

Algoritmos y Estructura de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 2

Grupo 1

Integrante	LU	Correo electrónico
Hernandez, Nicolas	122/13	nicoh22@hotmail.com
Kapobel, Rodrigo	695/12	rok_35@live.com.ar
Rey, Esteban	657/10	estebanlucianorey@gmail.com
Tripodi, Guido	843/10	guido.tripodi@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Contents

1	Ejercicio 1	3
1.1	Descripción de problema	3
1.2	Explicación de resolución del problema	3
1.3	Algoritmos	5
1.4	Análisis de complejidades	5
1.5	Demostración de correctitud	6
1.6	Experimentos y conclusiones	7
1.6.1	1.5	7
1.6.2	1.5	9
2	Ejercicio 2	16
2.1	Descripción de problema	16
2.2	Explicación de resolución del problema	16
2.3	Algoritmos	17
2.4	Análisis de complejidades	18
2.5	Demostración de correctitud	18
2.6	Experimentos y conclusiones	19
2.6.1	2.5	19
2.6.2	2.5	22
3	Ejercicio 3	28
3.1	Descripción de problema	28
3.2	Explicación de resolución del problema	28
3.3	Algoritmos	29
3.4	Análisis de complejidades	29
3.5	Demostración de correctitud	30
3.6	Experimentos y conclusiones	30
3.6.1	2.5	30
3.6.2	2.5	32
4	Aclaraciones	38
4.1	Aclaraciones para correr las implementaciones	38

1 Ejercicio 1

1.1 Descripción de problema

Luego de haber estado llenando las mochilas, Indiana y el equipo encuentra un mapa, dicho mapa es muy parecido a un laberinto el cual indica donde estaban y una X muy curiosa, marcada en el mapa. El equipo decidió cruzar el laberinto, como cuentan con pico y pala puede romper paredes. Su único problema es el cansancio que traen auestas luego de todo lo hecho hasta ahora.

Es por esto que nuestro objetivo será brindarles el menor camino posible teniendo en cuenta la cantidad de paredes que pueden llegar a romper.

1.2 Explicación de resolución del problema

El enunciado presenta el problema de hallar el camino entre 2 puntos que se pueda recorrer en el menor tiempo posible, dándonos un escenario en donde los caminos presentan paredes que podemos atravesar rompiéndolas, teniendo a la vez, una determinada cantidad de rupturas a poder realizar. La solución buscada será entre todos los caminos posibles desde el origen al destino, el que resulte mínimo.

Cada camino esta compuesto de unas unidades "caminables", que las llamaremos baldosas. Ir de una baldosa a otra consume la misma cantidad de tiempo en todos los casos. Por otro lado, hay baldosas que demandan romper una pared para ser caminadas, con lo cual el camino que se esta transitando debe gastar 1 unidad de esfuerzo en romperla, factor que no influye en el tiempo del recorrido.

Si caracterizamos cada baldosa con un nodo, y la posibilidad de ir de una baldosa a otra como una arista, entonces podemos modelar el problema con un grafo. Como el factor tiempo es constante en cada traslado entre baldosas, las aristas tendrán todas el mismo peso.

A partir de este grafo, sacar el camino mínimo del nodo origen al destino se puede realizar utilizando el algoritmo de búsqueda por anchura, el cual permite saber la distancia (o tiempo) a la que se encuentra determinado nodo con cada uno de los demas. Para ello el BFS utiliza una cola donde introduce nodos que aún no fueron marcados como visitados, para luego extraerlos de a uno y encolar las repectivas adyacencias que aún no fueron visitadas. Cada nodo pusheado a la cola representa, de esta manera, la cabeza de cada camino que forma el BFS para formar el árbol.

Como no hay ninguna otra restricción para encolar un nodo (y así descubrir un camino), de aplicar el BFS sin modificaciones se perdería la restricción de paredes para romper: por esto se debe agregar una validación al introducir un nodo a la cola del algoritmo, que impida pushear un nodo que demande romper más paredes de las parametrizadas.

Dado que al marcar un nodo como visitado se lo esta excluyendo de ser incluido en la cola nuevamente, de haber otro camino que lo necesite utilizar, el mismo no podrá contar con él. De suceder, se podría estar desestimando caminos que lleguen a la solución: Si llamamos C_0 al primer camino que paso por el nodo v_0 con tiempo 5 y paredes rotas 4, siendo el máximo 4 y C_1 a un camino más lento, con tiempo 10 pero con 0 paredes rotas, que está pasando por v_0 , entonces en la evaluación de agregar o no al nodo se optará por no, ya que fue visitado. No obstante, si para llegar al destino final se debe romper 1 pared más, C_0 nunca podrá llegar, y C_1 que hubiese llegado, fue desestimado anteriormente, perdiendo de esta forma la solución factible.

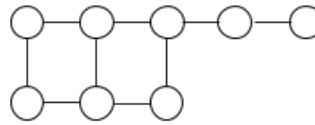
Para solucionar este problema se redefine la condición de visitado que utiliza el BFS:

Representación del problema y aplicaciones de algoritmos

Entrada : 1 pared a derribar como máximo

#	#	#	#	#	#
#	O	#	.	#	X
#	.	.	.	#	#
#	#	#	#	#	#

se representa
con

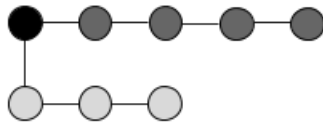


camino 1
camino 2
camino 1 y 2
origen
no visitado



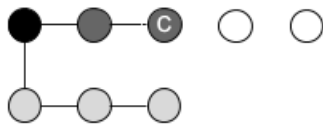
Algoritmos aplicados al grafo

BFS



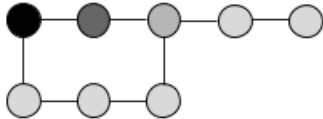
Dado que se ignora la restricción de la cantidad de paredes a romper, se genera una solución no válida

BFS + restricciones



Aplicando la restricción se deja de obtener la solución válida, pero como para el BFS el nodo "C" ya fue recorrido, entonces ningún otro camino que podría llegar a utilizarlo para llegar al destino lo tiene disponible

BFS +
restricciones +
modificación



Al permitir que un camino con menos paredes derribadas utilice un nodo, se permite alcanzar la solución requerida. A notar, el BFS modificado ya deja de generar un árbol.

Se considerará que la condición de visitado dependa de cada camino, es decir:

- Los nodos guardarán como datos propios el tiempo en el que fueron alcanzados y la cantidad de paredes que fueron destruidas para llegar a él en ese tiempo (es decir, los datos del camino que paso por él). Al inicio, ambos parámetros serán marcados como no determinados.
- Al ser chequeados por primera vez, al detectarse de que no tienen datos, entonces se les asignará el tiempo en el que fueron alcanzados (como en el algoritmo BFS original) y la cantidad de paredes derribadas para alcanzarlos.
- Si al chequearlos nuevamente se ve que ya fueron recorridos (es decir que el tiempo y paredes se encuentran determinados) entonces se evaluará si por el camino actual **se mejora la condición del alcance del nodo**, de ser así, se encola.
- En todos los casos, si al chequear un nodo, se detecta que hay que romper una pared, y el camino actual no posee más derribos posibles, entonces no se lo pushea a la cola.
- Si el nodo es el destino buscado, se lo marca con el tiempo y paredes logradas y se **finaliza el algoritmo**.

Condición de alcance del nodo

Se considera una mejor condición de alcance para un nodo si se cumple lo siguiente:

- El nodo no tiene datos determinados, es decir nunca fue pusheado a la cola.
- La cantidad de paredes para llegar a él es menor estricta que la lograda anteriormente para llegar al nodo.

1.3 Algoritmos

```

función main()
    encolo nodo origen en una cola de nodos O(1)
    Mientras  $\neg$  cola vacia hacer
        ciclo:  $O(F * C * P)$ 
        Nodo actual  $\leftarrow$  tope cola O(1)
        procesar vecinos O(4)
    fin ciclo
total:  $O(F * C * P)$ 

fin función

función procesarNodo(Nodo nod)
    Nodo nod es un vecino de actual en el Mapa
    si  $\#$  paredes rotas hasta actual + esPared(nod) <  $\#$  paredes rotas hasta actual nod
    entonces
        guarda:  $O(1)$ 
        para distancia hasta nod  $\leftarrow$  distancia hasta actual + 1 O(1)
        para  $\#$  paredes rotas hasta nod  $\leftarrow$   $\#$  paredes rotas hasta actual + esPared(nod) O(1)
        si nod = nodo destino entonces
            guarda:  $O(1)$ 
            devolver distancia hasta nodo destino y terminar O(1)
        fin si
        si  $\#$  paredes rotas hasta nod  $\leq P_{max}$  entonces
            guarda:  $O(1)$ 
            encolar nod O(1)
        fin si
    fin si
total:  $O(1)$ 

fin función

```

Aclaraciones de variables y transformación de entrada

- **Variable Mapa:** la matriz $(F * C)$ de nodos.
- **Tipo Nodo:** Dicho tipo, fue creado para contener más información de cada nodo, el mismo esta compuesto por posición i, j del nodo en el Mapa, si es pared o no y por último la cantidad de paredes destruidas y la distancia recorrida hasta dicho nodo.
- **cola:** Como la palabra lo indica, en esta cola se irán encolando los nodos.

1.4 Análisis de complejidades

El algoritmo BFS original, tiene una complejidad de $O(|V| + |E|)$, siendo V el conjunto de nodos y E el de aristas del grafo que se le pasa como parámetro. El sumando $|V|$ es obtenido a partir de que se encolan necesariamente cada uno de los $|V|$ nodos **1 sola vez**. Por otro lado, como cada nodo chequea 1 vez cada una de sus adyacencias (para ver si las encola o no), entonces cada nodo n es chequeado $d(n)$ veces (es decir hay $O(|E|)$ chequeos en total).

Siendo la entrada una matriz de $F \times C = N$, cada celda es transformada en un nodo, con lo cual el grafo resultante tendrá N nodos. Cada baldosa puede recorrerse a lo sumo en 2 direcciones y 4 sentidos distintos, es decir, los nodos que las representan tienen grado menor o igual a 4, osea orden constante. De aplicar BFS sobre el grafo de nuestro problema, la complejidad sería $O(N + 4N) \subseteq$

$O(N)$. No obstante, al modificar el BFS permitiendo que se puedan "visitar nodos ya visitados", la complejidad se ve afectada, ya que en la cola que guarda los nodos a evaluar permite reencolar a cada uno varias veces (permitiendo su reevaluación).

La cantidad de veces en la que es posible reencolar cada nodo, esta dada por la cantidad de caminos que pasen por él, los cuales comparten la característica de que cada uno de ellos mejora la cantidad de paredes a romper necesarias para alcanzarlo. Como ningún camino puede derribar más de "P" paredes y bajo la lógica de que un camino podrá visitar a un nodo (por lo tanto reencolarlo) si lo alcanza con una menor cantidad de paredes derribadas que el camino anterior, entonces solo P caminos podrán mejorar la condición de acceso al nodo. Por lo tanto la cantidad de veces que se encolarán nodos será $O(N \times P)$ determinando, de esta forma, la complejidad del algoritmo propuesto.

1.5 Demostración de correctitud

Para demostrar que nuestro algoritmo es válido para solucionar el problema en cuestión, tendremos que probar las siguientes condiciones:

- El algoritmo finaliza siempre.
- De haber solución, el algoritmo la encuentra.
- El camino resultante es el más corto dentro de los posibles.

El algoritmo finaliza

Como nuestro algoritmo puede encolar nodos ya visitados, queremos ver que no se producen ciclos infinitos entre nodos con la condición mencionada. Como se vio en la sección de complejidad, la máxima cantidad de veces que cada nodo puede ser recorrido es "P", con lo cual, necesariamente el algoritmo termina.

El algoritmo halla la solución

Como cada nodo se puede recorrer siempre y cuando se posea la cantidad de paredes a romper necesarias para alcanzarlo, entonces particularmente, el nodo destino será alcanzado solo si existe un camino que rompa una cantidad menor o igual a la parametrizada, es decir, si existe solución a la instancia del problema.

La solución encontrada es la mejor

Al igual que el algoritmo *BFS*, el grafo se recorre en anchura, por lo tanto en la cola todos los nodos a una distancia $d > 1$ del origen estarán encolados de manera contigua, es decir, cuando se visita un nodo a distancia $d - 1$ del origen, se encolan los vecinos, que son los que están a distancia d del nodo inicial, siempre y cuando la cantidad de paredes rotas para llegar a los mismos no exceda el límite. Si interpretamos a cada nodo encolado como la cabeza de un camino que empieza en el nodo origen, puede verse que en cada paso del algoritmo se están evaluando todos los caminos posibles de longitud creciente: de alcanzar en algún momento el nodo destino, por consiguiente, será a travez del camino más corto.

