observar también tarda menos en ejecutar cuando en la matriz no hay ninguna pared y el origen y el destino están uno al lado del otro, como se esperaba. Este es el mejor caso. Los casos en los cuales había una cantidad de paredes igual a la mitad de nodos y éstas estaban en posiciones horizontales y verticales fueron aún peor que en el caso que son todas paredes. Esto puede explicarse debido a que el algoritmo de BFS inspecciona algunos vecinos que en el caso de que son todas paredes no puede ya que no tiene ningún camino que pueda pasar sin romper paredes. Podemos notar como la cota de complejidad cumple, aunque no de manera exacta, su función para estos experimentos.

Luego se realizó otro tipo de experimento en el cual se utilizaron otras cinco matrices pero con tamaño y paredes dispuestas de manera aleatoria. Estas matrices fueron generadas usando la librería random de la STL con distribución uniforme. Este experimento se corrió bajo las mismas circunstancias que el otro experimento. Se corrieron 100 casos para cada una de las matrices, considerando que se pueden romper desde 0 a 99 paredes. Además también cada uno de los casos se repitió 100 veces y se tomó el promedio. Los resultados que se obtuvieron son los siguientes:

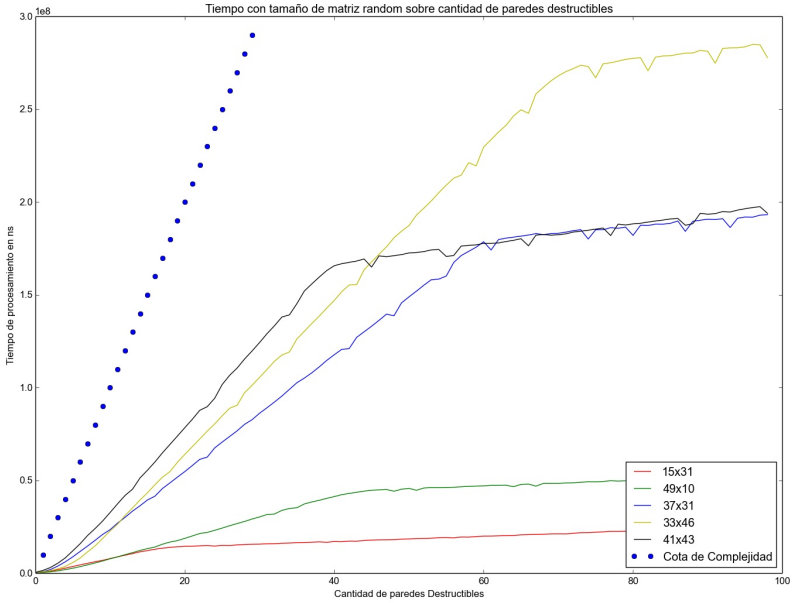

Figura 3

Se observa también como el tiempo de ejecución aumenta a medida de que se pueden romper más paredes para cada una de las matrices. Por lo general, los casos en los cuales la matriz tiene menos filas y columnas presentan un tiempo de ejecución menor pero pueden observarse que igualmente también depende de la cantidad de paredes que haya en la matriz. Para estos casos, la cota propuesta acota mejor de manera superior a todos los casos.

3. Ejercicio 2: Menos es más

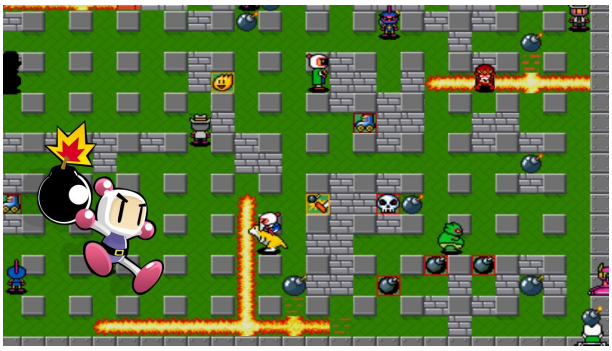

Figura 4: Romper paredes

3.1. Descripción del problema

En su destino hacia la cruz en el mapa recorrieron varias salas distintas. Se dieron cuenta que en cada sala se iban encontrando con un fragmento de una tabla con un mansucrito antiguo. Por lo tanto, antes de continuar, quieren juntarlos todos.

A medida que se iban abriendo camino se dieron cuenta que ciertas paredes derribables requerían cierto esfuerzo para abrirse. Este esfuerzo será representado por un $n \in \mathbb{N}$ donde $n < 10$. Luego, teniendo un mapa de $F * C$ (F = Filas y C = Columnas), donde se indique el n en las paredes derribables, quieren encontrar cual es el mınimo esfuerzo que deben hacer para acceder a todas las salas.

Por lo tanto, se deberá pasar por parametro al algoritmo, el mapa con sus paredes y el esfuerzo para romperlas.

