



DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Técnicas de Diseño de Algoritmos

27 de septiembre de 2013

Algoritmos y Estructuras de Datos III
Reentrega

Grupo E

Integrante	LU	Correo electrónico
Barrios, Leandro E.	404/11	ezequiel.barrios@gmail.com
Fernández Wortman, Maximiliano	892/10	maxifwortman@gmail.com
Gasco, Emilio	171/12	gascoe@gmail.com
Di Lorenzo, Leandro	101/06	leandro.jdl@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	3
1.1. Objetivos	3
1.2. Metodología a implementar	3
2. Desarrollo del TP	4
2.1. Problema 1: Pascual y el correo	4
2.1.1. Descripción	4
2.1.2. Hipótesis de resolución	5
2.1.3. Justificación formal de correctitud	6
2.1.4. Cota de complejidad temporal	8
2.1.5. Verificación mediante casos de prueba	8
2.1.6. Medición empírica de la performance	9
2.2. Problema 2: Profesores visitantes	13
2.2.1. Descripción	13
2.2.2. Hipótesis de resolución	13
2.2.3. Justificación formal de correctitud	15
2.2.4. Cota de complejidad temporal	16
2.2.5. Verificación mediante casos de prueba	17
2.2.6. Medición empírica de la performance	18
2.3. Problema 3: Una noche en el museo	20
2.3.1. Descripción	20
2.3.2. Hipótesis de resolución	21
2.3.3. Justificación formal de correctitud	26
2.3.4. Cota de complejidad temporal	26
2.3.5. Verificación mediante casos de prueba	27
2.3.6. Medición empírica de la performance	29
3. Apéndices	32
3.1. Código Fuente (resumen)	32
3.1.1. Problema 1: Pascual y el correo	32
3.1.2. Problema 2: Profesores visitantes	33
3.1.3. Problema 3: Una noche en el museo	35
3.2. Informe de Modificaciones	40
3.3. Bibliografía	41

1. Introducción

1.1. Objetivos

El presente trabajo pretende realizar un análisis de algunas de las diferentes técnicas algorítmicas utilizadas para la resolución de problemas de optimización o para analizar universos de soluciones en tiempos razonables. Principalmente nos enfocamos en las técnicas de *algoritmos golosos* y de *backtracking*. Bajo la resolución de problemas concretos se midieron complejidades y correctitudes de los procedimientos descriptos y se verificaron mediante casos de prueba.

1.2. Metodología a implementar

Para la realización de este trabajo utilizamos las siguientes herramientas:

- C++ como lenguaje de programación,
- compilado bajo `gcc`,
- `matplotlib` para armar los gráficos,
- `LATEX` para la composición del documento,
- bajo los siguientes Sistemas Operativos
 - Debian GNU/Linux
 - Ubuntu
 - FreeBSD

2. Desarrollo del TP

2.1. Problema 1: Pascual y el correo

2.1.1. Descripción

Pascual es el encargado de ordenar los paquetes en un servicio de correo. Su tarea consiste en recolectar los paquetes desde una cola, es decir, a medida que los mismos van llegando, y acomodarlos en los distintos camiones que luego se encargarán de transportarlos. Sabemos que cada paquete tendrá un peso determinado, el cual se nos presenta como un número entero positivo, y que cada camión soporta una carga máxima, representada de la misma forma. Además, se nos asegura que no puede existir ningún paquete de forma tal que este supere la carga máxima que soportan los camiones, de lo cual se deduce que dado m el peso máximo que soporta un camión, el peso de cualquier paquete de la cola estará contenido en el rango $[1...m]$.

Se nos indica que Pascual cuenta con un criterio determinado de antemano, el cual utilizará para acomodar los paquetes en los camiones: dado un paquete, acomodará el mismo en uno de los camiones cargados, seleccionando específicamente de todos los que haya disponibles el que menos carga presente en ese momento, siempre que el peso del mismo no supere la capacidad disponible del camión en cuestión; caso contrario, utilizará un nuevo camión.

Los jefes de Pascual desean averiguar de antemano la carga que contendrá cada camión luego de que todos los paquetes sean acomodados. Se nos plantea diseñar una solución a este problema, y que la misma esté acotada por una complejidad estrictamente menor a $\mathcal{O}(n^2)$.