Como el procedimiento permite volver a recorrer nodos, es necesario que al alcanzar el nodo destino se finalice el algoritmo: de no suceder, puede existir otro camino que pueda volver a recorrer

el destino con una menor cantidad de paredes rotas pero necesariamente con un mayor tiempo de recorrido, sobrescribiendo el tiempo logrado por el camino minimo y generando una solución que no es la mejor.

1.6 Experimentos y conclusiones

1.6.1 Test

Para verificar el correcto funcionamiento de nuestro algoritmo , elaboramos diversos tests, los cuales serán enunciados a continuación.

Caso 1: No existe camino para atravesar el laberinto

Este caso se da cuando en cualquier camino nos encontramos con paredes irrompibles sin la posibilidad de esquivarlas.

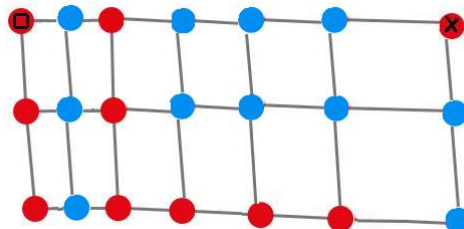
```

5 9 3
#####
#o# .###x#
#.#.#####
#.#.#####
#.#.....##
#####

```

Ejemplo 1.1 - Caso Sin Solución

El grafo que representa a esta entrada es de la siguiente forma:



Representacin 1.1 - Caso Sin Solución

Caso 2: Existe un camino sin atravesar paredes para recorrer todo el laberinto

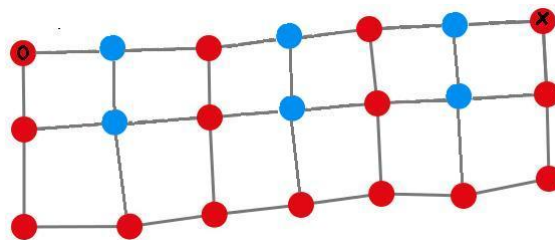
```

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

```

Ejemplo 1.2 - Caso Sin Romper Paredes

El grafo que representa a este tipo es de la siguiente forma:



Representacin 1.2 - Caso Sin Romper Paredes

Caso 3: Rompiendo todas las paredes posibles para pasar el laberinto

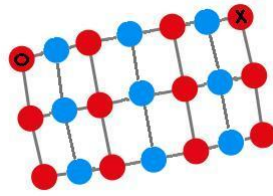
```

5 9 3
#####
#o#.#.#x#
#.#.#.#.#
#.#.#.#.#
#####

```

Ejemplo 1.3 - Caso Rompiendo Todas Paredes Las Posibles

El grafo que representa a este tipo es de la siguiente forma:



Representacin 1.3 - Caso Rompiendo Todas Paredes Las Posibles

Caso 4: Rompiendo una cantidad menor de paredes posibles para pasar el laberinto

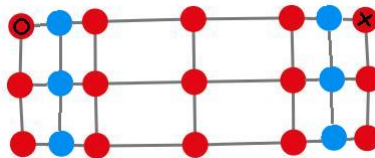
```

5 9 5
#####
#o#...#x#
#.#...#.#
#.#...#.#
#####

```

Ejemplo 1.4 - Caso con soluci3n

El grafo que representa a este tipo es de la siguiente forma:



Representacin 1.4 - Caso con soluci3n

Aclaraciones:

- Nodo color celeste es una pared.
- Nodo color rojo es caminable.

1.6.2 Performance del algoritmo

En lo que sigue, mostraremos buenos y malos casos para nuestro algoritmo, y a su vez, daremos el tiempo estimado seg3n la complejidad del algoritmo calculada anteriormente.

A continuaci3n mostraremos un gr3fico de tiempos comparativo entre distintas familias de casos:

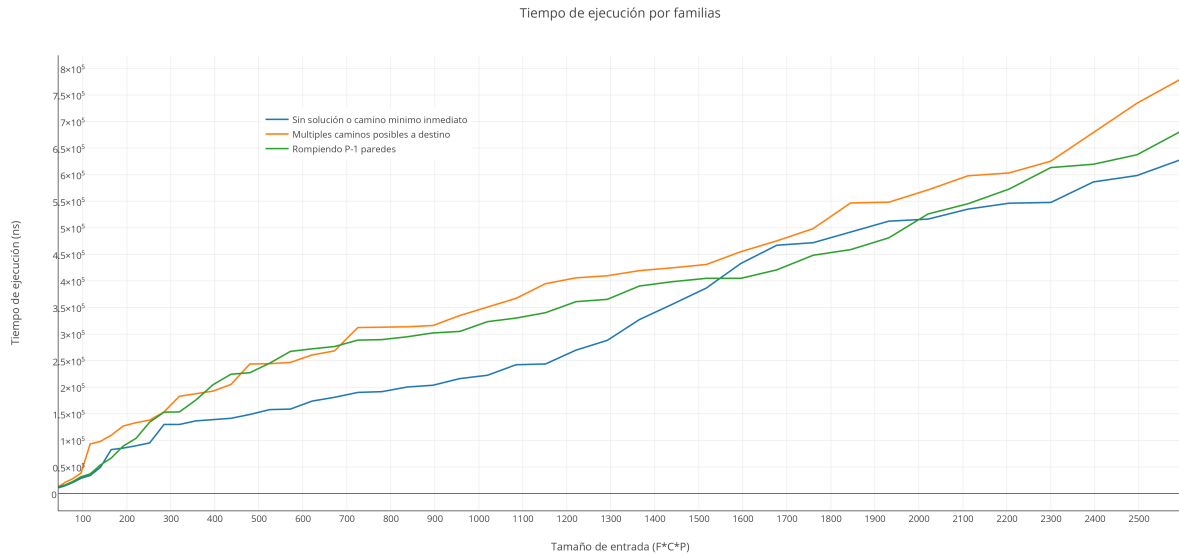


Gráfico 1.1 - Comparativo

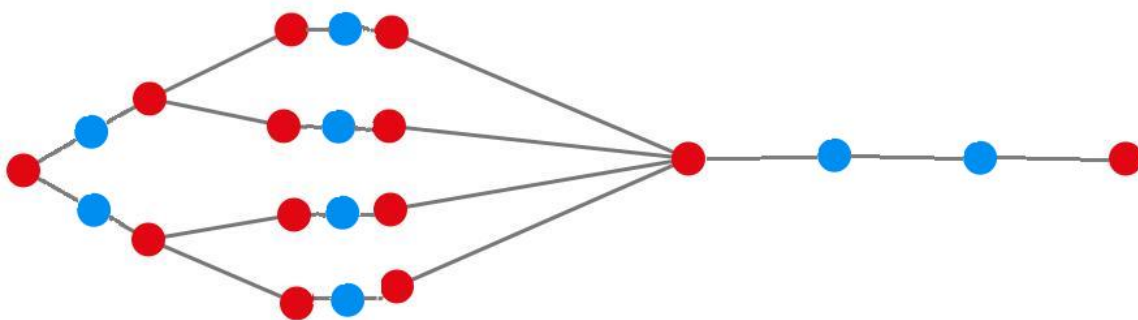
Se puede observar en el gráfico, tres funciones las cuales representan el tiempo de ejecución de las familias de casos:

- Sin solución o con camino mínimo inmediato
- Rompiendo una cantidad de P-1 paredes
- Múltiples caminos a destino posibles

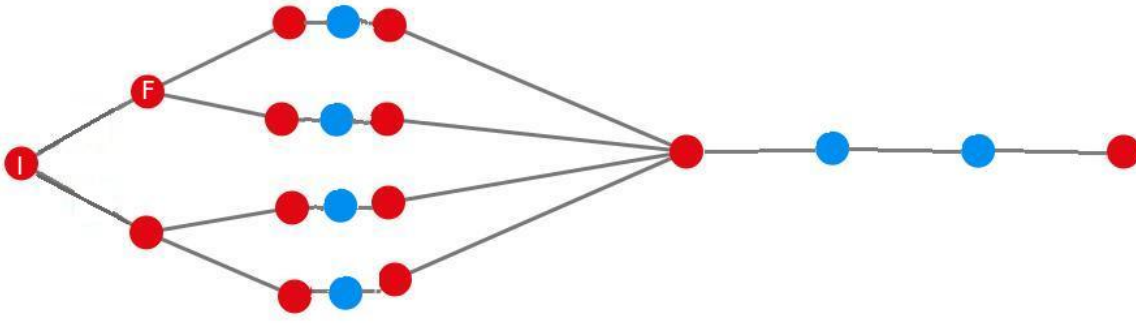
Como se observa en el gráfico la función representativa de la flia número 1, presenta una mejor performance en relación a las otras. Esto se debe a que en el primer paso no puede salir por ningún camino posible, o su adyacente es el nodo destino por lo tanto chequea solo los nodos adyacentes del origen y finaliza su ejecución.

Por lo cual, el mejor caso teniendo en cuenta a nivel performance de nuestro algoritmo será cuando se presente un problema **Sin solución o con camino mínimo inmediato**

Los grafos que representan lo dicho serían los siguiente:



EjemploGrafo G1.1 - Mejor Caso Con P = 0



EjemploGrafo G1.1 - Mejor Caso Con $P = 0$

Para una mayor observación desarrollamos el siguiente gráfico con las instancias:

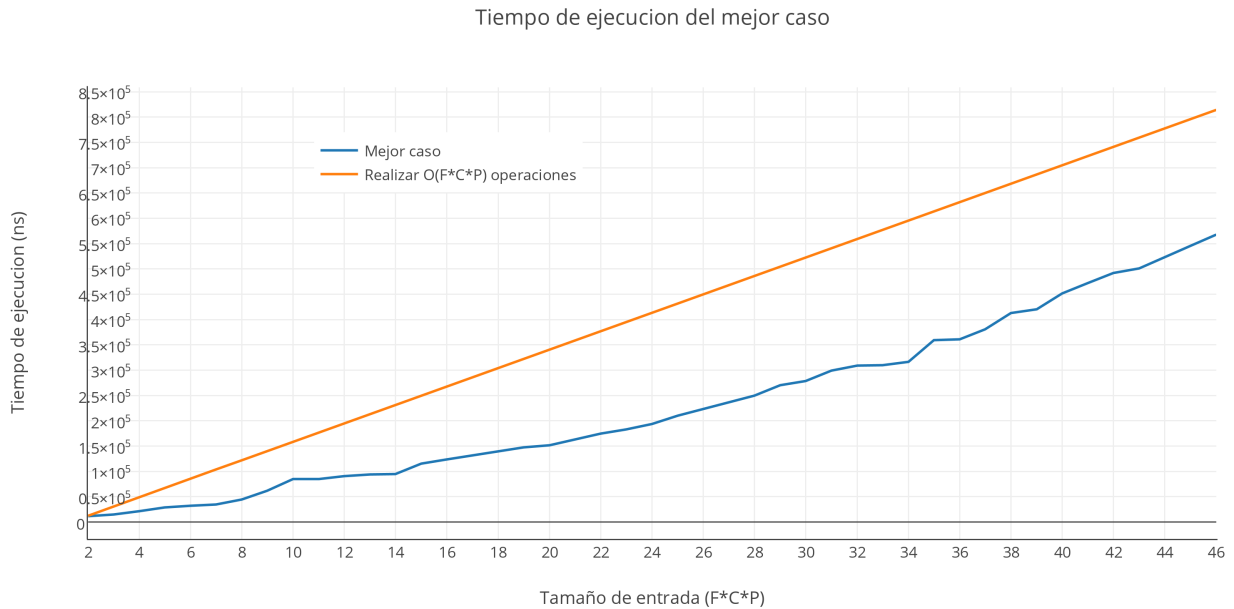


Gráfico 1.1 - Mejor Caso

Como la complejidad que calculamos se basa en la cantidad de nodos posibles que puede haber multiplicado por las paredes contrastaremos nuestras mediciones con los valores de $O(F \times C \times P)$.

