DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II: Indiana Jones

Algoritmos y Estructuras de Datos III

Segundo Cuatrimestre de 2016

Integrante	LU	Correo electrónico
Juan Cruz Basso	627/14	jcbasso95@gmail.com
Brian Bohe	706/14	brianbohe@gmail.com
Francisco Figari	719/14	figafran@gmail.com
Ignacio Mariotti	651/14	mariotti.ignacio@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 presenta la resolución de tres problemas aplicando técnicas algorítmicas estudiadas en la materia. Se demostró que las implementaciones sean correctas y se justificó su complejidad temporal. Se experimentó en cada caso para corroborar que se cumpliera la complejidad justificada previamente.

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

1. Sólo se midió el costo temporal de generar la solución, no de lectura y escritura del problema.
2. Todas las pruebas se ejecutaron en los laboratorios de la facultad.

2. Ejercicio 1

2.1. Descripción del problema

En este problema Indy encuentra un mapa que parece un laberinto, pero compuesto de distintos cuartos. El mapa tiene una x marcada y como reconoce donde están él y su expedición, se propone ir hacia allí. Pero como están cansados, quieren recorrer la menor distancia posible y no esforzarse mucho rompiendo paredes. Se busca entonces encontrar la forma de llegar hasta la x caminando lo menos posible y rompiendo a lo sumo cierta cantidad de paredes.

Para resolver este problema se eligió representar el mapa como un digrafo donde cada nodo es una posición del mapa y tiene aristas salientes hacia las posiciones a las cuales se puede mover. Deben tenerse en cuenta algunos detalles:

- Los movimientos dentro del mapa son sólo horizontales o verticales. Esto implica que en el digrafo con el cual se represente el problema los nodos tendrán a lo sumo cuatro vecinos salientes.
- Algunos nodos corresponden a paredes. Al momento de construir el digrafo debe tenerse en cuenta cuales nodos son paredes y cuales no.
- Más allá del límite de paredes, romper una parede no tiene costo en cuanto al movimiento, por lo que todos los movimientos cuestan lo mismo. Se puede considerar entonces que todos los arcos tienen peso 1.

Una vez construido el digrafo correspondiente, el problema se reduce a encontrar, si es que existe, el camino mínimo entre el origen y el destino tal que se rompan a lo sumo la cantidad de paredes indicada.

2.2. Solución propuesta

La solución propuesta se basa en representar el problema con un digrafo adecuado, de manera tal que luego simplemente se recorra el digrafo con BFS para encontrar el camino mínimo. Para lograr esto el digrafo debe cumplir las condiciones de BFS (aristas sin peso), pero también debe representar fielmente el problema que se busca resolver.

Este digrafo tiene $(P + 1)$ pisos de $F \cdot C$ nodos cada uno ($\mathcal{O}(F \cdot C \cdot P)$ nodos en total). Sea G el digrafo y $G(i, j, k)$ el nodo correspondiente al de la posición (i, j) del piso k. En cada piso k están los nodos correspondientes a romper k paredes. Es decir que visitar $G(i, j, k)$ equivale a alcanzar la posición (i, j) del mapa habiendo roto k paredes. Si a cada $G(i, j, k)$ se le dan los vecinos adecuados, G contendrá todas las posibles formas de recorrer el mapa.

La demostración de correctidud será una demostración constructiva, pues basta ver que G se construye adecuadamente. Dada una posición (i, j) existen cuatro movimientos posibles: moverse horizontalmente hacia izquierda o derecha, o moverse verticalmente hacia arriba o abajo. Obviamente, estos movimientos quedan restringidos si la posición en cuestión está en algún borde del mapa: si corresponde a una esquina tendrá dos movimientos posibles, mientras que si está en un costado tendrá tres. Pero no sólo se quiere simular recorrer el mapa, sino que también se quiere tener en cuenta haber roto paredes para llegar a alguna posición. Para esto se modifican los posibles movimientos recién explicados para tener en cuenta las paredes.

Suponer que la posición (i, j) en la cual se está parado se encuentra el en piso k con $k < P$ y que moverse a (i', j') necesita romper una pared. En este caso $G(i, j, k)$ no tendrá una arista hacia $G(i', j', k)$ pero hacia $G(i', j', k+1)$. Por otro lado, si ya se rompieron P paredes, i.e. $k = P$, no se pueden romper más paredes. Esto provoca no poder moverse desde la posición (i, j) a la posición (i', j'), pues se estarían rompiendo más paredes de las que se puede. Por lo tanto en está situación no se agrega la arista entre $G(i, j, k)$ y $G(i', j', k + 1)$. Si (i', j') no es una pared, simplemente se agrega la arista hacia $G(i', j', k)$. Siguiendo estas reglas se capturan todos los movimientos posibles desde un nodo, ya que cualquier otro movimiento rompería las reglas del problema.

Entonces dado un nodo correspondiente a la posición (i, j) y a haber roto k paredes, se está agregando una arista hacia cada nodo al cual se puede mover. Como esta construcción se hace para toda posición (i, j) y todo piso k, resulta que el grafo G contiene todos caminos posibles que pueden realizarse dentro del mapa, teniendo en cuenta romper paredes.

Debe destacarse un detalle de la construcción de G. Suponer que el nodo $G(i, j, k)$ se conecta a $G(i', j', k)$ pero que en la posición (i, j) hay una pared. En este caso al $G(i', j', k)$ tendrá una arista hacia $G(i, j, k+1)$. Esta última arista indicaría que se alcanza (i, j) rompiendo $k+1$ paredes cuando en realidad esto no sucede. Habrá entones caminos extras en el grafo. Sin embargo no presenta problemas: este nuevo camino no es ninguno de los caminos mínimos, pues estaríamos repitiendo posiciones del mapa. En particular se estarían formando ciclos. Por lo tanto estos caminos extras que se generan no molestan al momento. Este problema se detalla en la sección de experimentación.

Una vez construido dicho grafo, alcanza recorrerlo con BFS para encotrar el camino mínimo al destino. Sin embargo hay que tener en cuenta que en este caso no se tiene un sólo destino, sino que cualquier nodo correspondiente a la posición (i, j) será un destino posible, sin importar el piso en el cual se encuentre este nodo. Entonces luego de aplicar BFS basta mirar a que distancia se encuentra cada uno de estos nodos destino para saber, si es que existe, la solución al problema.