Formato de entrada: La primer línea consta de dos enteros positivos F y C, donde F es la cantidad de filas en el mapa, y C es la cantidad de columnas. Las siguientes F filas contienen C caracteres. Un "." representa un lugar caminable, los números del 1 al 9 indican el esfuerzo en romper esa pared, un "#" representa una pared indestructible. Las paredes con más de 2 paredes adyacentes son indestructibles (contendran un "#"). Las paredes con menos de 2 paredes adyacentes son indestructibles (contendran un "#"). Dos salas se pueden conectar si existe una pared adyacente a las dos salas que se puede destruir (Es decir, no quedan conectadas si se rompen muchas paredes adyacentes). Por último, los bordes del mapa tendrán un "#".

```
F C
F_1
...
F_n
```

Salida: La salida consta de un número indicando el esfuerzo que debe hacer el equipo para acceder a todas las salas. Si no es posible recorrer todas las salas se imprime un 1. La salida tendrá el siguiente formato:

```
E
```

Se tiene como restricción que F y C está en el siguiente rango: $1 <= F, C <= 10^4$. La complejidad temporal debe ser a lo sumo $O(FC * \log FC)$.

3.2. Solución propuesta

3.2.1. Modelado

Para resolver este problema, se pedía encontrar el mínimo esfuerzo, para pasar por todas las salas del mapa. Para esto se usó el algoritmo de Kruskal, cuyo fín es encontrar el **Arbol Generador Mínimo** dentro de un grafo, cuyo pesos en las aristas sea positivo. En nuestro caso, esto representaría el mínimo esfuerzo, para poder visitar todas las salas. Para poder usar Kruskal, modelamos el problema en grafos. Para esto, se creó una matriz de $F * C$, donde fueron guardados los caractéres del mapa pasados como parámetro de entrada.

Luego se usó esta matríz para generar los nodos y las aristas. Para esto, en principio cada posicion de la matriz es un nodo, numerados según la posición dentro de la matriz ($filaDelNodoActual * cantColumnas + columnaDelNodoActual$).

Una vez generada la matríz, se la vuelve a recorrer, para generar las aristas. Sea nodoActual, el caracter ubicado en la posición de la matríz en la cual estoy queriendo generar una arista,

- Si nodoActual es "#", no hago nada.
- Caso contrario, me fijo si nodoActual es "." o un número.
  - Si es un ".". Se debe fijar en la columna siguiente de la matriz, a este nodo lo llamaremos, nodoVecino.

    Si nodoVecino es distinto de "#". Genero una arista, cuyo comienzo es el número de nodoActual, su fin es el de nodoVecino, y el peso es 0. Análogo para la posición inmediata dentro de la matriz, de la fila siguiente.
  - Si es un número. Deberá hacer lo mismo que en el caso anterior. Con la diferencia que, si uno de sus vecinos es también un número, no deberá hacer nada y que solo tendrá un vecino ".", en la columna inmediata superior o, en la fila siguiente, dentro de la matriz, y cuyo peso de la arista, será el número del nodo.

Luego para el siguiente mapa,

```
#########
#.1.#.#.#
#.#.3.5.#
#.2.#.#.#
#########
```

Figura 5: Mapa a modelar

Queda el siguiente grafo,

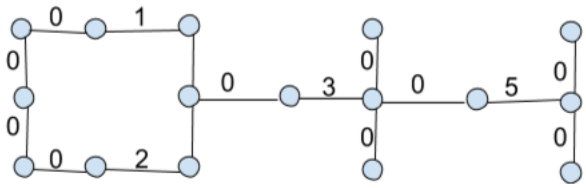

Figura 6: Grafo del mapa

Cada una de estas aristas, es guardada en un vector de aristas. Luego, este vector de aristas, es usados como parámetro de entrada, en el algoritmo de Kruskal, quien devolverá la suma del **Arbol Generador Mínimo**. En caso de no tener solución, es decir, no poder recorrer el mapa, devolverá -1.

3.2.2. Implementación

Habiendo introducido la idea, se detalla el comportamiento del algoritmo para una entrada arbitraria.

La primera parte consiste en la inicialización de valores para el algoritmo. Se guardar la cantidad de filas y de columnas del mapa pasado por standard input. Se crea una matriz con este mismo tamaño. Luego se guardan en esta matriz, los caractéres pasados como parámetros de entrada. Se inicializan valores booleanos para chequear la posibilidad donde no haya solución.

```
for i = 0 .. filas
    for j = 0 .. columnas
        matriz[i][j] <-- (cin >> caminoPared);
    end
end
```

Luego el ciclo principal, consiste en la creación de aristas y nodos. Los nodos serán cada una de las posiciones de la matriz, enumerados desde la posición de la matriz (0,0) hasta (#filas, #columnas) cada arista creada será guardada en un vector de aristas. Por cada posición recorrida de la matriz se hacen los siguientes chequeos