Luego, dividiendo la función resultante de nuestro algoritmo sobre el tiempo que demanda realizar $O(F \times C \times P)$ nos da lo siguiente:

Tiempo de ejecución del mejor caso sobre la complejidad teorica

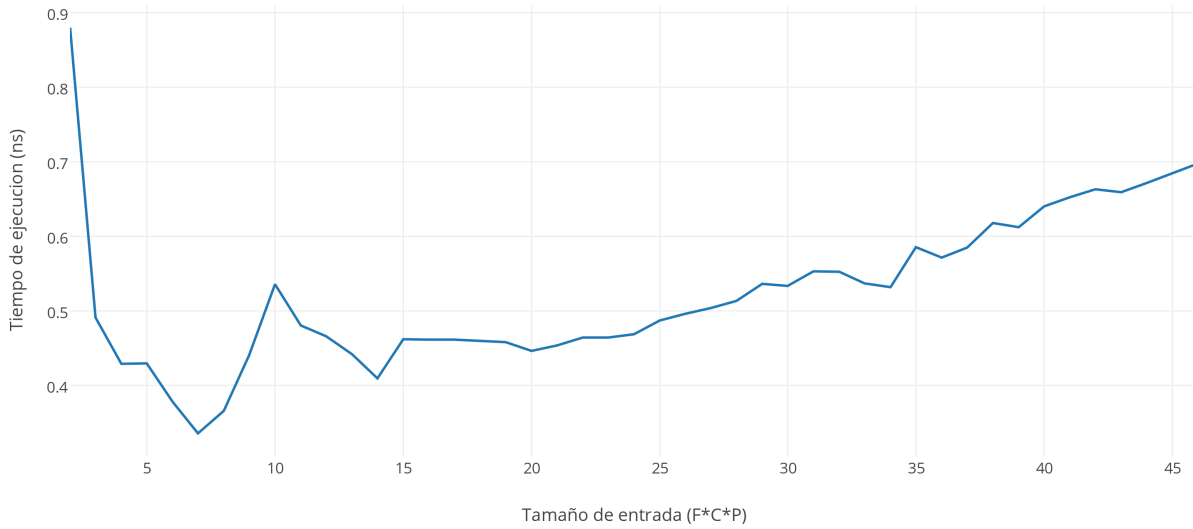
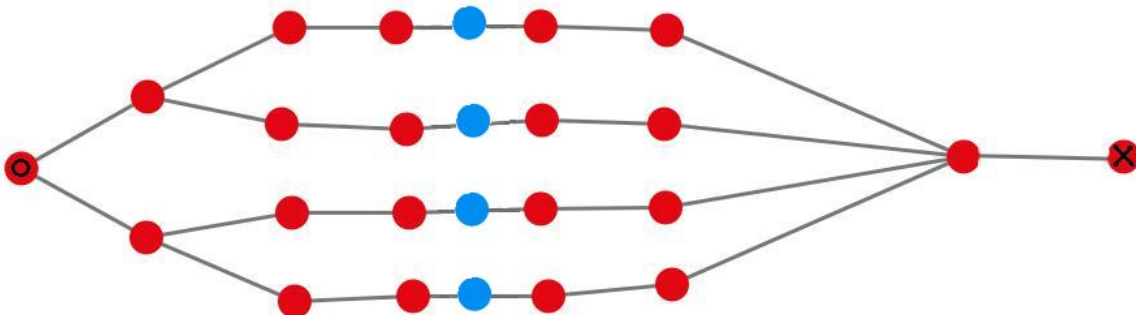


Gráfico 1.2 - Mejor Caso Sobre Complejidad $O(F \times C \times P)$

Para obtener dichas instancias nos resulto prudente realizar aproximadamente unas 20 corridas con el mismo input y sacar el promedio de estas 20 corridas para cada instancia para obtener resultados más consisos.

Se observa en el gráfico 1.1 como la función que representa al tiempo de nuestro algoritmo en el mejor caso es considerablemente mejor que la función de la cota teórica, mientras que al realizar la división entre ambas funciones se llega a que la misma nunca supera el valor 0.9 quedando siempre por debajo de 1 mostrando así que para cualquier valor de entrada posible el mejor caso queda siempre en el orden de $O(F \times C \times P)$.

Luego, uno de los peores casos para nuestro algoritmo es en el cual **se debe recorrer todos los posibles caminos ya que son todos exactamente iguales**, esto se da así ya que nuestro algoritmo chequea todos los caminos posibles y como todos pueden ser solución posible avanza por todos y llega al final del laberinto con el mismo valor en todos los posibles caminos



EjemploGrafo G1.2 - Peor Caso

Para este grafo realizamos las respectivas mediciones las cuales arrojaron los siguientes resultados:

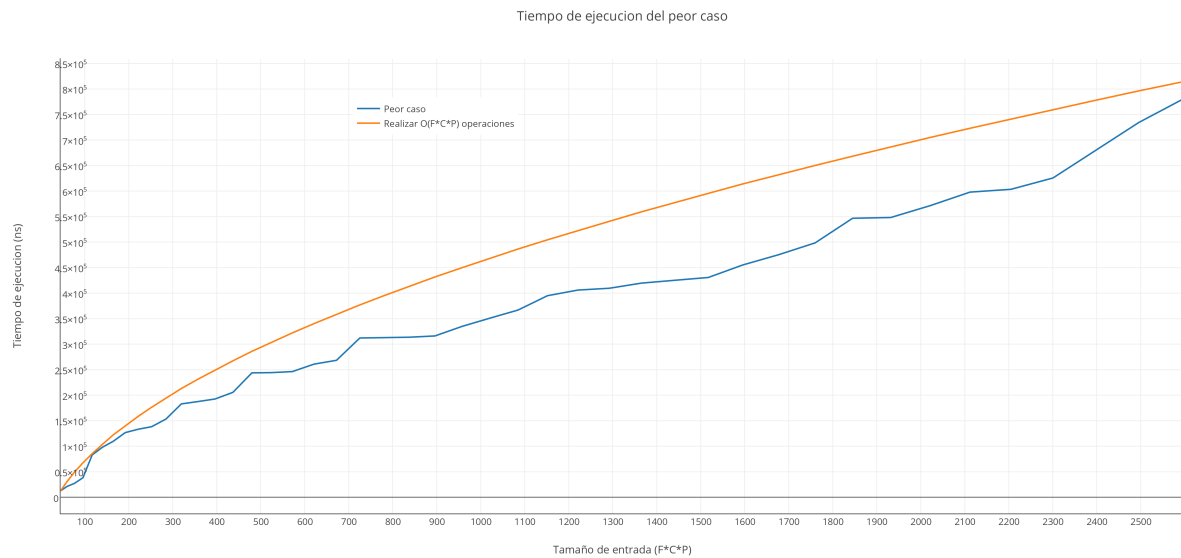


Gráfico 1.3 - Peor Caso

Dividiendo por la complejidad se llego a lo siguiente:

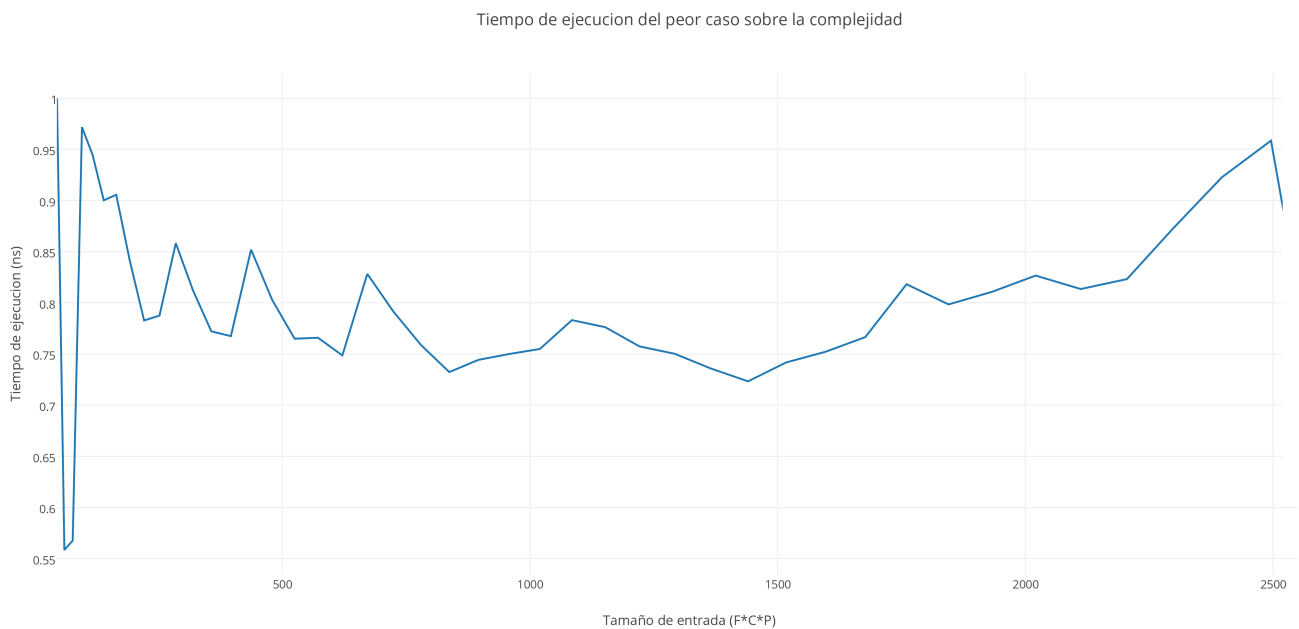


Gráfico 1.4 - Peor Caso Sobre Complejidad $O(F \times C \times P)$

Para obtener dichas instancias nos resulto prudente realizar aproximadamente unas 20 corridas con el mismo input y sacar el promedio de estas 20 corridas para cada instancia para obtener resultados más consisos.

Podemos observar en la figura 1.3 como la función resultante de nuestro algoritmo en el peor caso se mantiene por debajo de la función final del tiempo de realizar $O(F \times C \times P)$ operaciones lo cual fue la complejidad precalculada. Y, en el gráfico 1.4 es posible observar como la función obtenida presenta diversos picos los cuales nunca llegan a superar 1, y cuando el tamaño de entrada aumenta la función queda asintotizada por 0.95.

Por último mostraremos un gráfico comparativo con el mejor, el peor y el caso promedio contra la complejidad precalculada .

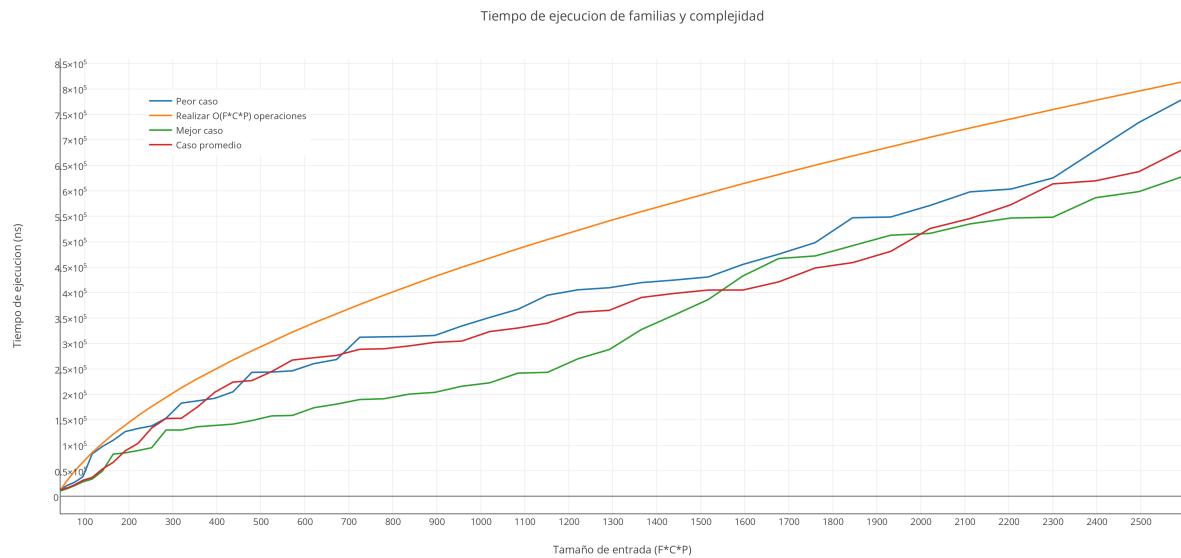


Gráfico 1.5 - Comparativo

Es visible en este gráfico comparativo como las funciones relacionadas a nuestro algoritmo están acotadas por la función de la complejidad teórica.

Mostraremos a continuación un caso en el cual se dejaron fijas las variables F y C , ambas en 50 y se trabajó con P móvil para ver como se comporta nuestro algoritmo en esta situación.