Implementación y complejidad teórica

La implementación de la solución propuesta se realizó en `C++` y consiste en dos etapas: inicializar el grafo que representa el problema; aplicar luego BFS para encontra el camino mínimo. Los nodos serán simplemente numerados desde 0 hasta la cantidad total de nodos. El grafo es representado con un vector de listas de nodos, que tiene en la *i-ésima* posición la lista de vecinos del *i-ésimo* nodo. Se tiene además un vector de distancias, donde en la *i-ésima* posición se guarda la distancia del *i-ésimo* nodo al origen. Este último vector debe inicializarse en -1, valor que permite saber si un nodo fue visitado. Se guarda además como variable al nodo que es origen, y un vector con los nodos correspondientes a los $P + 1$ destinos posibles.

Debe tenerse en cuenta que los bordes del mapa no son transitables. Si bien no es necesario tenerlas en cuenta, estas posiciones son consideradas pues simplifica la implementación. Estos nodos simplemente no tendrán vecinos.

El grafo G tiene entonces $\mathcal{O}(F \cdot C \cdot P)$ nodos. Para armar G se debe agregar a cada nodo los vecinos correspondientes. Para saber cuales, hay que verificar si las posiciones adyacentes son paredes. También hay que verificar si el nodo en cuestión corresponde a un borde del mapa. Dado que todas estas operaciones tienen costo constante y que cada nodo tiene a lo sumo cuatro vecinos, resulta que la inicialización de un nodo es $\mathcal{O}(1)$. Además debe preguntarse si el nodo en

cuestión corresponde al origen o a alguno de los destinos, lo que también cuesta $\mathcal{O}(1)$.

Para realizar la inicialización alcanza entonces con recorrer cada posición del mapa de cada piso e inicializar el nodo correspondiente. Mirando el pseudocódigo 1 resulta evidente que la complejidad de la inicialización es $\mathcal{O}(F \cdot C \cdot P)$.

Algorithm 1 Inicialización

```
1:  procedure INICIALIZACION(mapa)
2:      for k = 0 : P do                          ▷ Este ciclo itera O(P) veces
3:          for i = 0 : F − 1 do                  ▷ Este ciclo itera O(F) veces
4:              for j = 0 : C − 1 do              ▷ Este ciclo itera O(C) veces
5:                  nodo ← j + C · i + C · F · k                    ▷ O(1)
6:                  distancia[nodo] ← −1                            ▷ O(1)
7:              end for
8:          end for
9:      end for
10:     for k = 0 : P do                          ▷ Este ciclo itera O(P) veces
11:         for i = 1 : F − 2 do                  ▷ Este ciclo itera O(F) veces
12:             for j = 1 : C − 2 do              ▷ Este ciclo itera O(C) veces
13:                 nodo ← j + C · i + C · F · k                    ▷ O(1)
14:                 if k == 0 ∧ mapa[i][j] == 'o' then              ▷ O(1)
15:                     origen ← nodo                               ▷ O(1)
16:                 end if
17:                 if mapa[i][j] == 'x' then                       ▷ O(1)
18:                     destinos[k] ← nodo                          ▷ O(1)
19:                 end if
20:                 INICIALIZARNODO(nodo)                           ▷ O(1)
21:             end for
22:         end for
23:     end for
24: end procedure
```

Una vez construído el grafo, se lo debe recorrer con BFS para encontrar los largos de los caminos mínimos desde el origen hasta los posibles destinos. Recorrer un grafo $G = (V, E)$ con BFS tiene complejidad $\mathcal{O}(|V| + |E|)$. Sin embargo, en el grafo particular G que representa el problema a resolver, cada nodo tiene a lo sumo cuatro vecinos salientes (los cuatro movimientos posibles). Por lo tanto, en este caso $|E| = \mathcal{O}(4 \cdot |V|)$. Entonces recorrer este G particular con BFS tiene complejidad $\mathcal{O}(|V| + 4 \cdot |V|) = \mathcal{O}(|V|) = \mathcal{O}(F \cdot C \cdot P)$. El pseudocódigo 2 detalla esto.

Algorithm 2 Búsqueda del camino mínimo (con BFS)

```
1: procedure caminoMinimo
2:     Inicializar cola
3:     cola.push(origen)
4:     distancia[origen] ← 0
5:     while ¬ cola.empty do
6:         nodo ← cola.pop()
7:         for v in vecinos de nodo do
8:             if distancia[v] = −1 then
9:                 distancia[v] ← distancia[nodo] + 1
10:                cola.push(v)
11:            end if
12:        end for
13:    end while
14: end procedure
```

2.3. Experimentación

Complejidad

Esta sección busca mostrar que efectivamente el algoritmo tiene complejidad $\mathcal{O}(F \cdot C \cdot P)$. Para esto realizaremos dos análisis, en uno fijando el límite de paredes a romper e iterando sobre el tamaño del mapa, y en el segundo realizando lo inverso. En ambos casos se utilizan instancias generadas aleatoriamente.

Para el primer análisis se crean instancias con mapas cuadrados de lado creciente, con una límite de paredes a romper fijo. Como una de las variables se transforma en constante, queremos ver que los tiempos de corrida están acotados por $c \cdot i^2$, donde c es una constante e i es el tamaño del mapa.

Para el segundo análisis se crea una mapa de tamaño fijo, y luego se toman los tiempos de la misma instancia pero con distinto límite de paredes. En este caso se fijan dos variables (cantidad de columnas y cantidad de filas del mapa), por lo que queremos ver que los tiempos que se consiguen son linealmente proporcionales a la cantidad de paredes.

Las figuras 1 y 2 muestran que efectivamente se respetan estas cotas. En cada caso la función T corresponde a la función de complejidad del programa dejando libre la respectiva variable.