1. Si matriz[filaActual][columnaActual] es un "#", no hago nada.
2. Caso contrario, si es un ".":
    2.1. Chequeo matriz[filaActual+ 1][columna] . Si este es distinto de "#", creo una arista que empiece desde matriz[filaActual][columnaActual], termine en matriz[filaActual + 1][columnaActual], con peso 0. Lo mismo para matriz[filaActual][columnaActual + 1]. Además aumento la cantidad de nodos y chequeo que el mismo no esté en encerrado, guardando esto en una variable booleana **encerradoRes**
3. Luego, si es un número:
    3.1. Chequeo matriz[filaActual+ 1][columna] . Si este es un ".", creo una arista que empiece desde matriz[filaActual][columnaActual], termine en matriz[filaActual + 1][columnaActual], con peso igual al número. Lo mismo para matriz[filaActual][columnaActual + 1]

```
    for i = 0 .. filas
        for j = 0 .. columnas
           if (matriz[i][j] == '.')
                nodos++
                if (matriz[i][j+1] != '#')
                     creo arista a
                     a.inicio <-- matriz[i][j]
                     a.fin <-- matriz[i][j+1]
                     a.peso <-- 0
                     aristas.push_back(a)
                endif
                if (matriz[i+1][j] != '#')
                     creo arista a
                     a.inicio <-- matriz[i][j]
                     a.fin <-- matriz[i+1][j]
                     a.peso <-- 0
                     aristas.push_back(a)
                endif
           endif
           if (esNumero(matriz[i][j]))
                if(matriz[i][j + 1] == '.')
                     creo arista a
                     a.inicio <-- matriz[i][j]
                     a.fin <-- matriz[i][j+1]
                     a.peso <-- matriz[i][j]
                     aristas.push_back(a)
                endif
                if(matriz[i + 1][j] == '.')
                     creo arista a
                     a.inicio <-- matriz[i][j]
                     a.fin <-- matriz[i+1][j]
                     a.peso <-- matriz[i][j]
                     aristas.push_back(a)
                endif
           endif
       end
    end
```

Luego se chequea que hayan más de un nodo o ".", y en este caso, que no haya ninguno inalcanzable con encerradoRes. Si pasa que no hay nodos, o que si hay más de uno, que alguno esté encerrado, se devuelve -1 (no hay solución).

```
    if(nodos == 0 || (nodos > 1 && encerradoRes))
     res <-- -1
     cout << res << endl;

    endif
```

Si la solución es posible, se usa Kruskal con el vector de aristas, quien generará un vector solución de aristas con el **AGM**. Se sumará el peso de todas las aristas de este vector, para devolver el peso mínimo para recorrer todos los "." del mapa.

```
    int V = f*c; //Kruskal
init(V);
sort(aristas.begin(), aristas.end())
    vector<arista> solucion;
    for i = 0 .. aristas.size()
        arista a = aristas[i]
        if (find(a.inicio) != find(a.fin))
            solucion.push_back(a)
            union(a.inicio, a.fin)
        endif
    end //Fin Kruskal

  res <-- 0

    for i = 0 .. solucion.size()
        res += solucion[i].costo
    end
```

3.2.3. Detalles Implementativos

1. Para guardar el mapa de parámetro de entrada, se usó una matriz de Char.
2. Para chequear que un Char es un número válido, se usó la función **isdigit(char)** perteneciente a la librería **ctype**, y luego se chequeó que el número este entre 1 y 9.
3. Para guardar las Aristas, se usó la estructura Aristas. La misma contaba con los campos Inicio, Fin y Costo.
4. Para usar Kruskal, se generó un vector con todas las aristas del grafo.
5. En Kruskal se usó la estructura de datos **Disjoint Set Union** (DSU), estructura que cuenta con las operaciones **find** y **union**.

3.2.4. Demostración de correctitud

Habiendo visto cómo funciona el algoritmo desarrollado, se procede a justificar por qué devuelve una solución. Para esto, se explicará detalladamente, porque el modelado es correcto. Dado que para la segunda parte, se usa Kruskal, algoritmo que definitivamente, es correcto.

Correctitud de Modelado

A la hora de modelar un grafo en base a la matriz de entrada, se decidió modelarlo desde la posición de la matriz (0,0) hasta (n,m) siendo n, cantidad de filas y m, cantidad de columnas. El único caso el cual tomamos una decisión la cual se debe demostrar correctitud, es para los nodos en donde hay un número. Para estos casos, por restricciones del enunciado, cada nodo que contiene un número, puede tener solo una posibilidad. Es decir, sea (i,j) la posición del nodo con número, su vecino al cual puede acceder (es decir, la posición de la matriz con un nodo que contiene un “.”) podría ser (i,j+1) o (exclusivo) (i+1,j).

Luego, según a cual se puede acceder, se crea una arista con el peso igual al número del nodo. Esta arista comienza desde el nodo que contiene el número y finaliza en su vecino accesible.

Esto genera, además de las aristas sin peso, aquellas que representarán el esfuerzo para romper una pared.

Si bien, se toma una decisión al chequear desde el nodo con número hacia sus vecinos (i, j+1) o (i+1,j), pues podría ser desde un nodo (i,j) hacia (i, j-1) o (i-1,j). Por un tema de equivalencia, se comparó con los vecinos (i, j+1) o (i+1,j). Esta equivalencia es dada, pues, al generar una arista con peso, significa que existe la posibilidad que se necesite ir desde la pared con el número, hacia un sector accesible con un

“.”. Pero, dado el caso que sea necesario atravesar esa pared, al algoritmo de Kruskal, le es indiferente, si la arista va desde el nodo con un número (i,j) a (i, j-1) o a (i,j+1). Si es necesario atravesar esa pared, lo hará de izquierda a derecha o de derecha a izquierda. De otra manera, si no es necesario no la atraviesa, pues hay un camino para acceder a los extremos de la arista con peso, de una manera óptima, o sea, con menor peso. Análogo es para los casos donde los vecinos son los inmediatos superiores o inferiores o sea (i-1, j) o (i+1,j).

Luego, dada esta decisión, es generado un grafo, donde las aristas entre dos nodos “.”, tienen peso 0, y las aristas desde un número hacia su vecino tienen el peso correspondiente al número y representan romper la pared.