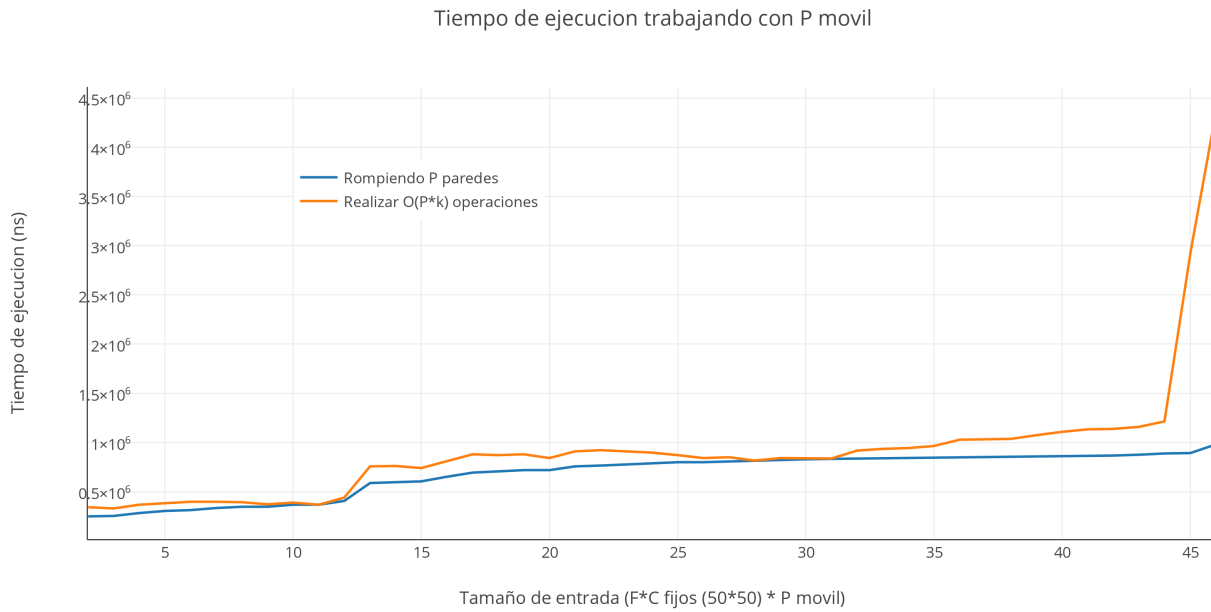


Gráfico 1.6 - P movil y F,C Fijos

Se puede observar en el gráfico 1.6 como al trabajar con este caso en el cual a medida que P aumenta es necesario romper P paredes para llegar a destino, la función resultante se comporta de la misma manera que $O(P)$, chequeando de esta forma que trabajando con una cantidad fija de nodos nuestro algoritmo realiza $O(P)$ operaciones por cada elemento del grafo.

Luego de lo mostrado, se pudo observar que, tanto en el mejor como en el peor caso nuestro algoritmo se encuentra en el orden de la complejidad calculada.

2 Ejercicio 2

2.1 Descripción de problema

Una vez que el equipo fue avanzando por el laberinto en búsqueda de la X, se fueron encontrando con piezas de una tabla en el suelo, las cuales tenían un manuscrito antiguo. Por lo tanto, quisieron juntar todas las piezas que componían la tabla. Similar al punto anterior, encontraremos el camino posible realizando el menor esfuerzo para recorrer todas las salas y juntar las piezas para armar la tabla completa.

2.2 Explicación de resolución del problema

En esta ocasión se debe encontrar, dada una serie de habitaciones, la forma menos costosa de unir las, teniendo que derribar determinadas paredes para lograrlo. Las mismas poseen un costo en energía que va del cero al nueve.

Dada 2 habitaciones: el muro menos costoso a derribar que une a las mismas será la elección a tomar para formar la solución.

Tenemos que tener en cuenta que hay muros que no son derribables. Estos se representan con un numeral ('#').

Para resolver este problema, caracterizaremos el mismo sobre un grafo que representa cada habitación como un conjunto de puntos caminables adyacentes entre si. Y a las paredes rompibles, es decir, las paredes numeradas como aristas entre los puntos caminables.

Las paredes irrompibles no conectarán habitaciones.

Lo que se busca entonces, es una forma de unir todas las habitaciones de la forma menos costosa.

Para que exista solución, tiene que haber una forma de cruzar de una habitación a otra. Por lo tanto, tiene que existir una pared rompible que las una. La misma conectará dos puntos caminables, uno en cada habitación.

Dentro de cada habitación, los puntos caminables estarán unidos por aristas de valor nulo.

Una pared numerada tiene que unir exactamente dos puntos de izquierda a derecha o de arriba hacia abajo. Por lo tanto, las direcciones no utilizadas tienen que estar ocupadas por paredes, ya que si no, existe ambigüedad para determinar que puntos conecta.

Como en este problema no existen movimientos en diagonal sobre el mapa, los únicos casos válidos serán:

	●	
#	X	#
	●	

Y el simétrico.

Para obtener la solución, se busca el arbol generador mínimo (AGM) del grafo planteado mediante el algoritmo de Kruskal.

A dicho AGM resultante del algoritmo se le calcula el esfuerzo total, que será la suma de todas las paredes destruidas para unir las habitaciones, es decir, la solución al problema planteado.

Para desarrollar el algoritmo de búsqueda del AGM del grafo, implementamos las funciones find y unión, las cuales se encargan de ir armando el arbol avanzando dentro de una habitación o cruzando a otra habitación rompiendo una pared.

2.3 Algoritmos

```

función Kruskal
  ordenar aristas O(4 * F * C * log(4 * F * C))
  representanteFinal ← -1 O(1)
  Para cada Arista ar ∈ aristas hacer
    ciclo: O(4 * F * C)
    si Buscar(ar.inicio) = Buscar(ar.fin) entonces
      guarda: O(2 * log(4 * F * C))
      Unir(ar.inicio, ar.fin, ar.costo) O(1)
    fin si
    si representanteFinal ≥ 0 entonces
      finalizar ciclo O(1)
    fin si
  fin para
  si representanteFinal ≥ 0 entonces
    devolver costCompLider[representanteFinal] O(1)
  fin si
  de lo contrario
    devolver sinSolucion O(1)
  fin si
fin función
total: O(F * C * log(F * C))

función Buscar(entero x)
  si padre[x] = x entonces
    devolver x O(1)
  fin si
  para entero p ← Buscar(padre[x]) guarda: O(log(4 * F * C))
  para padre[x] asignar p O(1)
  devolver p O(1)
fin función
total: O(log(4 * F * C))

```

```

función Unir(enteros x, y, costo)
    para entero x asignar Buscar(X) O(1)
    para entero y asignar Buscar(Y) O(1)
    si  $\text{altura}[x] \leq \text{altura}[y]$  entonces
        para padre[x] asignar y O(1)
        para cantAristas[y] asignar cantAristas[y] + cantAristas[x] + 1 O(1)
        para costCompLider[y] asignar costCompLider[y] + costCompLider[x] + costo O(1)
        si  $\text{altura}[x] = \text{altura}[y]$  entonces
            para altura[x] incrementar 1 O(1)
        fin si
        si  $\text{cantAristas}[y] = \# \text{nodos} - 1$  entonces
            para representanteFinal asignar y O(1)
        fin si
    de lo contrario
        para padre[y] asignar x O(1)
        para cantAristas[x] asignar cantAristas[y] + cantAristas[x] + 1 O(1)
        para costCompLider[x] asignar costCompLider[x] + costCompLider[y] + costo O(1)
        si  $\text{cantAristas}[x] = \# \text{nodos} - 1$  entonces
            para representanteFinal asignar x O(1)
        fin si
    fin si
fin función
total: O(1)

```

Aclaraciones de variables y transformación de entrada

- **Arreglo cantAristas:** guarda la cantidad de aristas que tiene el representante de una componente conexa.
- **Arreglo costCompLider:** guarda el costo total de una componente conexa por representante.
- **Arreglo padre:** para cada nodo, quien es su representante, es decir, en que componente conexa se encuentra.
- **Arreglo altura:** indica la altura de cada componente conexa.

2.4 Análisis de complejidades

Siendo F la cantidad de filas y C la cantidad de columnas de la entrada, $F * C$ es la cantidad de puntos caminables máximos que puede tener una instancia. El grafo implícito tiene entonces $V(G) = F * C$ nodos.

Dado que la implementación del algoritmo de Kruskal utiliza la técnica de union-find con tiempo de unión y búsqueda amortizados $E * \log(E)$ siendo E la cantidad de aristas del grafo y ordenamiento inicial de las aristas que se logra también en $E * \log(E)$ la complejidad total será de $O(E * \log(E))$.

Dado que el grafo más completo que puede ofrecer este problema es cuando se tiene una sola sala sin paredes, la cantidad de aristas total será $4 * V(G) = 4 * F * C$ dando como resultado un orden de complejidad $4 * F * C * \log(4 * F * C) \subseteq O(F * C * \log(F * C))$.

2.5 Demostración de correctitud

Necesitamos encontrar la forma de unir todas las habitaciones, es decir, lo que se necesita es encontrar en el grafo una única componente conexa.

Como en toda componente conexas tiene minimamente un arbol generador, de encontrarlo, se encuentra una posible solución al problema. Los únicos ejes del árbol a encontrar que tendrán peso serán aquellos que representan las paredes rompibles en el problema. Si pedimos que el peso del arbol a encontrar sea mínimo, entonces estaremos eligiendo las paredes con menor peso a romper para acceder entre habitaciones. Por lo tanto, el AGM producto de la aplicación del algoritmo de Kruskal sobre el grafo, representa una solución válida al problema propuesto, dado que garantiza que se utilizarán las paredes con menos peso posible para unir todas las habitaciones.

2.6 Experimentos y conclusiones

2.6.1 Test

Para verificar el correcto funcionamiento de nuestro algoritmo , elaboramos diversos tests, los cuales serán enunciados a continuación.

Caso 1: No existe arbol que conecte todas las salas

Este caso se da cuando en cualquier camino nos encontramos con paredes irrompibles sin la posibilidad de esquivarlas

Para este tipo de testeo mostraremos a continuación un ejemplo del mismo.

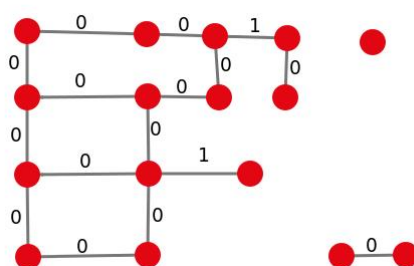
```

6 9
#####
#...1.#2#
#...#.#
#..1.####
#..###..#
#####

```

Ejemplo 2.1 - Caso Sin Solución

El grafo que representa a este tipo es de la siguiente forma:



Representación 2.1 - Caso Sin Solución

Caso 2: Existe un camino que conecta todas las salas de esfuerzo 0

Esta versión se da cuando la suma de los pesos de las aristas del AGM obtenido es 0. Para este tipo de testeo mostraremos a continuación un ejemplo del mismo.

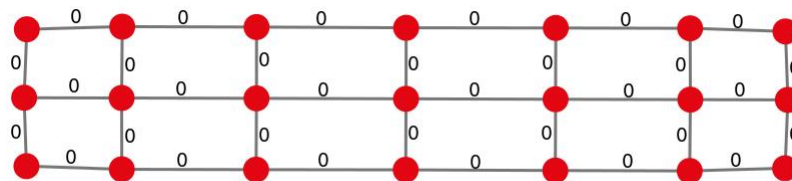
```

5 9
#####
# . . . . . #
# . . . . . #
# . . . . . #
#####

```

Ejemplo 2.2 - Caso Sin Romper Paredes

El grafo que representa a este tipo es de la siguiente forma:



Representacin 2.2 - Caso Sin Romper Paredes

Caso 3: El AGM es todo el grafo

Este caso se da cuando el laberinto no presenta varios caminos para poder ir a las salas sino un único camino.

Aquí veremos, un ejemplo del conjunto de test de este tipo.

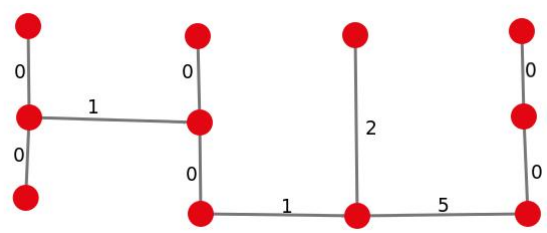
```

5 9
#####
#.#.#.#.#
#.#1.#2#.#
#.#.1.5.#
#####

```

Ejemplo 2.3 - Caso AGM = GRAFO

El grafo que representa a este tipo es de la siguiente forma:

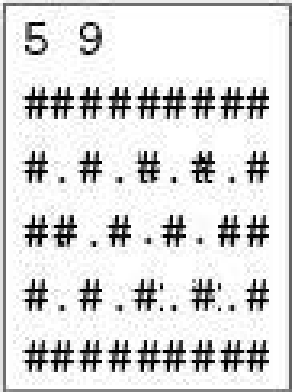


Representacin 2.3 - Caso $AGM = GRAFO$

Caso 4: Sin ejes

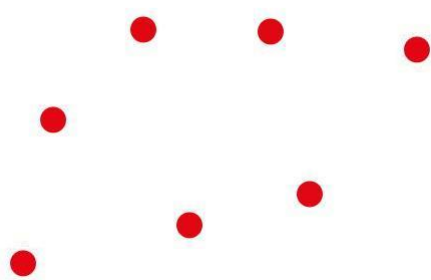
Este caso se da cuando todas las salas presentan paredes irrompibles sin la posibilidad de conectarse generando así un grafo sin aristas.

Siguiendo el desarrollo de dicho informe a continuación mostraremos.



Ejemplo 2.4 - Caso Sin Ejes

El grafo que representa a este tipo es de la siguiente forma:



Representacin 2.4.2 - Caso Sin Ejes

Caso 5: Camino por las salas random

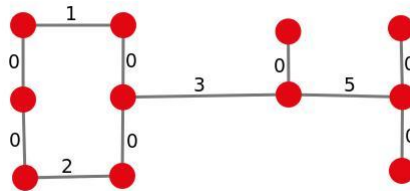
Este caso se da cuando hay posibilidad de conectar todas las salas rompiendo una cierta cantidad de paredes, también denominado caso random debido al desarrollo de nuestro algoritmo.