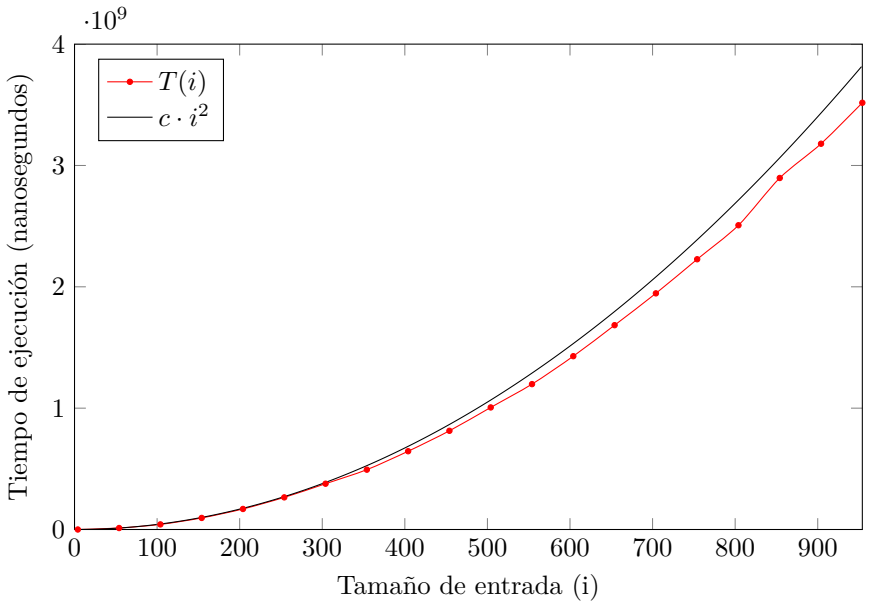

Figura 1: Complejidad fijando el límite de paredes

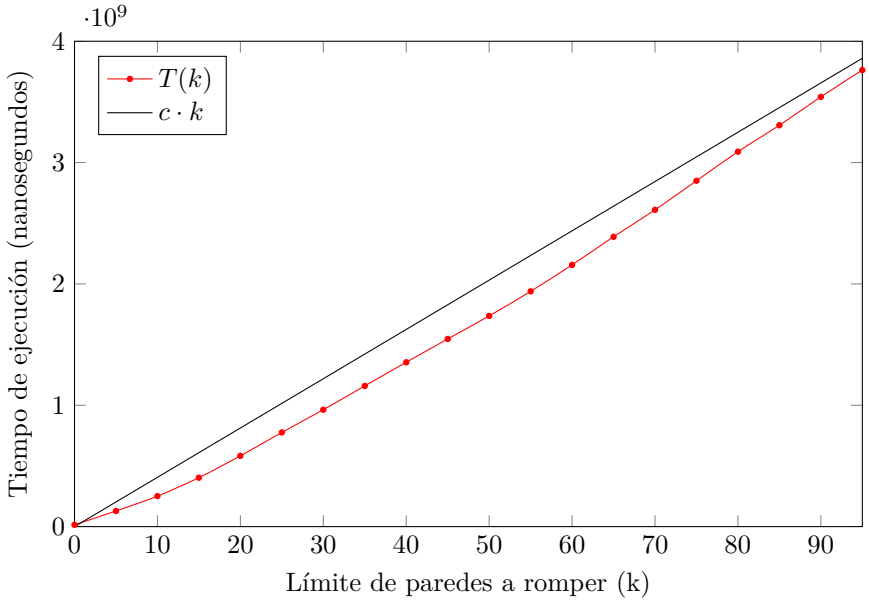

Figura 2: Complejidad fijando el tamaño del mapa

Optimización

Si bien la solución propuesta funciona y cumple la complejidad pedida, tiene dos problemas evidentes: genera todo el grafo y lo recorre por completo, sin importar si ya se visitó el destino. Cuando se utiliza BFS, una vez alcanzado el destino ya se puede cortar el recorrido, pues no habrá un camino con menor distancia que permita alcanzarlo. En este problema, ocurre lo mismo siendo la única diferencia que hay $P + 1$ nodos que son destinos. El primero que se alcance de estos sirve como solución pues cualquier otro camino a alguno de los destinos tomará un tiempo mayor o igual. La figura 3 ejemplifica esta situación.

Para solucionar esto se implementó una segunda versión del algoritmo que corrige directamente estos dos problemas. Ambas mejoras son posibles debido a como se numeran los nodos: el valor de un nodo es $nodo = j + C \cdot i + C \cdot F \cdot k$, donde i y j son las coordenadas en el mapa de la posición correspondiente y k la cantidad de paredes rotas para llegar al nodo.

La primer mejora es inicializar los nodos a medida que es necesario. Para esto no se le agregan a cada nodo sus vecinos en la inicialización del problema, sino cuando se los visita. Se modifica el BFS anterior para que realice esta inicialización. Para poder inicializar el nodo deben recuperarse las coordenadas i, j y k que le corresponden, lo cual puede lograrse con el siguiente resultado en

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

Figura 3: Caso donde la solución propuesta no es efectiva

$\mathcal{O}(1)$:

$$k = \frac{nodo}{C \cdot F}$$
$$i = \frac{nodo - C \cdot F \cdot k}{C}$$
$$j = nodo - C \cdot F \cdot k - C \cdot i$$

Dado que inicializar un nodo requiere $\mathcal{O}(1)$ operaciones totales, esta mejora no modifica la complejidad de la solución.