Finalmente, el algoritmo de Kruskal, va a buscar llegar a todos los nodos, con la menor cantidad de roturas de pared. O sea, pasando por las aristas con menor peso. En términos de grafos. Dado el grafo conexo G generado a partir de la matriz de entrada, cada una de las aristas tendrá un peso no negativo. Dada esta precondición es posible usar Kruskal, algoritmo que devolverá el **AGM** de G. Es decir, por que aristas se debe pasar para llegar a todos los espacios del mapa o nodos del grafo. Luego la suma de los pesos de estas aristas, es igual al resultado devuelto. En el caso que G sea disconexo, el algoritmo devolverá -1, pues significa, que existe un espacio del mapa, o un nodo del grafo, donde es imposible llegar.

3.3. Complejidad teórica

Para calcular la complejidad teórica de la solución propuesta se hará referencia a la sección **Implementación** donde se posée el pseudocódigo junto a su explicación.

El algoritmo tiene una complejidad temporal de O($fc * log(fc)$), siendo f = cantidad de filas y c = cantidad de columnas, del mapa. Por lo tanto cumple con la restricción de complejidad del enunciado.

Esta complejidad es dada por los siguiente

- El armado de la matriz principal de Chars, es un ciclo que recorre una vez, cada posición del mapa y guarda el Char en una matriz de F*C. Esto es O(FC)
- Luego para armar cada una de las aristas, vuelve a recorrer la matriz de Char, y crea las aristas. Recorrer nuevamente la matriz es O(FC), mientras que crear la arista e insertarla en el vector es O(1). Luego este ciclo es O(FC) $*$ O(1) = O(FC).
- Chequear que haya solución, es hacer comparaciones booleanas en O(1).
- Luego Kruskal, dado que usamos una estructura DSU, la complejidad resultante es O($V+E*logV$), siendo E la cantidad de aristas y V las de vértices. Como en el peor de los casos cada posición del mapa es un nodo, entonces O(V) = O(FC), entonces O($V + ElogV$) = O($FC + E * logFC$). Pero en el peor de los casos, cada vértice esta conectado a sus 4 vecinos dentro de la matriz. Esto es, que en el peor de los casos, cada vértice tiene 4 aristas.

  $$4V >= \sum_{veV} d(v) = 2E$$

  Luego $2V >= E$, o sea. E = O($2V$). Como O($2V$) = O($2FC$), entonces O(E) = O($2FC$) Luego O($V + ElogV$) = O($FC + FC * logFC$).
- Recorrer el vector solución, en el peor caso, tiene todas las aristas, es decir, O($2FC$) = O(FC).

Finalmente, hacer dos ciclos en la misma matriz y luego hacer Kruskal y finalmente recorrer el vector solución de aristas, implica que el algoritmo tiene complejidad

O($FC + FC + 1 + FC + 2FC * logFC$) = O($FC + FC * logFC$) = O($FC * logFC$)

Es decir, la complejidad final del algoritmo es O($FC * logFC$).

3.4. Complejidad espacial

Las distintas estructuras que forman el algoritmo propuesto con sus complejidades espaciales son,

- Matriz para almacenar los Char pasados por parametro O(FC).
- Vector de aristas, en donde el peor de los casos, el tamaño de vector es O($2FC$) = O(FC), esto significa que el grafo, es denso.
- Para Kruskal se usa la estructura **DSU**, que en el fondo, lo único que hace es crear árboles de los vértices, por lo tanto, esta estructura es O(FC).
- Luego el vector solución seguro es menor a O($2FC$), pues esa es la cota máxima de aristas que puede tener el grafo.

Finalmente la complejidad espacial es O(FC).

3.5. Experimentacion

Ya explicada las complejidad e implementación, se pasará a hacer pruebas experimentales para verificar que el tiempo de ejecución en el peor caso, tiende a O($FC * log(FC)$), la complejidad teórica demostrada en la sección anterior, siendo F las filas del mapa y C sus columnas, o visto de otra manera, FC, son en el peor caso, todos los nodos del grafo. Dado que la complejidad depende directamente del tamaño del mapa, se generaros distintas experimentaciones. Por un lado se experimentó, con casos de mapas particulares de distintos tamaños, y conocidos. Mientras que por otro lado, se hizo lo mismo pero con casos random. Luego se midieron los tiempos de procesamiento en ns, y se comparó con relación al tamaño del mapa, pasado como parámetro de entrada. Para los casos random se usó la librería Random de **STL**, con distribución uniforme.

3.5.1. Casos Fijos

Para casos fijos, se buscó simular el peor, mejor y caso promedio. Para esto, se generaron tres tipos de mapa de prueba. Luego, por cada tipo de mapa, se generó 500 matrices (o mapas) de distinto tamaño, de 1x1 hasta 500x500. Luego para cada caso se corrió por cada tamaño de mapa, 50 veces el algoritmo, y se sacó un promedio del tiempo. Los tipos de mapas son los siguientes

- Mapa donde hay algunas paredes indestructibles, otras que sí son posibles de destruir y por último algunos sectores accesibles con un ".".
- Mapa donde todas sus paredes son indestructibles.
- Mapa donde son todos ".".