Siguiendo el desarrollo de dicho informe a continuación mostraremos.

```
5 9
#####
#.1.#.#.#
#.#.3.5.#
#.#.2.#.#
#####
```

Ejemplo 2.5 - Caso Random

El grafo que representa a este tipo es de la siguiente forma:



Representación 2.5.2 - Caso Random

Aclaraciones:

- Nodo aislado significa que está rodeado de paredes irrompibles no hay camino posible de conectar dicho nodo a la componente conexa.

2.6.2 Performance del algoritmo

Acorde a lo solicitado, mostraremos distintos tipos de familias de casos para nuestro algoritmo, y además, daremos el tiempo estimado según la complejidad del algoritmo calculada anteriormente.

A continuación mostraremos un gráfico de tiempos comparativo entre distintas familias de casos:

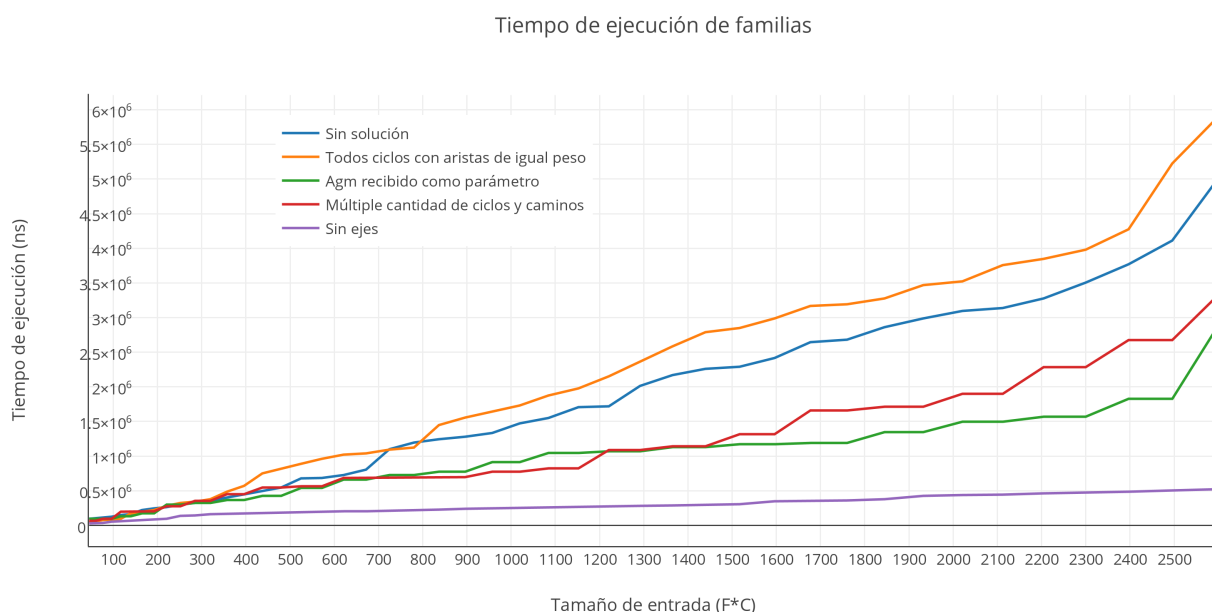


Gráfico 2.1 - Comparativo

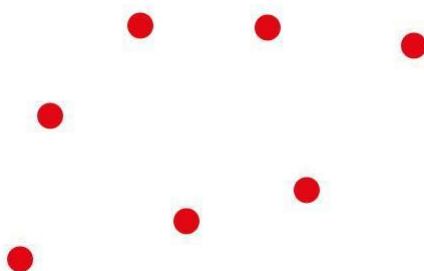
Se puede observar en el gráfico, cinco funciones las cuales representan el tiempo de ejecución de las familias de casos:

- No hay conexión posible entre las salas
- El grafo recibido como parámetro es el AGM
- No hay arbol posible que conecte a todas las componentes conexas
- Existe un arbol de peso 0
- El grafo recibido como parametro presenta multiples ciclos con aristas de disversos pesos

Como se observa en el gráfico la función representativa de la flia número 1, presenta una mejor performance en relación a las otras. Esto se debe a que nuestro algoritmo al intentar chequear las aristas, observa que no hay ninguna por lo tanto finaliza su ejecución insumiendo unicamente la creación del grafo (nodos aislados)

Habiendo chequeado dichas instancias, llegamos a la conclusión que la familia de casos que presenta una mejor performance para nuestro algoritmo es en el cual **el grafo que se obtiene de transformar el laberinto recibido como parámetro no presenta ninguna arista**

Un grafo representativo de lo dicho sería el siguiente:



Grafo 2.1 - Mejor Caso

Para llegar a dicha conclusión trabajamos con 50 instancias.

Para una mayor observación desarrollamos el siguiente gráfico con las instancias:

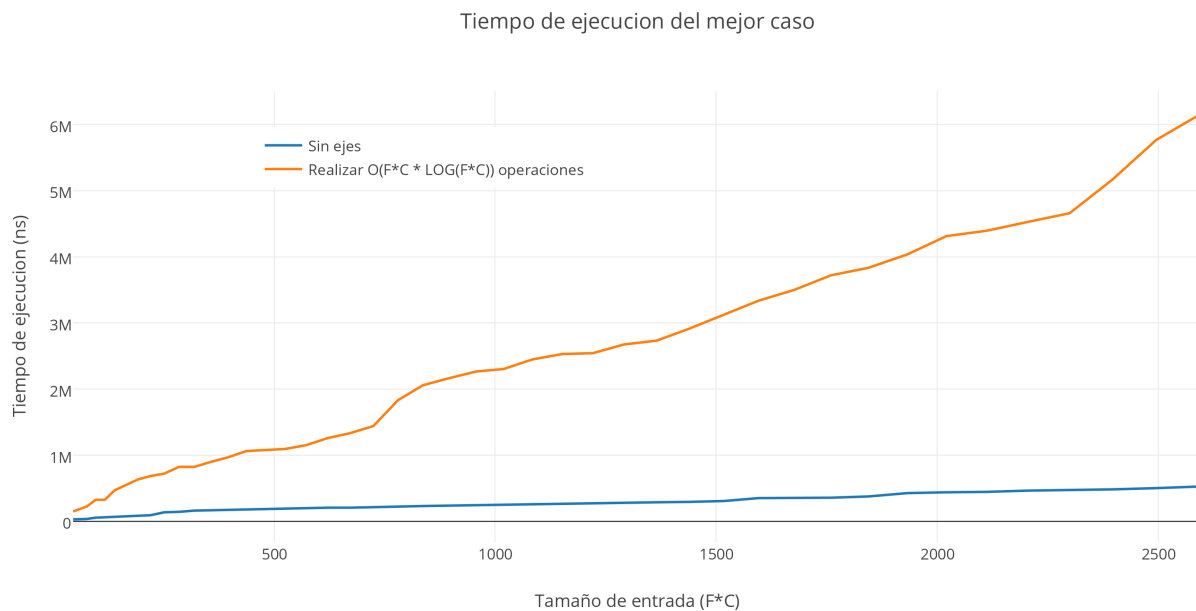


Gráfico 2.1 - MejorCaso

Como es posible observar en el gráfico, la función resultante de la cota teorica crece mucho más rápido que la de nuestro algoritmo la cual queda considerablemente por debajo .

Luego, dividiendo por la complejidad teórica de nuestro algoritmo llegamos a:

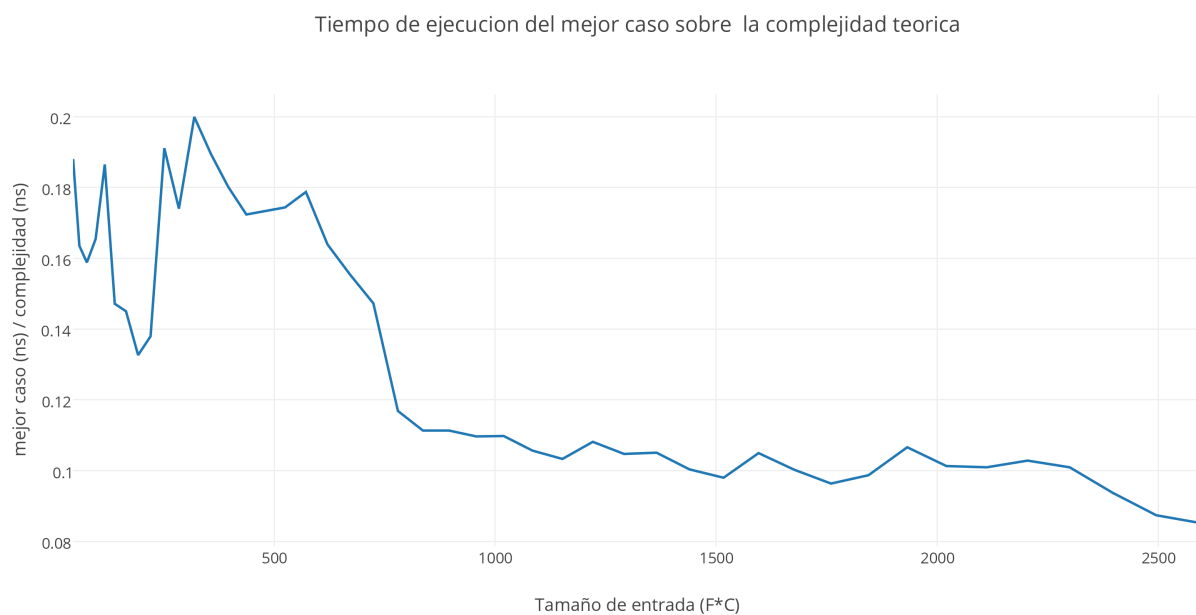
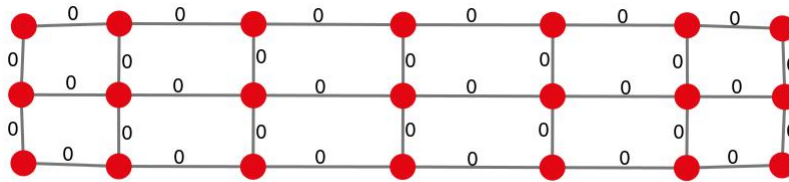


Gráfico 2.2 - Mejor Caso / Complejidad $O(F \cdot C \cdot \log(F \cdot C))$

Para realizar esta experimentación nos pareció prudente, realizar un promedio con el mismo input de aproximadamente 20 corridas tanto para la complejidad como para nuestro algoritmo y una vez calculado dicho promedio de ambas cosas realizamos la división para obtener resultados más relevantes.

Se puede observar en el gráfico 2.2, como luego de realizar la división por la complejidad se ve que la función resultante llega a un máximo igual a 0.2 y cuando el valor de entrada aumenta la función tiende a 0. Por lo tanto, podemos concluir que para el mejor caso nuestro algoritmo se encuentra considerablemente por debajo de la cota teorica $O(F * C * \log(F * C))$.