La segunda mejora es dejar de recorrer el grafo una vez que se alcanza alguno de los destinos. Para esto, dentro del BFS debe preguntarse al nodo en cuestión si es un destino. Una forma sencilla de hacer esto es guardar una lista con todos los nodos que son destinos, y recorrerla para ver si el nodo en cuestión es alguno de estos. Pero esto tiene complejidad $\mathcal{O}(P)$ y debe hacerse para cada nodo, por lo que empeoraría la complejidad del algoritmo. Sin embargo, si se guardan las coordenadas i_{dest} y j_{dest} del destino, se puede verificar si un nodo es destino en $\mathcal{O}(1)$ con el siguiente resultado:

$$\begin{aligned} n \text{ es un nodo de destino} &\iff n = j_{dest} + C \cdot i_{dest} + C \cdot F \cdot k \\ &\iff \frac{n - j_{dest} - C \cdot i_{dest}}{C \cdot F} = k \\ &\iff \frac{n - j_{dest} - C \cdot i_{dest}}{C \cdot F} \in \mathbb{Z} \\ &\iff n - j_{dest} - C \cdot i_{dest} \equiv 0 \pmod{C \cdot F} \end{aligned}$$

El pseudocódigo 3 muestra como queda la búsqueda del camino mínimo en la nueva solución. Por su parte, la inicialización se modifica quitando el ciclo que inicializa los vecinos de cada nodo. Dado que ambas mejoras se realizan en $\mathcal{O}(1)$ la complejidad de la solución no empeora. Pero tampoco mejora, pues si bien no se inicializan los vecinos de cada nodo, deben inicializarse su distancia en -1. La complejidad temporal es entonces la misma.

Para obtener resultados acerca de las mejoras obtenidos, se realiza una comparación en los tiempos de computos para las mismas instancias con ambas implementaciones. Dado un mapa cuadrado de lado 250 generado aleatoriamente, se hace variar la distancia a la cual se encuentran el origen y el destino La figura 4 muestra los resultados obtenidos. Se ve como la segunda implementación es efectivamente mejor: cuando destino y origen están a distancia menor a 100, la primera implementación resulta más de 10 veces más rapida. Incluso cuando la distancia es máxima (500 de distancia), la implementación optimizada otorga mejores resultados. También se puede ver como los tiempos de cómputos de la segunda implementación se acercan a los de la primera, a medida que aumenta la distancia. Esto es razonable, la ventaja de la segunda es implementación se debe a encontrar antes el destino.

Algorithm 3 Búsqueda del camino mínimo (con BFS)

```
1:  procedure caminoMinimo
2:      Inicializar cola
3:      cola.push(origen)
4:      distancia[origen] ← 0
5:      destinoVisitado ← false                                  ▷ O(1)
6:      while ¬ cola.empty ∧ ¬ destinoVisitado do
7:          nodo ← cola.pop()
8:          inicializarNodo(nodo)                                ▷ O(1)
9:          for v in vecinos de nodo do
10:             if distancia[v] = −1 then
11:                 distancia[v] ← distancia[nodo] + 1
12:                 cola.push(v)
13:                 if v es destino then                         ▷ O(1)
14:                     destinoVisitado ← true                   ▷ O(1)
15:                 end if
16:             end if
17:         end for
18:     end while
19: end procedure
```

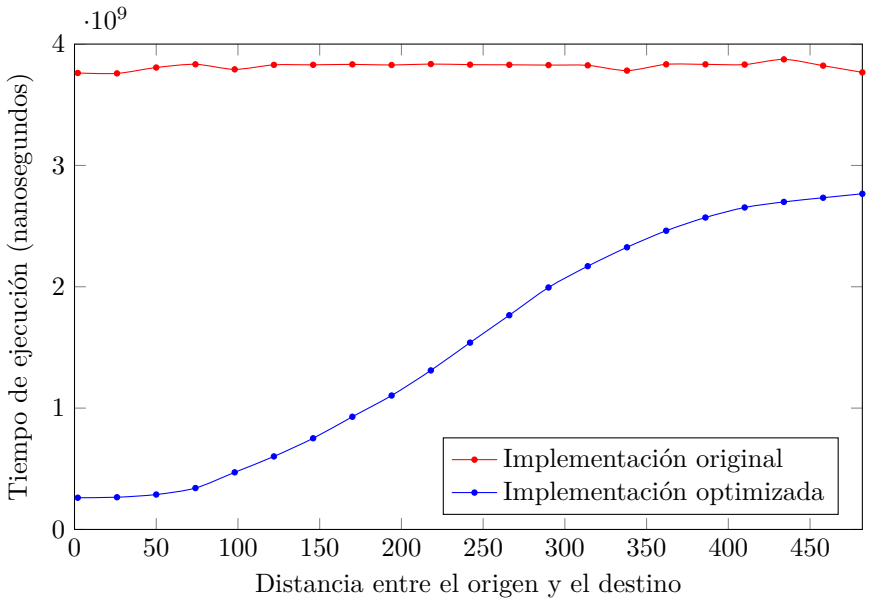

Figura 4: Comparación de tiempos entre ambas implementaciones para instancias con distancia creciente entre origen y destino

Los dos cambios que se realizaron mejoran también la cantidad de memoria utilizada por el programa. Esto se debe a que al no inicializar nodos que no es necesario, se está ahorrando

la memoria utlizada en las listas de adyacencia. La tabla 1 se ilustra esto con tres instancias donde el origen y el destino están a distancia dos. Efectivamente la segunda implementación aloca menos memoria.

	Memoria utlizada (en kilobytes alocados)	
Tamaño de la instancia	Implementación original	Implementación optimizada
120x120	27,614	6,565
200x200	77,002	18,089
500x500	484,079	112,603

Tabla 1: Comparación de memoria utilizada
en tres intancias generadas aleatoriamente

Corrección de problema original

En la página 5 se detalló un problema de la solución propuesta: se recorren caminos que sabemos que no vale la pena recorrer. En particular si un nodo en la posición (i, j) ya fue visitado rompiendo k paredes, no es necesario visitarlo de vuelta rompiendo la misma o mayor cantidad de paredes: el camino mínimo desde $G(i, j, k)$ al destino es más corto o igual que el camino desde $G(i, j, k')$ para todo k' mayor o igual a k. En consecuencia, si un nodo ya fue visitado rompiendo k paredes, no sirve explorar un camino que pase por un nodo correspondiente a la misma posición pero rompiendo mayor cantidad de paredes.

Para traducir esto en una mejora, al inicializar un nodo preguntaremos si ya fue visitado rompiendo menor o igual cantidad de paredes. Si fuera el caso, al nodo no se le agregan los vecinos que le correspodan, evitando que se exploren los caminos que parten desde este nodo. Si no, se le agregan los vecinos correspondientes como se hace en la solución original. Esto requiere tener una matrix paralela al mapa, tal que se pueda consultar en $\mathcal{O}(1)$ el k correspondiente a la menor cantidad de paredes con la cual se visitó la posición correspondiente al nodo. Si la posición no hubiera sido visitada este valor valdría -1, caso en el cual se inicializa el nodo.