Cabe destacar, que, en el caso donde todas las paredes son indestructibles, dado que el algoritmo genera aristas entre nodos destructibles o ".", para este caso la cantidad de nodos es nula. Por lo tanto, la única complejidad que pesa para este caso, es la de recorrer el mapa y generar una matriz, es decir O FC. Y luego volver a recorrer esta matriz, para crear aristas. Pero como ninguna arista es creada, Kruskal no se ejecuta y el resultado es devuelto. Luego, a este caso lo denominaremos el mejor caso.

Por otro lado, el caso donde dentro del mapa, algunas paredes son destructibles y otras no, vendría a ser nuestro caso promedio, pues el algoritmo debe correr Kruskal para ciertos nodos. Por último, nuestro peor caso, es donde no hay paredes destructibles e indestructibles (sin contar los bordes). Pues para este caso, se creará un nodo por ".", y una arista por cada par de ".". Es decir, para este caso, el grafo generado sería el mas denso. Luego el algoritmo de Kruskal, será quien tenga que ordenar una mayor cantidad de aristas y correr con estas.

En la siguiente figura, se puede observar los resultados de correr la experimentación explicada,

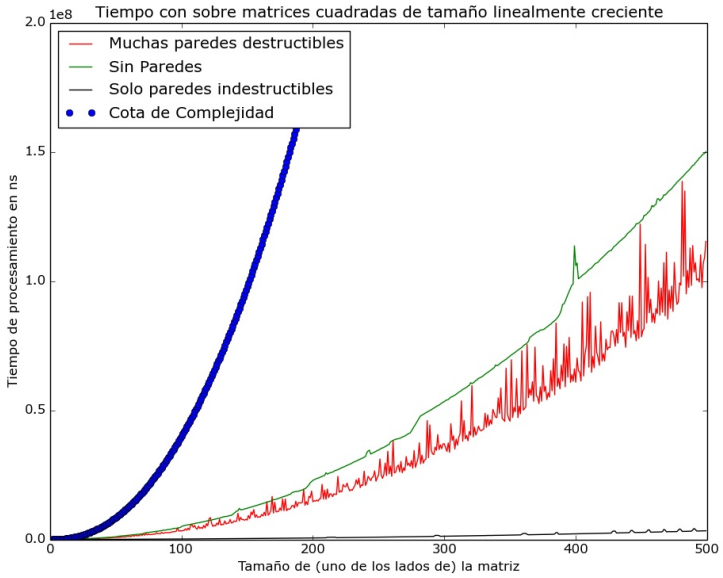

Figura 7

Luego, se puede observar, que el algoritmo cumple con la complejidad O($FC * log(FC)$). Por otro lado se pueden observar los 3 casos mencionados. Donde el caso en donde no debe generar aristas, ni encontrar el **AGM**, es aquel con menor complejidad. Por otro lado, el caso donde ciertas paredes son destructibles y otras no, tiene cierta varianza en cuanto a su tiempo de ejecución. Esto se debe a que hay ciertos casos en donde la cantidad de paredes indestructibles, son casi nulas, y por esto tiende a parecerse al mejor caso, donde no hay paredes indestructibles. Finalmente, el peor caso, es decir, aquél que mayor cantidad de aristas debe generar, es aquel donde su ejecución fue más lenta. Esto se debe por sobre todas las cosas, a Kruskal. Pues en este caso, se genera un grafo denso, o sea, la cantidad de aristas a ordenar, como así también encontrar el **AGM**, se vuelve más complejo.

3.5.2. Casos Random

Al igual que la experimentación con casos fijos, para la siguiente experimentación, se buscó generar mapas random, de tamaños entre 1x1 a 500x500. Por cada tamaño de mapa se corrió 50 veces el algoritmo, finalmente se sacó un promedio y los resultados fueron los siguientes,

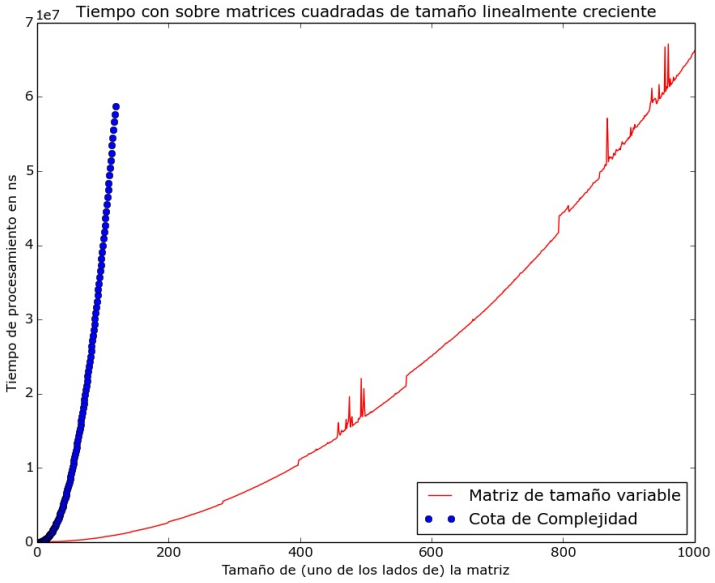

Figura 8

Al igual que en el experimento anterior, existen ciertos casos que pueden volver el algoritmo más lento con respecto a otros casos, dado a que es muy diferente el caso donde no hay caminos accesibles con ".", y donde son todos ".". Pues mientras el primero no contiene aristas, del segundo, se genera un grafo denso. Es por esto que en ciertos valores del gráfico, se notan ciertos crecimientos del gráfico. De este último experimento, se puede verificar que la cota asintótica es efectivamente $O(FC * log(FC))$, de hecho el gráfico muestra que el tiempo de ejecución es menor a la cota de complejidad $O(FC * log(FC))$.