Llamaremos «algoritmo pascual» al que, dada una cola de paquetes, pretende (sin éxito) minimizar la cantidad de camiones a utilizar (potencialmente infinitos) eligiendo en cada paquete aquel camión en uso con menos carga. Si no lo consigue, el método indica que debe poner en uso un nuevo camión. Los paquetes se cargan en el orden en que vienen dados, y cada paquete tiene un peso mayor estricto que cero, y menor o igual que la capacidad de un camión (todos tienen el mismo máximo de carga). Los paquetes se cargan enteros, es decir, un mismo paquete no puede «ser dividido en distintos camiones». A efectos prácticos, aunque no lo sepamos, de aquí en adelante admitiremos que Pascual efectúa la elección del camión en forma lineal, para intentar mejorarlo.

Por lo general, los algoritmos golosos se distinguen por contar con:[1, p. 188][7]:

- Un conjunto de candidatos
- Un método de selección
- Un indicador de factibilidad
- Una función objetivo
- Una función de solución

Por ejemplo podría suceder lo siguiente: se disponen de infinitos camiones, de forma tal que cada camión c_i soporta una carga máxima de $100kg$, y una lista de paquetes de peso p_j . Una carga posible podría ser $\langle 30, 50, 90, 5, 80, 12, 20 \rangle$ haciendo que se genere el siguiente proceso de carga:

Carga de Camiones		
p_1	$c_1 \leftarrow 30$	$c_1 = 30$
p_2	$c_1 \leftarrow 50$	$c_1 = 80$
p_3	$c_2 \leftarrow 90$	$c_1 = 80, c_2 = 90$
p_4	$c_1 \leftarrow 5$	$c_1 = 85, c_2 = 90$
p_5	$c_3 \leftarrow 80$	$c_1 = 85, c_2 = 90, c_3 = 80$
p_6	$c_1 \leftarrow 12$	$c_1 = 97, c_2 = 90, c_3 = 80$
p_7	$c_3 \leftarrow 20$	$c_1 = 97, c_2 = 90, c_3 = 100$

Como mejor caso podríamos recibir $\langle 10, 20, 8, 17, 12, 11, 2 \rangle$ entrando todo en un solo camión:

Carga de Camiones		
p_1	$c_1 \leftarrow 10$	$c_1 = 10$
p_2	$c_1 \leftarrow 20$	$c_1 = 30$
p_3	$c_1 \leftarrow 8$	$c_1 = 38$
p_4	$c_1 \leftarrow 17$	$c_1 = 55$
p_5	$c_1 \leftarrow 12$	$c_1 = 67$
p_6	$c_1 \leftarrow 11$	$c_1 = 78$
p_7	$c_1 \leftarrow 2$	$c_1 = 80$

O nos podría llegar uno de los peores casos $\langle 50, 80, 60, 45, 75, 90, 15 \rangle$ debiendo utilizar tantos camiones como paquetes:

Carga de Camiones		
p_1	$c_1 \leftarrow 10$	$c_1 = 10$
p_2	$c_2 \leftarrow 80$	$c_1 = 10, c_2 = 80$
p_3	$c_3 \leftarrow 60$	$c_1 = 10, c_2 = 80, c_3 = 60$
p_4	$c_4 \leftarrow 45$	$c_1 = 10, c_2 = 80, c_3 = 60, c_4 = 45$
p_5	$c_5 \leftarrow 75$	$c_1 = 10, c_2 = 80, c_3 = 60, c_4 = 45, c_5 = 75$
p_6	$c_6 \leftarrow 90$	$c_1 = 10, c_2 = 80, c_3 = 60, c_4 = 45, c_5 = 75, c_6 = 90$
p_7	$c_7 \leftarrow 15$	$c_1 = 10, c_2 = 80, c_3 = 60, c_4 = 45, c_5 = 75, c_6 = 90, c_7 = 15$

El último ejemplo además sirve para mostrar que Pascual no está minimizando la cantidad de camiones. Los paquetes 50 y 45 podrían haberse cargado juntos, al igual que los paquetes 80 y 15, utilizando dos camiones menos.