A diferencia de las otras implementaciones, esta tercer solución no recorre caminos que repitan posiciones del mapa. Por lo tanto se espera que los tiempos para resolver este problema con esta solución sean mejores a los tiempos de las otras dos soluciones. Para esto se realizó un experimento análogo al de la sección anterior: se tomaron tiempos de resolviendo el mismo mapa pero con distancia entre origen y destino creciente.

En la figura 5 se ve como la tercer implementación tiene efectivamente mejores tiempos que las otras.

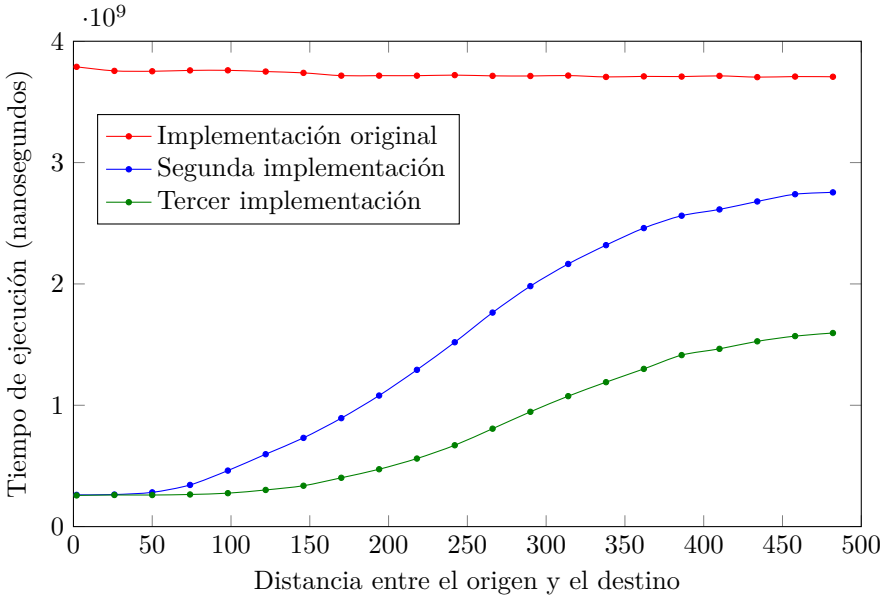

Figura 5: Comparación de tiempos entre la segunda y tercer implementación para instancias con distancia creciente entre origen y destino

Dados los resultados se puede concluir que ambos cambios realizados a la solución inicial son efectivamente mejoras. Queda pendiente analizar cada mejora de manera independiente para obtener más detalle respecto de las ventajas que otorga cada una.

3. Ejercicio 2

3.1. Descripción del problema

Indiana descubrió que en cada sala del laberinto se esconde un fragmento con un manuscrito antiguo y quiere juntarlos todos. Tiene en su poder un mapa de toda la arquitectura el cual indica cuanto esfuerzo requiere para romper cada una de las paredes existentes entre las diferentes salas, si es que se pueden romper. Quiere saber, si es que existe una forma, el esfuerzo mínimo que le tomaría llegar a cada sala para obtener los fragmentos. El mapa (ver figura 6) es una grilla de C columnas y F filas en el que se puede encontrar:

- $e_1, e_2, \cdots, e_N$ espacios que pueden recorrerse, con $N \in \mathbb{N}$.
- $p_1, p_2, \cdots, p_M$ paredes que pueden ser derribadas, con $M \in \mathbb{N}_0$.
- $k_1, k_2, \cdots, k_M$ esfuerzo asociados a las M paredes derribables, con $(\forall 1 \leq i \leq M)\ k_i \in \mathbb{N}$.
- $i_1, i_2, \cdots, i_R$ paredes que no pueden ser derribadas.

Cumple las siguientes propiedades:

- Los bordes del mapa se componen de paredes no derribables.
- Una pared derribable tiene a lo sumo dos espacios adyacentes que pueden recorrerse.
- Si hay dos paredes derribables adyacentes, se considera que aunque ambas sean derribadas, no se puede recorrer de una a la otra.
- No hay paredes derribables con más de dos paredes no derribables adyacentes.

Sabiendo que sólo se puede caminar entre celdas del mapa adyacentes, y no se pueden realizar movimientos diagonales. Se buscan los $c_1, c_2, \cdots, c_M \in \{0, 1\}$. De manera que si $c_i = 1$, la pared p_i es derribada, y si $c_i = 0$, p_i no es derribada. Que minimicen:

$$\sum_{i=1}^{M} c_i \cdot k_i \tag{1}$$

De forma tal que se puedan recorrer todos los espacios e_i.

3.2. Solución propuesta

Se puede modelar el problema utilizando un grafo simple con pesos asociados a sus aristas. Los vértices son los espacios recorribles del mapa. Por cada par de vértices adyacentes se agrega una arista de peso 0 y por cada pared derribable p_i se agrega una arista de peso k_i entre los vértices que representan los espacios recorribles adyacentes a p_i.

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

Figura 6: '.' parte de una sala. '#': pared irrompible. '1', '2' y '3' paredes derribables con esfuerzo 1, 2 y 3 respectivamente.

Observación 3.1. *Sean V y E los conjuntos de vértices y aristas del grafo. $|V|$ y $|E|$ son $\mathcal{O}(C \cdot F)$. A lo sumo hay tantos vértices como puntos del mapa, de dimensiones $C \cdot F$ y a lo sumo cada vértice tiene 4 vecinos, por lo que $|E|$ es a lo sumo $4 \cdot |V|$.*

Observación 3.2. *El grafo resultante es conexo sii existe solución al problema original. Todos los movimientos posibles del mapa son modelados en el grafo y nada más que esos. El problema original no tiene solución si existe un e_i no accesible por algún e_j. Es decir, no existe un camino desde e_i a e_j, por lo que el grafo resultante no es conexo.*

Una vez modelado como grafo, el problema se reduce a encontrar las aristas que unan todos los vértices con el menor peso posible. Es decir, encontrar un árbol generador mínimo (AGM).