4. Ejercicio 3: Escapando

Figura 9: El tren se va

4.1. Descripción del problema

Luego de colectar todas las piezas Indy se encuentra conforme con lo obtenido, aunque todavía no comprende lo que dice, el material encontrado lo ayudará a conseguir subsidios para continuar con la investigación.

Ya se encuentran en el lugar donde estaba la cruz en el mapa, hay unos carritos sobre vias que parecen dirigirse hacia afuera de la fortaleza. De repente se escuchan unos ruidos muy fuertes desde adentro del laberinto. Luego de romper todas las paredes internas buscando las partes de la tabla se dieron cuenta que se esta desplomando toda la estructura, por lo que deben hacer un escape rápido.

Al lado del carrito se encuentra un gráfico con la red de las vias. Hay puntos numerados que parecen indicar lugares donde los carritos pueden hacer paradas (estaciones). La estación 1 es donde se encuentran, y la última estación es donde quieren llegar. Las estaciones estan unidas con vías. Al lado de cada vía hay flechas con un número indicando el tiempo que tarda en moverse entre dos estaciones.

Ayudemos a escapar cuanto antes a todos antes de que mueran aplastados.

En resumen...

Dado una cantidad de estaciones y una cantidad de vías que cada una conecta un par de estaciones con cierta duración, se debe encontrar la forma más rápida de llegar de la primera a la última estación.

Formato de entrada: El formato de entrada contiene dos numeros. La cantidad de estaciones N, y la cantidad de vıas M. Luego siguen M lıneas, cada una contiene tres enteros A, B y C que indican que ir de A a B tarda C segundos. Las estaciones estan númeradas de 1 a N.

```
N M
A_1 B_1 C_1
...
A_M B_M C_M
```

Formato de salida: La primera lınea deberá tener un entero T indicando el mınimo tiempo para ir de la estación 1 a la estación N. Si no es posible, devolver 1. En caso de que sea posible, la segunda lınea deberá tener un entero S indicando la cantidad de estaciones que debe recorrer y la tercera lınea contendrá S enteros que indican la forma de escapar lo más rápido posible.

```
T
S
E_1 E_2 ... E_S
```

Ejemplos:
1) Una posible entrada válida sería:

```
N = 5   M = 5
[A B C] = { 1 2 1, 2 3 4, 3 4 5, 4 5 3, 2 4 2 }
```

Y su salida válida sería:

```
T = 6   S = 4
E = { 1 2 4 5 }
```

2) Otra posible entrada válida sería:

```
N = 6   M = 9
[A B C] = { 1 2 8, 1 3 4, 2 4 2, 2 5 1, 3 5 1, 4 5 1, 5 4 10, 4 6 1, 5 6 7 }
```

Y su salida válida sería:

```
T = 11  S = 4
E = { 1 2 4 6 }
```

3) Otra posible entrada válida sería:

```
N = 3   M = 2
[A B C] = { 1 2 1, 2 1 1 }
```

Y su salida válida sería:

```
T = -1
```

4.2. Solución propuesta

Como solución decidimos modelar el problema en un grafo y usar el *Algoritmo de Dijkstra* para obtener el camino mínimo de manera tal que no supere la complejidad pedida.

Las estaciones y los recorridos serán modelados en un grafo en el cual los vértices representarán a las estaciones, las aristas representarán a las vías, y el peso de cada arista representará al tiempo que se tarda en llegar de una estación a la otra.

Este grafo será implementado con una lista de adyacencia. Es decir, una lista de N posiciones, siendo N la cantidad de estaciones. Cada posición i tendrá una lista con las estaciones vecinas alcanzables desde la estación i. Cada estación vecina j será una tupla de dos valores: el número de la estación j y el tiempo que se tarda en llegar desde la estación i a la estación j.

```
                                                                          ▷ O(m)
function GRAFO LISTADY(Int estaciones, Int vias, Lista(estacion1, estacion2, distancia) recorridos)
    listAdy ← estaciones ∗ {}                                             ▷ O(1)
    for rec ∈ recorridos do                                               ▷ O(m)
        listAdy_{rec.estacion1}.agregar((rec.estacion2, rec.tiempo))      ▷ O(1)
    end for
    return listAdy
end function
```

Luego se implementará el algoritmo de Dijkstra sobre esta lista de adyacencia. Es decir, siendo N la cantidad de estaciones, el algoritmo resolverá de manera eficiente como llegar de la estación 1 a la N de la forma más rápida.