2.1.2. Hipótesis de resolución

El algoritmo que Pascual emplea para (su) minimización de camiones es goloso porque cumple con las características:

- Un conjunto de candidatos: los camiones.
- Un método de selección: elegir el de menor carga a cada momento.
- Un indicador de factibilidad: cuando no entra en el menor, entonces no entra en ningún otro.
- Una función objetivo: a cada paquete, los camiones usados son solución parcial.
- Una función de solución: al no quedar paquetes tenemos los camiones.

de manera que se garantiza que se obtiene alguna solución al problema. Como este procedimiento no logra garantizar que los sub-problemas que quedan luego de una elección exhiban una subestructura óptima, no se puede garantizar que la solución brindada sea óptima[2, p. 381], tal como se mostró en el ejemplo de peor caso en la sección anterior. En este caso, el no existir solución sería contradictorio con la premisa de pesos de los paquetes. Al tener cada paquete un peso perteneciente al rango entre cero y la capacidad del camión, si existe por lo menos un paquete, seguro que entra en un camión.

El problema a tratar no es obtener un algoritmo óptimo en la minimización de camiones, sino mejorar la complejidad del «algoritmo pascual», quien satisface el problema con una complejidad temporal $\mathcal{O}(n^2)$ para n paquetes. Como los paquetes se recorren linealmente en orden (de llegada), esto implica que la elección del camión con menos carga se está haciendo también linealmente. El peor caso de camiones se da cuando cada paquete tiene un peso mayor estricto que la mitad de la capacidad de carga de los camiones o bien que para cualquier par de paquetes, la suma de sus pesos excede la capacidad máxima. También puede pasar, como en el ejemplo de la sección anterior, que un caso sea peor caso para Pascual pero no para la posible minimización de camiones. Entonces la cantidad de camiones a comparar, en el peor de los casos es n (uno por cada paquete). Los paquetes son finitos, los camiones si bien virtualmente infinitos, no van a ser mayor a la cantidad de paquetes.

Dado que siempre nos interesa el camión con menor carga, la búsqueda secuencial sobre estructuras lineales pierde sentido en pos de otras más provechadas, como ser una cola de prioridad. Entre las ventajas que tiene la cola de prioridad se pueden mencionar la obtención del mínimo elemento en tiempo constante, y agregar un elemento y modificar el mínimo en orden logarítmico. Modificar el mínimo implica quitar la raíz (generando una reestructuración de la cola) y volver a insertarlo modificado. Esto genera dos operaciones del orden de $\mathcal{O}(\log n)$.

El *Algoritmo 1* muestra el procedimiento llevado a cabo para ejecutar la tarea de Pascual con complejidad $\mathcal{O}(n \log n)$, dónde n está representado por *cant* (cantidad de paquetes que necesitan ser cargados). Si bien los paquetes llegan por la entrada estándar, a los fines del análisis podemos suponer que están representados en un arreglo, una lista o cualquier estructura que permita la iteración secuencial y que permita leer el dato en tiempo constante.

Los camiones utilizan dos estructuras, una lista y una cola de prioridad. Se usan estructuras provistas por la `stl` de C++ [3]. Para la lista se utiliza `std::list` y para la cola `std::priority_queue`. Quién asume la responsabilidad de la complejidad es la cola de prioridad, ya que es sobre ella que se genera la toma de decisiones. La inserción en la lista se puede realizar $\mathcal{O}(1)$ porque no se hace una inserción con prioridad, sino que se agrega al final. La lista guarda los camiones en el orden en que fueron utilizados por primera vez, lo cual es requisito de la devolución del problema. La cola de prioridad al realizar `INSERTAR CON PRIORIDAD` produce la remoción de la raíz, reacomodando y luego reinsertando con el valor actualizado. Como máximo se recorre la altura del árbol, o sea $\mathcal{O}(\log n)$. A los fines prácticos el empate de pesos en los paquetes se define por menor «*id*» (primero en usarse, más chico). En el peor de los casos van a existir n camiones, uno por cada paquete.

2.1.3. Justificación formal de correctitud

Tal como fuimos describiendo a medida que avanzábamos en el desarrollo del algoritmo, este problema fue planteado por la cátedra con un funcionamiento correcto. Los paquetes llegan en orden en una estructura *iterable*, y luego son recorridos. Cada paquete presenta la posibilidad de ser cargado en un camión, ya sea alguno en uso o uno nuevo. Como los pesos de los paquetes «encajan» en los camiones (expuesto de forma expresa en el propio enunciado), siempre resulta factible conseguir un camión. Esto nos garantiza que al terminar de recorrer todos los paquetes, estos van a encontrarse cargados en algún camión, y que todo

Algoritmo 1 Pascual Logarítmico

Entrada:

limite \leftarrow CARGA MÁXIMA \triangleright integer
cant \leftarrow CANTIDAD DE PAQUETES \triangleright integer
pesosPaquetes \leftarrow LISTA DE PESOS DE PAQUETES \triangleright array<integer>