3.2.1. Implementación y complejidad teórica

La implementación de la solución se realizó en `C++` y consta de dos partes: inicializar el grafo que modela el problema y aplicar luego el algoritmo de Kruskal para encontrar el *AGM* si existiera.

$G = (V, E)$ es el grafo que se utiliza para representar el problema. Se optó por representar G con una lista de aristas. Para construir G debe recorrerse el mapa original que tiene $\mathcal{O}(F \cdot C)$ posiciones. Los puntos del mapa son los nodos de G, por lo que $\mathcal{O}(|V|) = \mathcal{O}(F \cdot C)$. Por la observación 3.1 se sabe que cada nodo tiene una cantidad acotada de vecinos, por lo que inicializar un nodo es $\mathcal{O}(1)$. Entonces el costo de la inicialización es $\mathcal{O}(|V|)$.

El algoritmo de Kruskal de consite en partir de $|V|$ componentes disconexas (una por cada nodo), lo cual es un bosque. Progresivamente se van agregando aristas entre estas componentes tal que en cada paso se siga teniendo un bosque. Una forma de hacer esto es utilizar la estructura de *disjoint-set forest* en la que se representan las componentes disconexas con un arbol donde su raiz funciona como su representante. El representante de un nodo se obtiene con la función `find`. Realizar la union de componentes conexas es simplemente unir una raiz a otra como hijo, lo que se realiza con la función `union`. Verificar que una arista nueva no genera un ciclo es ver que ambos extremos de la arista no pertenezcan a la misma componente conexa.

Pueden realizarse dos optimizaciones a este algoritmo:

- Colgar el árbol de menor altura al de mayor altura, de esta manera el arbol resultante tiene una altura igual al mayor. Esto permite que los futuros llamados a `find` sean más rápidos, dado que su costo es a lo sumo la altura del árbol.
- Cada vez que se realiza un `find(`x`)` se actualiza el padre de x para disminuir la altura del arbol.

Si se utilizan estas dos optimizaciones se puede probar que realizar $\mathcal{O}(|V|)$ operaciones `union` o `find` es $\mathcal{O}(|V| \cdot log^*|V|) = \mathcal{O}(|V| \cdot log|V|)$. Inicializar las componentes conexas para Kruskal tiene complejidad $\mathcal{O}(|V|)$ y ordenar las aristas tiene costo $\big|E| \cdot log|E|$. Entonces la complejidad final de aplicar Kruskal es $\mathcal{O}(|V| + |V| \cdot log|V| + |E| \cdot log|E|) = \mathcal{O}(|V| \cdot log|V| + |E| \cdot log|E|)$ El pseudocódigo 4 corresponde al de este algoritmo.

G tiene el atributo cantidad_nodos que indica la cantidad de nodos del grafo, y cantidad_aristas_AGM que indica la cantidad de aristas del *AGM* obtenido hasta el momento.

Algorithm 4 costoMinimo

```
1: procedure ALGORITMOKRUSKAL(grafo G)
2:     res ← 0                          ▷ En esta variable se guarda el peso total del árbol solución
3:     G.cantidad_aristas_AGM ← 0
4:     Inicializar componentes conexas
5:     SORT(G.lista_aristas)                                   ▷ Se ordenan las aristas por peso
6:     for (u,v) in G.lista_aristas do
7:         if FIND(u) ≠ FIND(v) then
8:             UNION(u,v)
9:             res ← res + (u, v).costo
10:            G.cantidad_aristas_AGM ← G.cantidad_aristas_AGM + 1
11:        end if
12:    end for
13:    return res
14: end procedure
```

Una vez que se aplica el algoritmo de Kruskal, basta verificar la cantidad de aristas del árbol otenido para saber si se encontró solución:

- Si tiene menos de $|V| - 1$ aristas resulta que no hay solución, por lo que se devuelve -1
- Si tiene $|V| - 1$ aristas el resultado es efectivamente un árbol generador mínimo, por lo que se devuelve su costo.

Esto es exactamente lo que detalla el algoritmo 5.

Algorithm 5 costoMinimo

```
1: procedure COSTOMINIMO(grafo G)
2:     costo_minimo ← ALGORITMOKRUSKAL(G)
3:     if |V| − 1 ≠ G.cant_aristas_AGM then
4:         costo_minimo ← −1
5:     end if
6:     return costo_minimo
7: end procedure
```

La complejidad de la solución propuesta es sumar el costo de la inicialización y de aplicar Kruskal, que es $\mathcal{O}(|V| + |V| \cdot log|V| + |E| \cdot log|E|) = \mathcal{O}(|V| \cdot log|V| + |E| \cdot log|E|)$. Como $\mathcal{O}(|E|) = \mathcal{O}(|V|)$ se deduce entonces que la complejidad final de `costoMinimo` es $\mathcal{O}(|V| \cdot log|V|) = \mathcal{O}(F \cdot Clog(F \cdot C))$.

3.3. Experimentación

Se implementó un script en python para generar instancias aleatorias del problema dados F una cantidad de filas, C una cantidad de columnas, M una cantidad de paredes destruibles e I cantidad de paredes irrompibles. Se definieron tres tipos de instancias, y para cada caso se realizaron 100 repeticiones.

- Casi no sin esfuerzo: En estos casos N ($\frac{2}{3} \cdot F \cdot C$) es considerablemente mayor a M e I ($\frac{1}{6} \cdot F \cdot C$ cada una). Se espera que al tener mayor cantidad de vértices y ejes, consiga los peores tiempos de ejecución con respecto a las siguientes instancias. Se llama T_1 al tiempo para esta instancia.
- El límite es tu mente: En estos casos M ($\frac{2}{3} \cdot F \cdot C$) es considerablemente mayor a N e I ($\frac{1}{6} \cdot F \cdot C$ cada una). Se espera que esta instancia tenga mejores tiempos que T_1. Se llama T_2 al tiempo para esta instancia.
- Los indestructibles: En estos casos I ($\frac{2}{3} \cdot F \cdot C$) es considerablemente mayor a M y N ($\frac{1}{6} \cdot F \cdot C$ cada una). Se espera que los resultados de esta instancia sean similares a T_2. Se llama T_3 al tiempo para esta instancia.