El algoritmo de Dijkstra recorre los nodos en el orden de distancia al origen. El siguiente nodo a visitar lo elige como el más cercano al origen que no haya sido visitado anteriormente. Y al visitar un nodo actualiza la distancia de sus vecinos y sus respectivos precedentes.

```
                                                                  ▷ O((n+m) * log n)
function DIJKSTRA(Grafo listAdy, Int cantEstaciones)
    time ← cantEstaciones * ∞                                     ▷ O(n)
    antecesores ← cantEstaciones * ∞                              ▷ O(n)
    noVisitados ← ColaDePrioridad((tiempo, estacion))             ▷ O(log n)
    time_0 ← 0                                                    ▷ O(1)
    noVisitados.encolar((0, 0))                                   ▷ O(log n)
    while noVisitados tenga elementos do                          ▷ O((n+m) * log n)
        estacion ← noVisitados.extraerMin()                       ▷ O(log n)
        for vecino ∈ listAdy[estacion] do                         ▷ O(m * log n)
            if time_vecino > time_estacion + tiempo_vecino,estacion then   ▷ O(1)
                nuevoTiempo ← noVisitados.actualizarPrioridad()   ▷ O(log n)
                time_vecino ← nuevoTiempo                         ▷ O(1)
                antecesores_vecino ← estacion                     ▷ O(1)
            end if
        end for
    end while
    return time_cantEstaciones, obtenerSalida(antecesores)        ▷ O(n)
end function
```

Para este caso en particular, el algoritmo usará un árbol binario para almacenar los nodos no visitados. De esta manera, se podrá extraer el nodo más cercano al origen y actualizar la prioridad de un nodo cuando se encuentre una distancia menor en complejidad logarítmica.

Para que la función devuelva lo pedido, se obtiene el tiempo mínimo que se tarda en llegar de la estación inicial a la estación final con el valor de la última posición de la lista *time*. Y por otro lado, a partir de la lista *antecesores* se obtiene el recorrido más rápido, con sus estaciones.

El problema planteado se puede visualizar como un contexto en el cual se tiene un grafo con varios vértices (estaciones), con distintas aristas (vías), cada una con su peso (tiempo en llegar de una estación a la otra). Y a partir de este grafo, se desea obtener el camino mínimo desde el primer nodo hasta el último.

Según los algoritmos vistos en la cátedra, la mejor solución para este tipo de problemas es aplicar a ese grafo el *Algoritmo de Dijkstra*, el cual resuelve eficientemente el camino con menos peso para llegar de cierto nodo inicial a cierto nodo final.

Dado este análisis del problema, se llega la conclusión de que el procedimiento desarrollado es una solución eficiente que lo resuelve.

4.3. Complejidad teórica

Dada la explicación anterior, el algoritmo tiene complejidad temporal de $O((N + M) * \log N)$ donde N es la cantidad de estaciones y M es la cantidad de vías. Esto se da debido a que se usa el algoritmo de Dijkstra con una lista de prioridad implementada con un árbol binario. Notar que $O((N + M) * \log N)$ puede ser peor $O(n^2)$. Conviene usar min heap o un árbol cuando el grafo es esparso, es decir, cuando M es mucho menor que N^2.

4.4. Experimentación

Al igual que con los otros dos ejercicios, se realizaron pruebas experimentales para verificar que el tiempo de ejecución del algoritmo cumpliera con la cota asintótica de $O((N + M) * \log N)$, teóricamente demostrada para el peor caso...

Se plantearon los siguientes casos relevantes para verificar su funcionamiento:

- Se corrieron casos aleatorios alterando la cantidad de vértices, aristas y pesos de cada arista.

En la siguiente figura, se puede observar los resultados de correr la experimentación explicada. Se muestra el tiempo de procesamiento en función de los vértices...

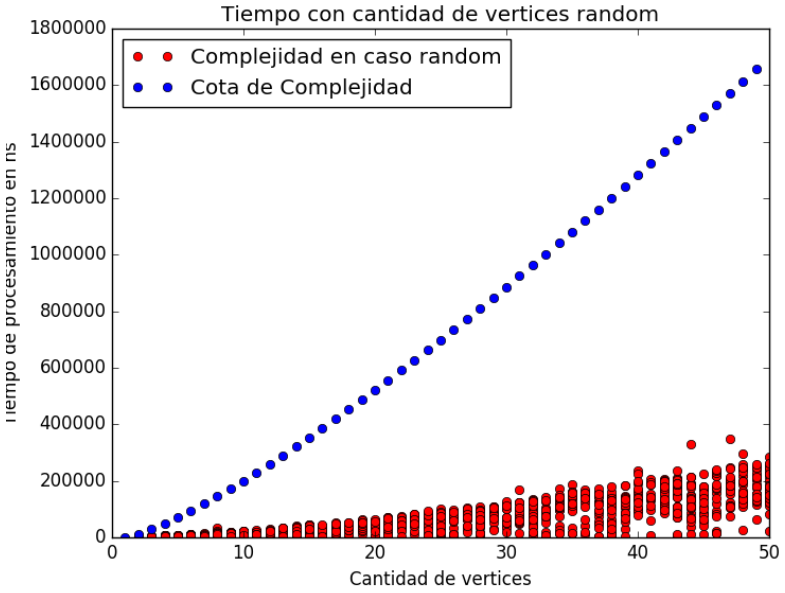

Figura 10

En la siguiente figura, se puede observar los resultados de correr la experimentación explicada. Se muestra el tiempo de procesamiento en función de las aristas...