Luego, verificando el peor caso, llegamos a la conclusión que la familia de casos en el que resulta menos beneficioso trabajar con nuestro algoritmo será cuando **el grafo que se obtiene de transformar el laberinto de entrada es aquel que presenta un ciclo por cada habitación posible**, dandonos el siguiente grafo una vez transformado:



Grafo 2.2 - Peor Caso

Realizando experimentos con un total de 50 instancias, desarrollamos dos gráficos los cuales mostraremos a continuación:

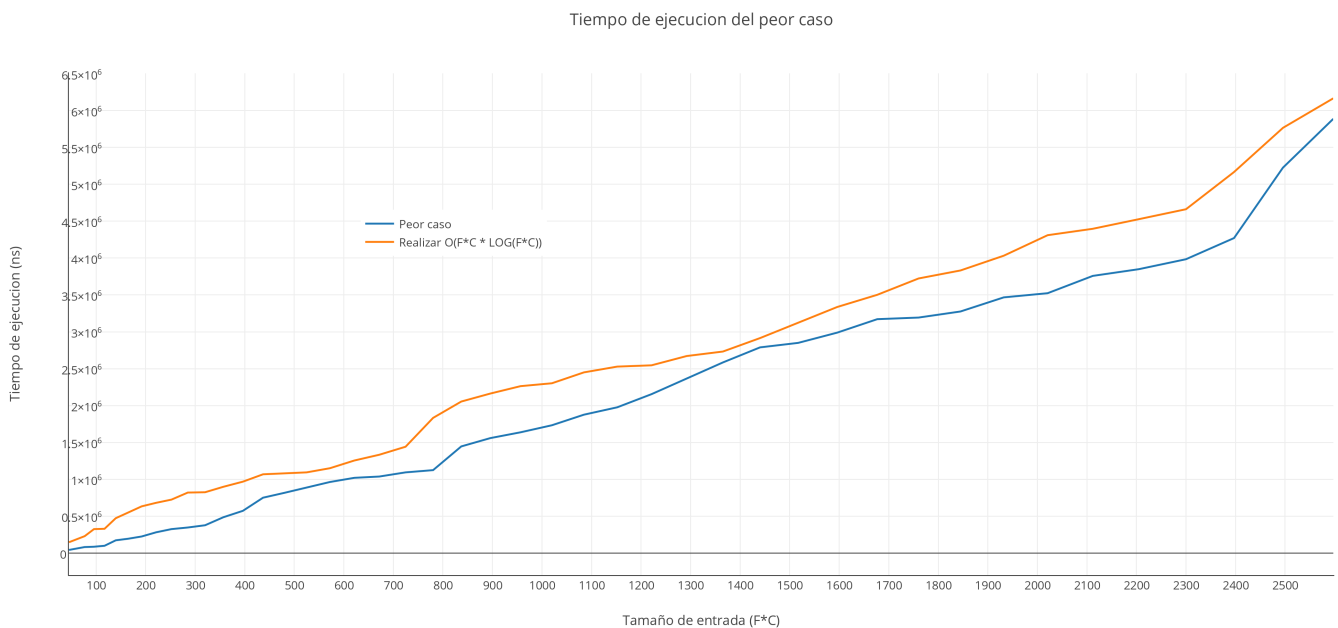
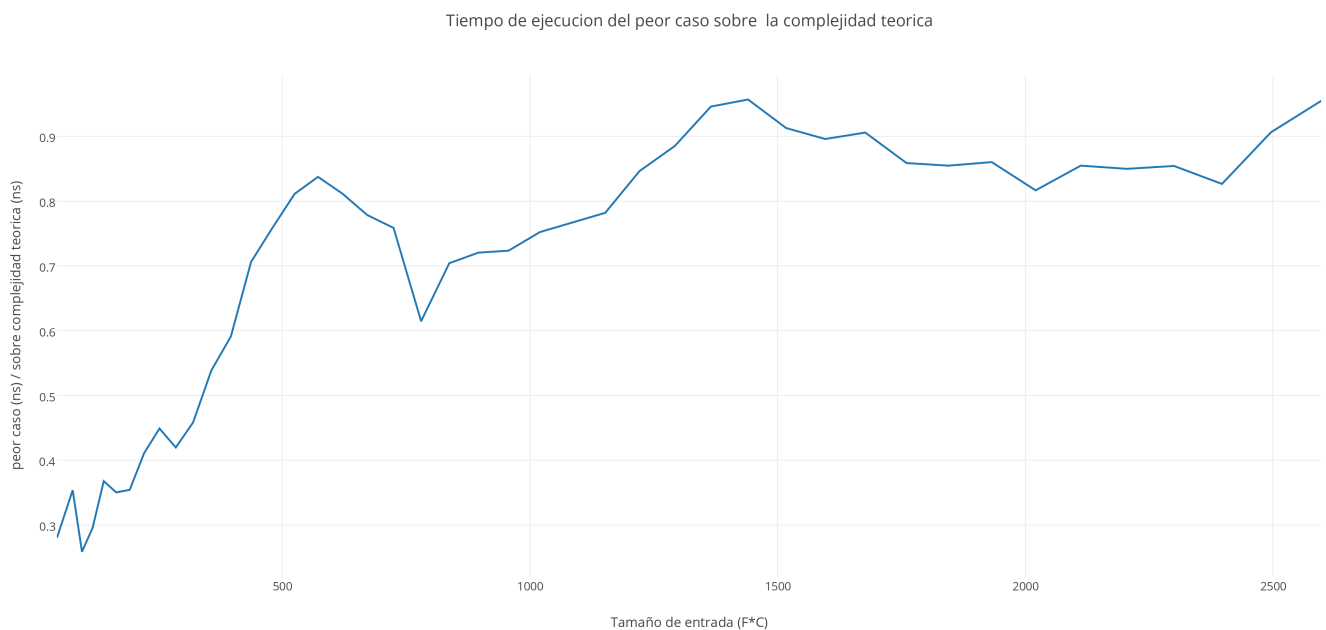


Gráfico 2.3 - Peor Caso

Dividiendo por la complejidad propuesta llegamos a:



*Gráfico 2.4 - Peor Caso / Complejidad $O(F * C * \log(F * C))$*

Para realizar esta experimentación nos pareció acorde, realizar un promedio con el mismo input de aproximadamente 20 corridas tanto para la complejidad como para nuestro algoritmo y una vez calculado dicho promedio de ambas cosas realizamos la división para obtener resultados más relevantes.

Como se puede observar en el gráfico 2.3, la función resultante de nuestro algoritmo es considerablemente mejor que la de la cota teorica y presenta un tiempo similar al de la función resultante de la cota. Luego, en el gráfico 2.4 se ve como la función resultante esta asintotizada por 1.

Por último, mostraremos un gráfico comparativo entre el mejor y peor caso contra la complejidad que se solicito:

Tiempo de ejecucion de familias y complejidad teorica

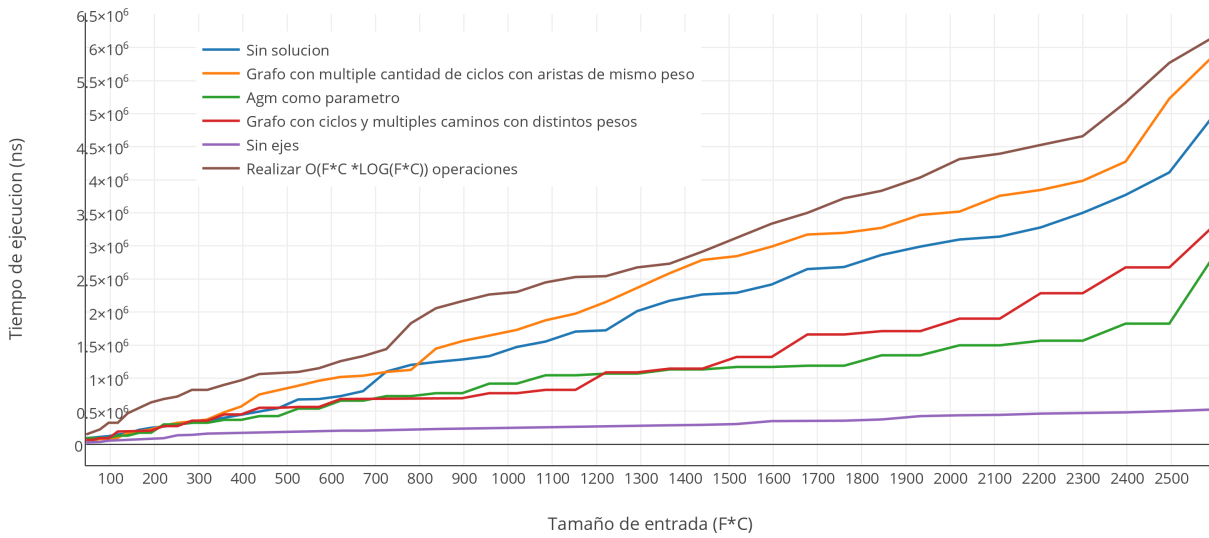


Gráfico 2.5 - Comparativo

Luego de dichos experimentos y casos probados, se puede concluir que a pesar de tener ciclos en todas las salas y donde dichos ciclos presenten aristas con pesos iguales lo que generara al algoritmo la posibilidad de crear varias ramas posibles de solución nos mantenemos dentro de la complejidad propuesta como habíamos mostrado en nuestro desarrollo de la complejidad.

3 Ejercicio 3

3.1 Descripción de problema

Luego de haber conseguido todas las piezas, llegan a la cruz y se encuentran con unos carritos apoyados sobre unas vías las cuales al parecer llegan hasta afuera de la fortaleza. De un momento a otro, el equipo empieza a escuchar ruidos de adentro del laberinto, de tanto romper las paredes, la estructura se estaba desplomando, es por esto que deben conseguir un escape rápido, como al lado del carrito ven que se encuentra un mapa con estaciones y el final, nos solicitan ayuda para encontrar el camino mínimo de estaciones para llegar afuera de la forma más rápida y eficiente.

3.2 Explicación de resolución del problema

Dado un mapa con N estaciones y el tiempo que toma viajar entre dos de ellas, debemos encontrar la forma más rápida de viajar entre la primera estación y la última. Para ello, decidimos representar el mapa como un grafo orientado con pesos en sus aristas, donde cada estación fue representada por un nodo y el camino de estación a estación por aristas. Dado este grafo (al cual decidimos implementar mediante listas de adyacencia) el problema se reduce a encontrar el camino mínimo sobre él.

Para lo enunciado, aplicamos el algoritmo de Dijkstra sobre nuestra representación del grafo.

En el mismo se recorren los nodos del grafo teniendo en cuenta primero los más cercanos al origen, comenzando por el origen mismo. Un invariante de este algoritmo es que, para el conjunto de nodos ya visitados se conoce definitivamente el camino mínimo desde el origen hacia cualquier nodo perteneciente a dicho conjunto.

Cuando se visita un nodo se comprueba si se puede desde este nodo llegar más rápido a sus vecinos que lo calculado anteriormente. De ser así, se actualiza la distancia de ese vecino al origen y se guarda el nodo adyacente desde el cual se pudo conseguir dicha distancia. Cuando visitamos el nodo destino es porque se encontró el camino más corto a él.

Por último, solo resta recorrer desde el nodo salida y armar el camino mínimo a partir de los nodos previos que guardábamos cuando se iban actualizando las distancias parciales de estos.

3.3 Algoritmos

A continuación se detalla el pseudo-código de la parte principal del algoritmo:

```
función Dijkstra(estaciones, vias)  
  para Grafo fortaleza asignar grafo(estaciones, vias) O(N2)  
  Para cada nodo  $\in$  fortaleza hacer  
    Ciclo: O(N)  
    para nodo.distancia asignar invalido O(1)  
    para nodo.visitado asignar falso O(1)  
  fin para  
  Mientras  $\exists$  nodo alcanzable que no haya sido visitado hacer  
    Ciclo: O(N) condicion: O(N)  
    para actual asignar masCercano(visitados, distancia, cantidadNodos) O(N)  
    para visitados[actual] asignar verdadero O(1)  
    para vecinos asignar vecinos(fortaleza, actual) O(1)  
    Para cada arista  $\in$  vecinos hacer  
      Ciclo: O(N)  
      para dist asignar distancia(actual) + peso(arista) O(1)  
      si dist < distancia(arista.destino) // distancia(arista.destino) == invalido  
        entonces  
          para distancia[v.destino] asignar dist O(1)  
          para previos[v.destino] asignar actual O(1)  
        fin si  
      fin para  
    fin ciclo  
  fin función
```

Complejidad total del algoritmo: $O(N^2)$ con N = cantidad de estaciones.

Aclaraciones de variables y funciones

- La función *grafo*, crea grafo uniendo las aristas con los respectivos nodos que son adyacentes.
- La función *vecinos* devuelve una lista de aristas correspondiente a los nodos adyacentes al vértice consultado
- La función *masCercano* devuelve el nodo con la menor distancia al origen, que no haya sido visitado.

3.4 Análisis de complejidades

La entrada de nuestro algoritmo tiene m líneas. Estas representan las aristas que va a contener nuestro grafo. Cada arista es procesada y almacenada en nuestro grafo en tiempo constante. Sabemos que un digrafo tiene la cantidad de aristas acotadas por $n * (n - 1)$ siendo n la cantidad de nodos. Entonces construir nuestro grafo tiene una complejidad de $O(n^2)$.

Para el algoritmo de Dijkstra implementamos su cola de prioridad con dos arreglos, uno con la mínima distancia encontrada hacia el nodo y otro que nos indica si un nodo fue visitado o no. Por lo tanto, conseguir el próximo nodo a visitar es recorrer uno por uno los elementos del arreglo de distancias y quedarnos con el índice del nodo con menor valor válido siempre que esté marcado como "no visitado" en el otro arreglo. Esta operación tiene costo lineal en la cantidad de nodos ($O(n)$) y se realiza en el peor caso n veces cuando el último nodo que se visite es la salida, lo cual resulta en un costo cuadrático.

Luego se recorren los vecinos de un nodo y se actualizan sus distancias. Dada nuestra representación en arreglos, actualizar la distancia de cada vecino toma tiempo constante. Para los n

nodos se recorren sus vecinos, que en principio podrían ser $n - 1$. Entonces, la complejidad de esta operación es $O(n^2)$.

Finalmente recorreremos en complejidad $O(n)$ el arreglo *prev* donde se guardan los vecinos que realizan el camino mínimo a cada nodo. De esta manera construimos efectivamente el camino mínimo desde el destino hacia el origen y lo imprimimos.