Los resultados sugieren que las hipótesis planteadas se cumplieron.

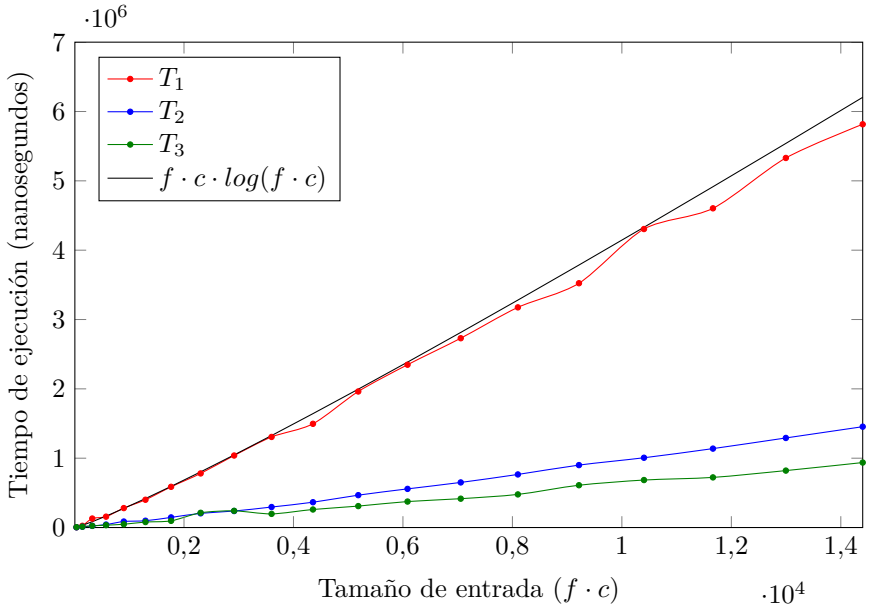

Figura 7: Comparación de tiempos

4. Ejercicio 3

4.1. Descripción del problema

Junior y su grupo de arqueólogos lograron salir del laberinto, se encuentran con un carrito sobre vías que parecen dirigirse fuera de la fortaleza. También observan un mapa que muestra el recorrido de las vías. Hay puntos numerados que parecen indicar lugares donde los carritos pueden hacer paradas (estaciones). La estación 1 corresponde a la estación en la cual encuentran y la última estación es a donde quieren llegar. Al lado de cada vía hay flechas con un número indicando el tiempo que requiere moverse entre dos estaciones. Antes de poder analizar como salir se dan cuenta que la fortaleza se está derrumbando. El objetivo del problema es ayudar a Indiana Jones a salir de la fortaleza encontrando el recorrido más rápido hacia la salida antes de terminar aplastado.

Una manera de plantear el problema es modelar el recorrido de las vías como un grafo dirigido en el que cada vertice representa una estación y cada vía un arco con peso indicando el tiempo que toma recorrerla. Luego la solución es buscar el camíno más corto entre el vértice número 1 y el último.

4.2. Solución propuesta

Dado que el tiempo que toma el viaje entre una estación y otra no puede ser negativo, una posible solución al problema es utilizar el algoritmo de camino mas corto de Dijkstra. El mismo garantiza encontrar el camino mas corto entre un vértice y todos los demás del grafo siempre y cuando los ejes del grafo sean positivos. Los pasos a seguir son los siguientes:

Notación 4.1. *Se denota $p(v, u)$ al peso asignado al arco que va de v a u.*

Notación 4.2. *Se define $\pi(u)$ al rótulo asignado al vértice u. Es un diccionario que toma un vértice y devuelve un entero. Indica la menor distancia encontrada desde el vértice inicial a u.*

Algorithm 6 Algoritmo de Dijkstra

```
1: procedure caminoMínimo
2:     for Vértice v en el grafo do
3:         π(v) ← ∞
4:     end for
5:     S ← {origen}
6:     π(origen) ← 0
7:     while S != {∅} do
8:         Tomar vértice v de S tal que π(v) sea mínimo
9:         S ← S − {v}
10:        for Vértice u sucesor de v do
11:            if π(u) != ∞ then
12:                S ← S ∪ {u}
13:            end if
14:            π(u) ← min(π(v) + p(v, u), π(u))
15:            Si π(u) cambió, guardar a v como predecesor de v.
16:        end for
17:    end while
18: end procedure
```

El conjunto S indica los nodos a visitar, i.e. el conjunto de nodos vecinos a los nodos que ya fueron fijados. Si un nodo no tiene rótulo infinito, significa que está en S o estuvo en S en algún momento, por lo que no hace falta agregarlo nuevamente. Que S esté vacío al principio del ciclo significa que se recorrió toda la componente conexa desde la que se partió y que todos nodos con rótulo infinito no son alcanzables desde el origen. Si solamente se requiere el camino hacia un nodo específico, una vez que es seleccionado como mínimo del conjunto S se puede terminar el algoritmo, aunque esto no modifica la complejidad final.

El camino mínimo se puede recuperar realizando una iteración desde el nodo destino hacia el origen o bien guardando el predecesor cada vez que se modifica el rótulo de un nodo.

Implementación y complejidad teórica

Para la implementación del algoritmo realizada en `C++` se representó el grafo dirigido utilizando un vector de n nodos y para cada nodo una lista de adyacencia, que indica los arcos que parten desde el nodo y su peso (tiempo para llegar de una estación a otra). La suma de los elementos de todas las listas es la cantidad de arcos del grafo m.

Tanto los rótulos como los antecesores se implementaron con vectores inicializados en -1, valor que representa distancia infinita. El conjunto S se implementó mediante una lista. En consecuencia la busqueda del mínimo rótulo es lineal en base a la cantidad de nodos en el conjunto y el agregado de un nuevo elemento es constante.

Para analizar la complejidad de la implementación se realizarán referencias a los números de líneas del pseudocódigo del algoritmo 6.

- El ciclo de inicialización (línea 2) itera n veces y realiza una asignación constante. Su complejidad es $\mathcal{O}(n)$.