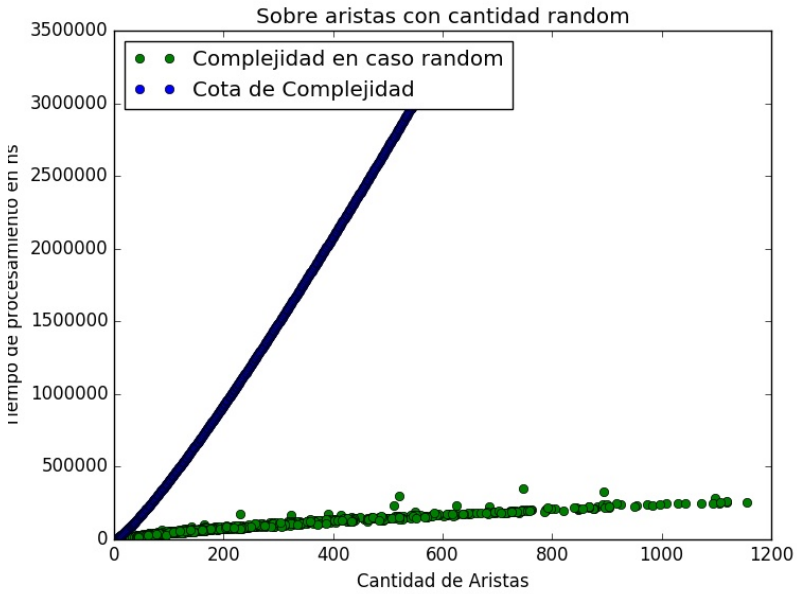

Figura 11

Se puede visualizar que los resultados obtenidos en los experimentos con entradas aleatorias difieren mucho con respecto a la cota de complejidad teórica. Esto se debe a que los casos no fueron suficientemente grandes como para alcanzarla. También se puede observar en el gráfico que el tiempo de ejecución no depende únicamente de la cantidad de vértices dado que un grafo puede ser muy denso y esto daría al Algoritmo de Disjktra muchas más posibilidades para recorrer que en el caso en el que hayan pocas aristas. Por eso en el gráfico de tiempo en función de aristas se ven los puntos agrupados en una pendiente creciente, y en el gráfico de tiempo en función de vértices puntos más dispersos.

- Se corrieron casos particulares en los que se debería alcanzar la cota de complejidad. Es decir, los peores casos. Para esto, se probó con grafos de n vértices. Para cada vértice i ($1 \leq i < n$), existe todas las posibles aristas (i, j) tal que j pertenece al intervalo (i, n). Y luego un arista (n-1, n). De esta manera el algoritmo tendrá que recorrer todos los vértices para obtener el camino mínimo.

- Se corrieron casos particulares en los que la complejidad debería ser la mínima posible, es decir, los mejores casos. Para esto, se probó con grafos en los que únicamente existía un arista que unía el primer vértice con el último.

En la siguiente figura, se puede observar los resultados de correr la experimentación explicada...

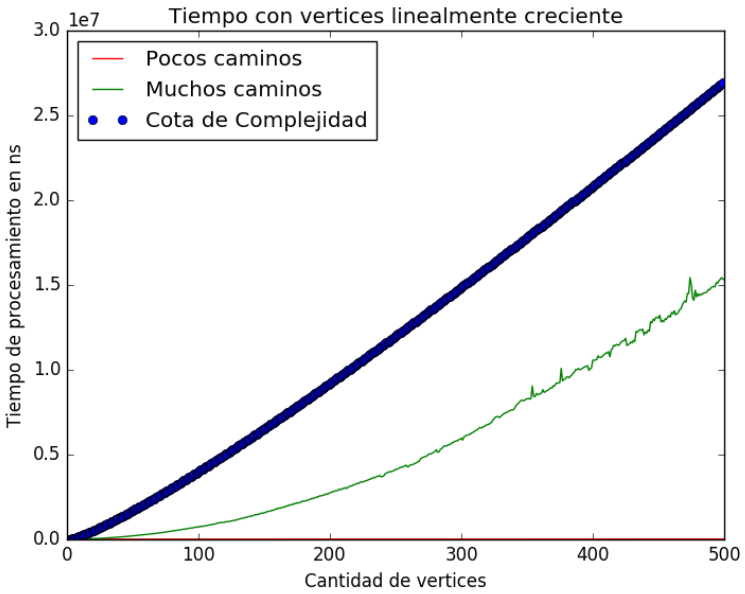

Figura 12

Se puede visualizar para los peores casos una curva creciente que tiende a la cota de complejidad. Esto se debe a que el Algoritmo de Dijkstra recorre todos los caminos posibles para llegar al último vértice. En cambio en el peor caso, como existe una única arista que recorrer, el Algoritmo de Dijkstra encuentra trivialmente el único caso posible. En este caso, lo único que le pesa a la complejidad es la construcción de la lista de adyacencia.

Cada uno de estos casos consta de entre 500 y 1000 entradas diferentes, y para cada una de esas entradas, se ejecuta el algoritmo 50 veces. En cada una de estas 1000 pruebas se calcula el tiempo promedio (el tiempo de cada ejecución, sobre 50).

Conclusión: Como se podrá observar realmente el algoritmo para diversos casos cumple la cota dada en la sección de complejidad teórica. También se puede observar que el algoritmo de Dijkstra lo que chequea son las aristas, es decir, la diferencia de la cantidad de aristas es lo que realmente pesa en el tiempo de ejecución para el algoritmo.