Sumando estas operaciones nuestra complejidad final es:

$$O(n^2) + O(n^2) + O(n^2) + O(n) = O(n^2)$$

3.5 Demostración de correctitud

Nuestro problema plantea un escenario donde deseamos ir de la estación 1 a la estación n en el menor tiempo posible. Tomamos cada estación como un nodo del grafo y el trayecto entre dos estaciones como una arista. Utilizamos como el peso de estas aristas el tiempo que toma ir de una estación a otra. Luego, el tiempo que toma recorrer un camino entre estaciones es la suma del peso de sus aristas.

Teniendo en cuenta esto, podemos aplicar algún algoritmo que busque el camino mínimo en nuestro grafo. El algoritmo de Dijkstra nos va a permitir esto. Podemos aplicarlo al ser nuestro grafo dirigido y sin ejes negativos. Cada vez que visita un nodo nos asegura que ya encontro el camino mínimo al mismo, por lo tanto podemos dejar de buscar una vez que nuestro algoritmo visita el nodo que representa la salida de la fortaleza.

Para que el algoritmo pueda visitar un nodo tiene que ser el nodo no visitado con menor distancia al nodo origen. Esta distancia parcial se calcula con los caminos que se pueden formar con los nodos visitados. Si un nodo puede ser visitado entonces no es posible que exista un camino más corto hacia él, sino ese camino hubiera sido descubierto antes debido a que el algoritmo siempre visita los nodos más cercanos.

3.6 Experimentos y conclusiones

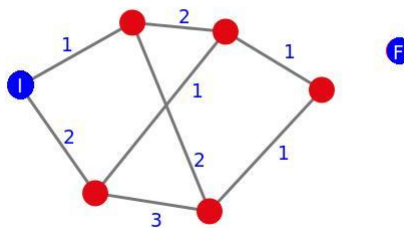
3.6.1 Test

Para corroborar el correcto funcionamiento de nuestro algoritmo implementado desarrollamos los siguientes tests:

Caso 1: Sin solución

Este caso se da cuando el nodo que representa a la estación final queda aislado

El grafo que representa a este tipo es de la siguiente forma:

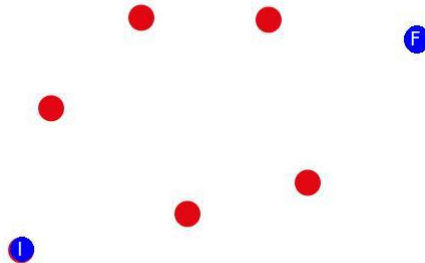


Ejemplo 3.1 - Caso Sin Solución

Caso 2: Sin ejes

Esta familia de casos se da cuando todos los nodos que representa a la estaciones quedan aislados

El grafo que visualiza a este tipo presenta la siguiente forma:

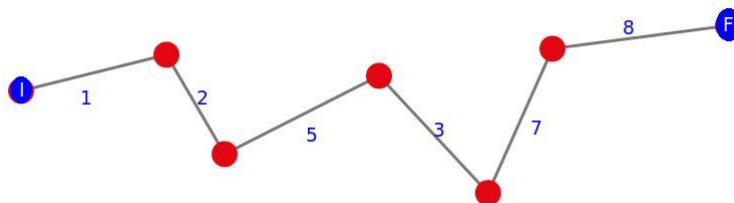


Ejemplo 3.2 - Caso Sin Ejes

Caso 3: Camino simple

Este caso se da cuando el grafo que se recibe como parámetro ya presenta un camino simple armado desde la estación inicial a la final

El grafo que describe a esta familia de casos es el siguiente:

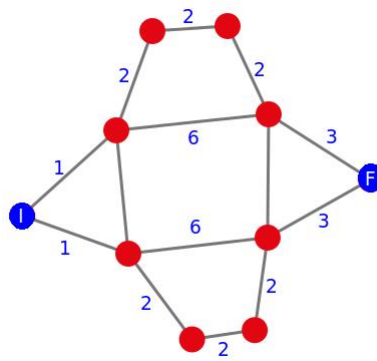


Ejemplo 3.3 - Caso Camino Simple

Caso 4: Múltiples caminos de igual peso llegan a destino

Este caso se da cuando existen varias ramas dentro del grafo que van desde el nodo inicial hasta el último y la suma de dichos pesos termina siendo igual. Veremos más adelante que por la implementación y desarrollo de nuestro algoritmo este terminará siendo el peor caso.

El grafo que describe a esta familia de casos es el siguiente:



Ejemplo 3.4 - Caso Con Múltiples Caminos

Caso 5 Random

Este caso se da cuando se recibe grafos con aristas y nodos sin ninguna particularidad.

3.6.2 Performance del algoritmo

Acorde a lo solicitado, mostraremos distintos tipos de familias de casos para nuestro algoritmo, y además, daremos el tiempo estimado según la complejidad del algoritmo calculada anteriormente.

A continuación mostraremos un gráfico de tiempos comparativo entre distintas familias de casos:

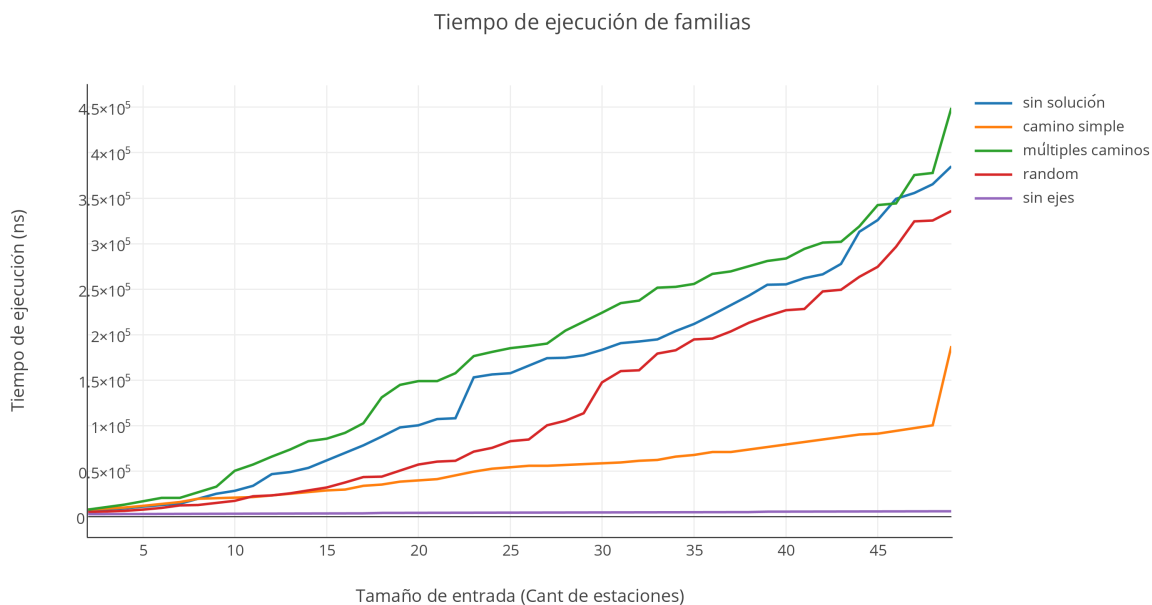


Gráfico 3.1 - Comparativo

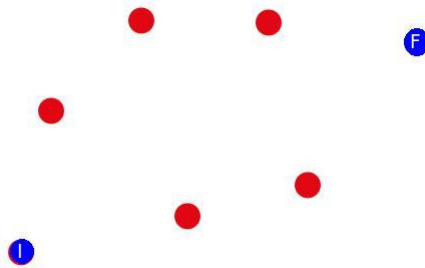
Se puede observar en el gráfico, cuatro funciones las cuales representan el tiempo de ejecución de las familias de casos:

- No hay ejes, todos los nodos desconectados
- El grafo recibido como parametro es el camino simple
- Sin solución, todos los nodos conectados salvo el ultimo
- Random
- Multiples caminos llegan a destino con el mismo valor

Como se observa en el gráfico la función representativa de la flia número 1, presenta una mejor performance en relación a las otras. Esto se debe a que nuestro algoritmo intenta chequear las aristas para armar los caminos y como encuentra que no hay ningun camino de ningun nodo hacia otro finaliza su ejecución demandando unicamente la creación del grafo.

Luego de chequear dichas instancias, pudimos llegar a la conclusión que la familia de casos que presenta una mejor performance para nuestro algoritmo es en el cual **No hay ejes, todos los nodos desconectados**

Un grafo representativo de lo dicho sería el siguiente:



Grafo 3.1 - Mejor Caso

Para llegar a dicha conclusión trabajamos con 50 instancias.

Para una mayor observación desarrollamos el siguiente gráfico con las instancias:

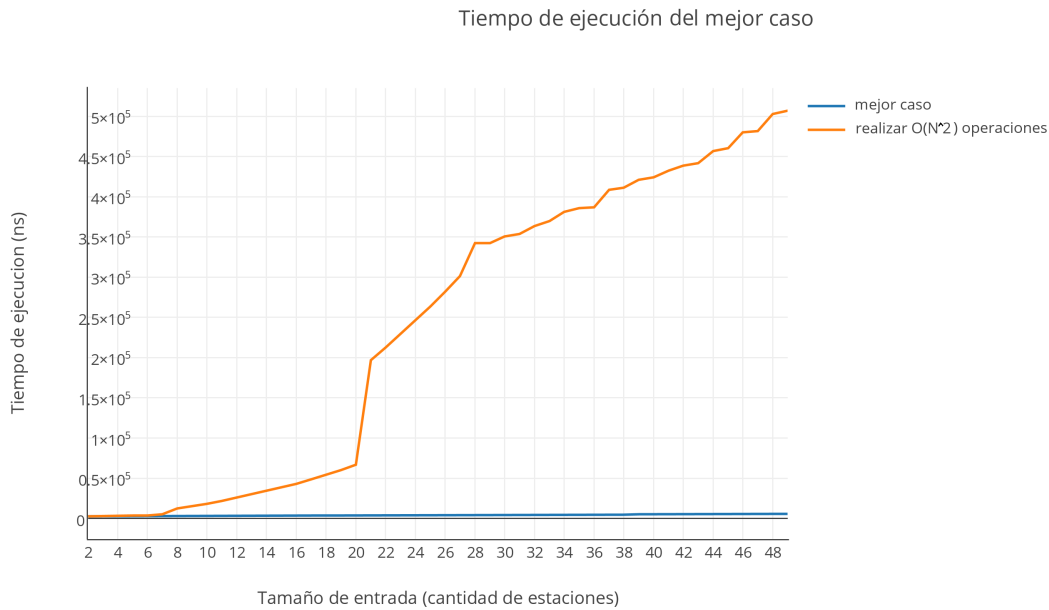


Gráfico 3.2 - Mejor Caso

Como es posible observar en el gráfico, la función resultante de la cota teorica crece mucho más rápido que la de nuestro algoritmo la cual queda considerablemente por debajo . Es por esto que graficamos el mismo contra la función de $O(N * k) \text{ con } k < N$

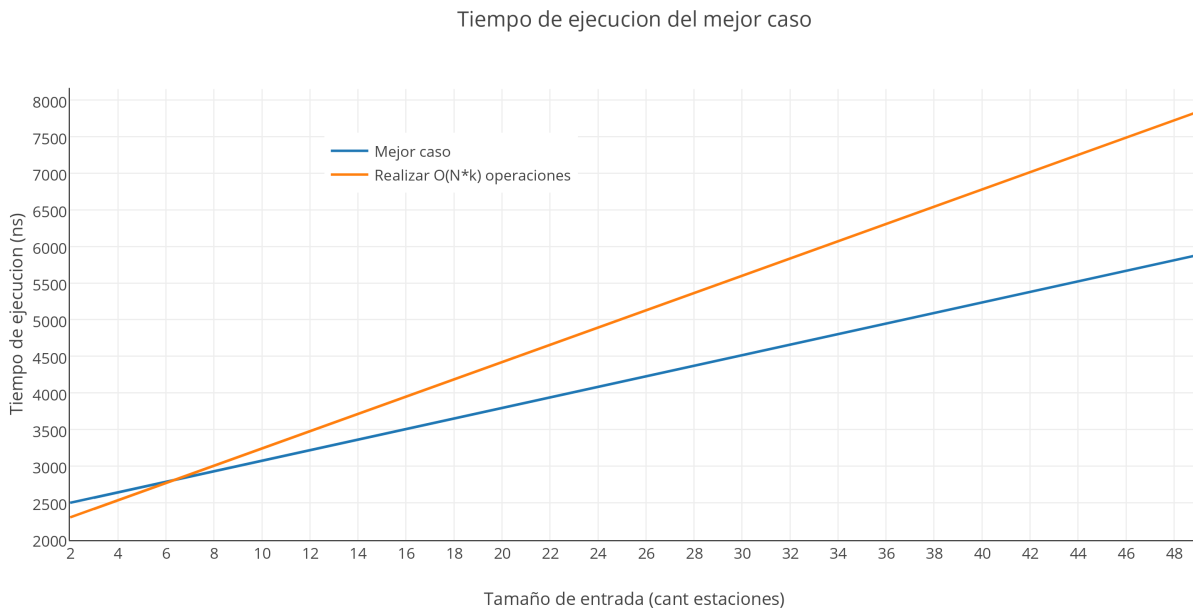


Gráfico 3.3 - Mejor Caso

Se puede observar como la función resultante de realizar $O(N*k)$ operaciones es similar a la de nuestro algoritmo mostrando que en el mejor caso nuestro algoritmo trabaja en el orden de $O(N)$. Luego, dividiendo por la complejidad teorica de nuestro algoritmo llegamos a:

Tiempo de ejecución del mejor caso sobre la complejidad teórica

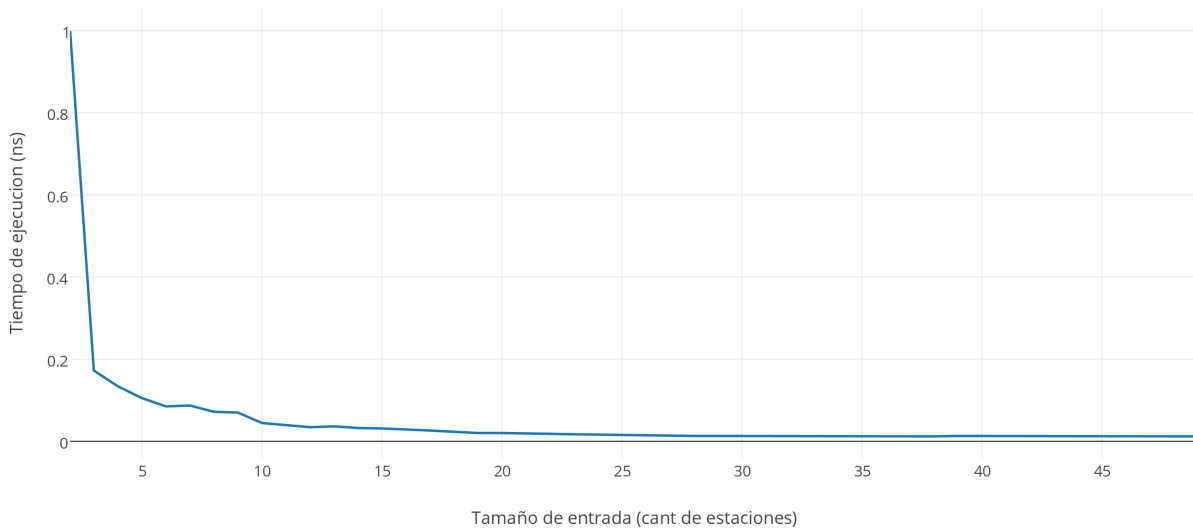
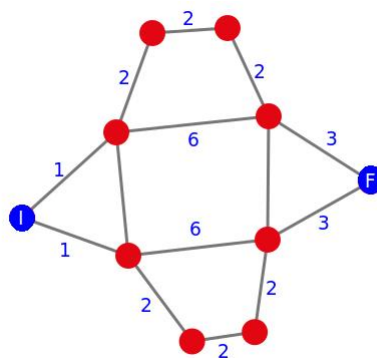


Gráfico 3.4 - Mejor Caso / Complejidad $O(N^2)$

Para realizar esta experimentación nos pareció prudente, realizar un promedio con el mismo input de aproximadamente 20 corridas tanto para la complejidad como para nuestro algoritmo y una vez calculado dicho promedio de ambas cosas realizamos la división para obtener resultados más relevantes.

Se puede observar en el gráfico 3.4, como luego de realizar la división por la complejidad se ve que la función resultante tiende a 0 ya que nuestro algoritmo en este tipo de caso trabaja en el orden de $O(N)$ lo cual es mucho menos que la complejidad teórica. Por lo tanto, podemos concluir que para el mejor caso nuestro algoritmo se encuentra considerablemente por debajo de la cota teórica $O(N^2)$.

Luego, verificando el peor caso, llegamos a la conclusión que la familia de casos en el que resulta menos beneficioso trabajar con nuestro algoritmo será cuando **el grafo que se obtiene de transformar el circuito de estaciones de entrada es aquel que presenta multiples caminos para llegar a destino donde la suma de estos caminos presentan el mismo valor**, dandonos el siguiente grafo una vez transformado:



Grafo 3.2 - Peor Caso

Realizando experimentos con un total de 50 instancias, desarrollamos dos gráficos los cuales

mostraremos a continuación:

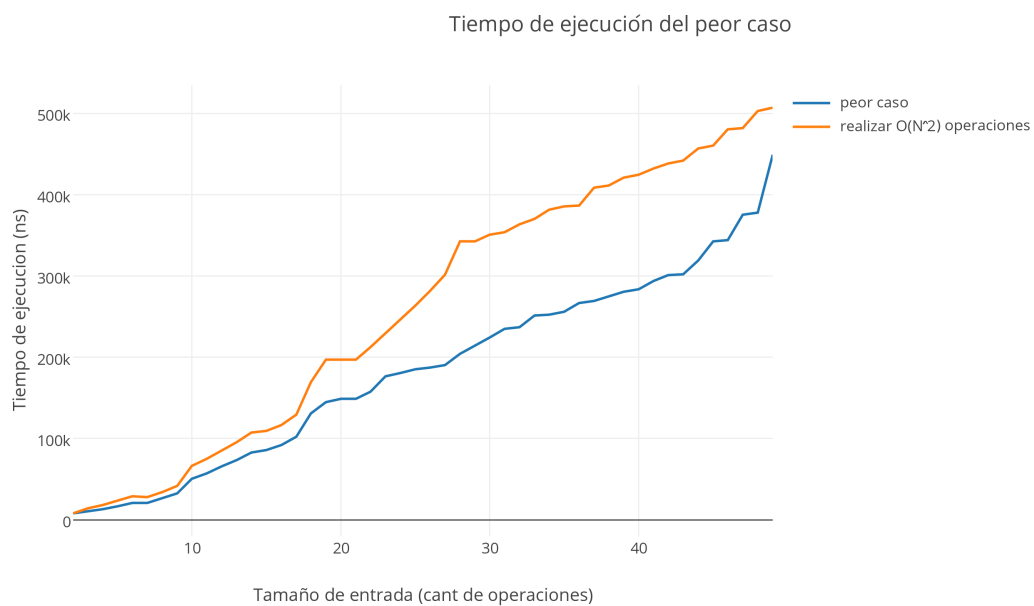


Gráfico 3.5 - Peor Caso

Dividiendo por la complejidad propuesta llegamos a:

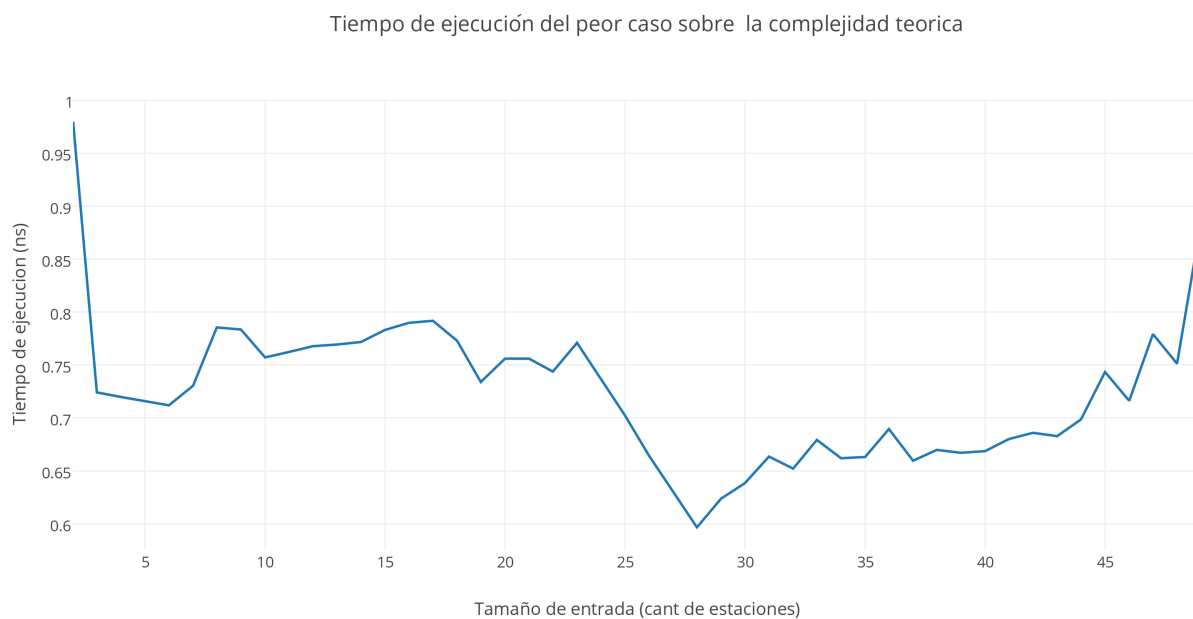


Gráfico 3.6 - Peor Caso / Complejidad $O(N^2)$

Para realizar esta experimentación nos pareció acorde, realizar un promedio con el mismo input de aproximadamente 20 corridas tanto para la complejidad como para nuestro algoritmo y una vez

calculado dicho promedio de ambas cosas realizamos la división para obtener resultados más relevantes.

Como se puede observar en el gráfico 3.5, la función resultante de nuestro algoritmo es considerablemente mejor que la de la cota teorica y presenta un tiempo similar al de la función resultante de la cota. Luego, en el gráfico 3.6 se ve como la función resultante presenta un máximo muy cerca de 1 y cuando el N aumenta la misma tiende a 0.

Por último, mostraremos un gráfico comparativo entre el mejor y peor caso contra la complejidad que se solicita:

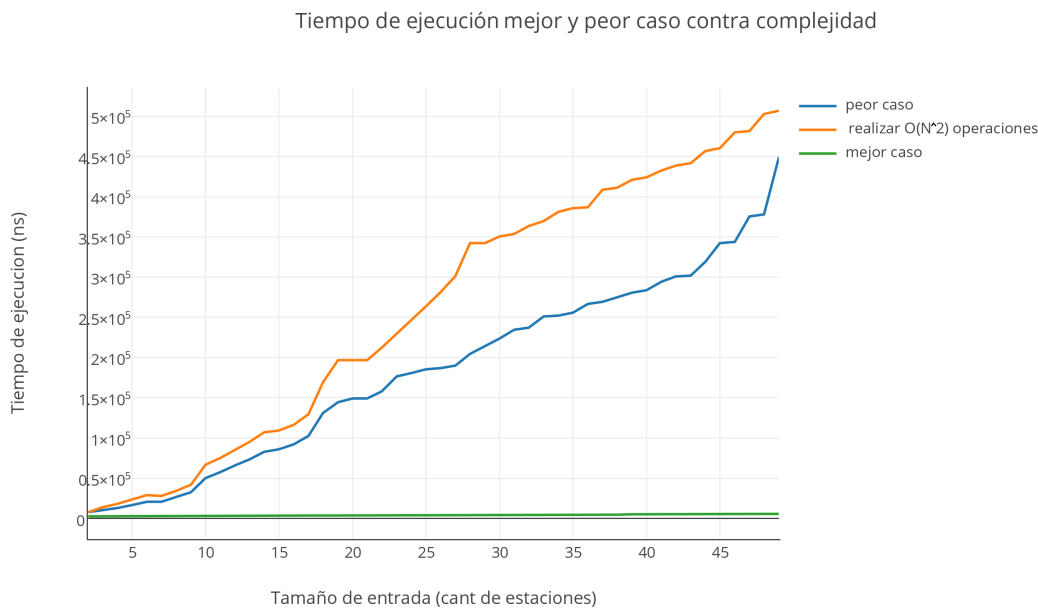


Gráfico 3.7 - Comparativo

Luego de dichos experimentos y casos probados, se puede concluir que a pesar de tener ciclos en todas las salas y donde dichos ciclos presenten aristas con pesos iguales lo que generara al algoritmo la posibilidad de crear varias ramas posibles de solución nos mantenemos dentro de la complejidad propuesta como habíamos mostrado en nuestro desarrollo de la complejidad.

4 Aclaraciones

4.1 Aclaraciones para correr las implementaciones

Cada ejercicio fue implementado con su propio Makefile para un correcto funcionamiento a la hora de utilizar el mismo.

El ejecutable para el ejercicio 1 sera ej1 el cual recibirá como se solicito entrada por stdin y emitirá su respectiva salida por stdout.

Tanto el ejercicio 2 como el 3, compilarán de la misma forma y podrán ser ejecutados con ej2 y ej3 respectivamente.

A su vez, para poder chequear las mediciones de tiempo y nuestros casos de testeo se creo una carpeta nueva por cada ejercicio. Dentro de cada una se cuenta con el respectivo makefile que compila todos los test pudiendo así probar cada uno por separado.