- El ciclo principal (línea 7) itera a lo sumo n veces. Cada nodo en el grafo alcanzable desde el vértice inicial se agrega una única vez a S, y por cada iteración se elimina un elemento del conjunto. La búsqueda del mínimo en la instrucción (línea 8) y la eliminación del mismo del conjunto se realiza linealmente, por lo que su costo es $\mathcal{O}(n)$. Tomando en cuenta sólo esta operación, el costo del ciclo resulta ser $\mathcal{O}(n^2)$.
- La iteración por los vecinos del nodo seleccionado (línea 10) se realiza en total m veces ya que por cada nodo que entra en S se evalúan sus vecinos un total de una vez. Las operaciones internas al ciclo de vecinos se realizan en tiempo constante por lo que la complejidad de este ciclo es $\mathcal{O}(m)$.

La complejidad final del algoritmo resulta ser $\mathcal{O}(n^2 + m) = \mathcal{O}(n^2)$ dado que la cantidad de arcos en un grafo se puede acotar por n^2.

4.3. Experimentación

Complejidad

Se separaron distintos casos o familia de grafos para evaluar el desempeño del algoritmo. Para cada caso se generaron 50 instancias con distinta cantidad de nodos, desde 2 hasta 500. Se garantizó la existencia de al menos un camino entre el nodo origen y todos los demás pertenecientes al grafo para controlar la cantidad de nodos que participan efectivamente en el algoritmo.

- T_0: Este caso consta de $n-1$ arcos y es un camíno simple desde el nodo 1 hasta el nodo n.
- T_1: Este caso es un grafo tipo estrella. Hay un arco desde el nodo 1 hacia todos los demás.
- T_2: Tiene alrededor de $\frac{(n-1)^2}{2}$ arcos.
- T_3: Es un grafo completo, con $\frac{(n-1)\cdot n}{2}$ arcos.

También se graficó una función de la forma $c \cdot n^2$ para una constante c a modo de mostrar la cota de complejidad.

Se puede ver en la figura 8 que la complejidad teórica se cumple y que al aumentar la cantidad de arcos en el grafo incrementa el tiempo de cómputo notablemente. También se ve con mejor detalle en la figura 9 que los tiempos en el caso de camino simple se comportan similar a una función lineal. Esto se debe a que sólo se carga un nodo por vez al conjunto S, por lo que cada busqueda de mínimo se realiza en tiempo constante.

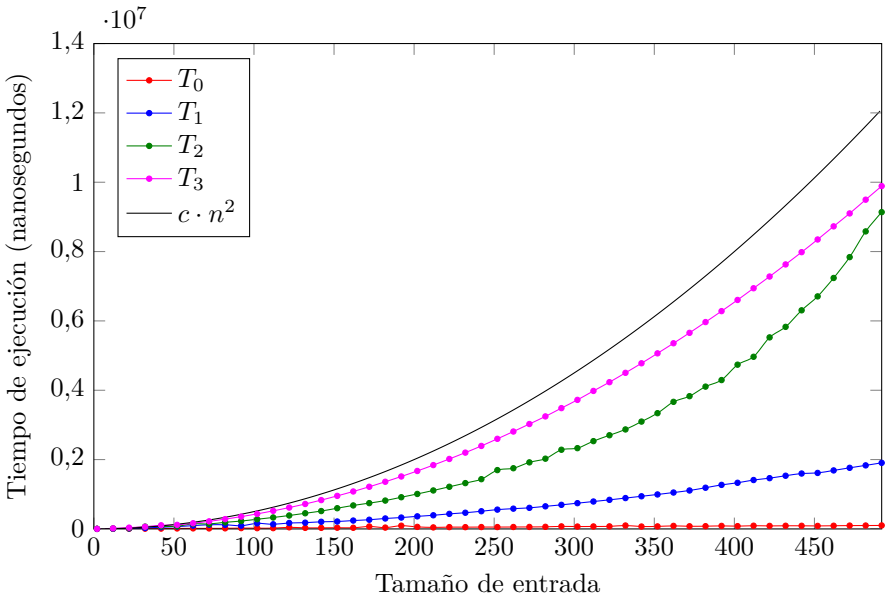

Figura 8: 100 repeticiones de camino más corto (Dijkstra) para cada instancia de los distintos casos

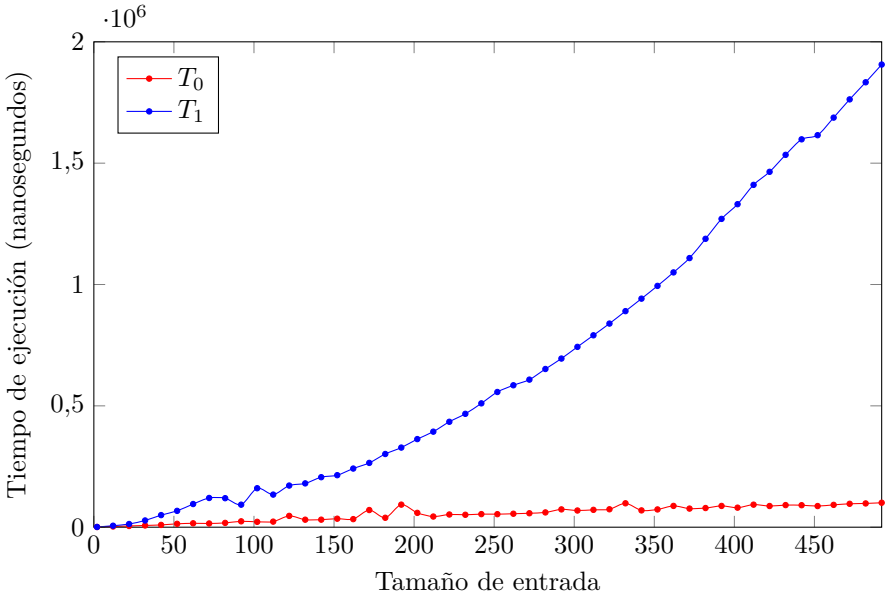

Figura 9: Comparación entre grafo camino simple y estrella