Salida:

CANTIDAD DE CAMIONES \triangleright integer
 LISTA DE PESOS CARGADOS EN CAMIONES \triangleright std :: vector<integer>

```

1: camionesOcupados  $\leftarrow$  0
2: listaCamiones  $\leftarrow$  NUEVA LISTA  $\triangleright$  std :: list<Camion>
3: colaCamiones  $\leftarrow$  NUEVA COLA DE PRIORIDAD  $\triangleright$  std :: priority_queue<Camion>,  $\mathcal{O}(1)$ 
4: para i desde 0 hasta cant – 1 hacer
5:   peso  $\leftarrow$  pesosPaquetes[i]  $\triangleright$   $\mathcal{O}(1)$ 
6:   si HAY LUGAR(peso) entonces  $\triangleright$  chequeo  $\mathcal{O}(1)$ 
7:     AGREGAR CARGA(peso)  $\triangleright$   $\mathcal{O}(\log n)$ 
8:   sino
9:     NUEVO CAMIÓN(peso)  $\triangleright$   $\mathcal{O}(\log n)$ 
10:  fin si
11: fin para  $\triangleright$  ciclo  $\mathcal{O}(n)$ 
12: retornar camionesOcupados, CARGAS(listaCamiones)  $\triangleright$  final  $\mathcal{O}(n \log n)$ 

13: función HAY LUGAR(peso)
14:   camionMenosCargado  $\leftarrow$  TOPE DE COLA(colaCamiones)  $\triangleright$   $\mathcal{O}(1)$ 
15:   carga  $\leftarrow$  CARGA(camionMenosCargado)
16:   retornar camionesOcupados > 0  $\wedge$  carga + peso  $\leq$  limite
17: fin función  $\triangleright$  final  $\mathcal{O}(1)$ 

18: procedimiento AGREGAR CARGA(peso)
19:   camionMenosCargado  $\leftarrow$  TOPE DE COLA(colaCamiones)  $\triangleright$   $\mathcal{O}(1)$ 
20:   SACAR TOPE(colaCamiones)  $\triangleright$   $\mathcal{O}(\log n)$ 
21:   SUMAR CARGA(camionMenosCargado, peso)  $\triangleright$   $\mathcal{O}(1)$ 
22:   INSERTAR CON PRIORIDAD(colaCamiones, camionMenosCargado)  $\triangleright$   $\mathcal{O}(\log n)$ 
23: fin procedimiento  $\triangleright$  final  $\mathcal{O}(\log n)$ 

24: procedimiento NUEVO CAMIÓN(peso)
25:   nuevoCamion  $\leftarrow$  CAMION(peso, camionesOcupados + 1)  $\triangleright$   $\mathcal{O}(1)$ 
26:   listaCamiones[camionesOcupados]  $\leftarrow$  nuevoCamion  $\triangleright$   $\mathcal{O}(1)$ 
27:   INSERTAR CON PRIORIDAD(colaCamiones, nuevoCamion)  $\triangleright$   $\mathcal{O}(\log n)$ 
28:   camionesOcupados  $\leftarrow$  camionesOcupados + 1
29: fin procedimiento  $\triangleright$  final  $\mathcal{O}(\log n)$ 

```

camión usado tendrá al menos un paquete, ya que la forma de «inicializarlo» es mediante la asignación de una carga. Por todo lo expuesto, debido a que el objetivo del problema es minimizar el gasto temporal de un algoritmo concreto, y no el de investigar o desarrollar un mecanismo de distribución de paquetes, es que asumimos que el algoritmo cumple con su objetivo.

2.1.4. Cota de complejidad temporal

En el algoritmo *Pascual Logarítmico* se recorren los n paquetes y se insertan en el mejor camión disponible en ese momento ($\log n$), tomando una complejidad final de $\mathcal{O}(n \log n)$.

El «algoritmo pascual» itera linealmente tanto sobre los paquetes como sobre los camiones. De esta manera genera una complejidad de $\mathcal{O}(n^2)$. En el peor caso se tendrán n camiones (uno por cada paquete), donde eso indica que en ningún caso se puede meter más de un paquete en un camión. Esto implica que ante cada nuevo paquete se recorren todos los camiones existentes sin conseguir lugar, necesitando uno nuevo.

Nuestro algoritmo respeta el procedimiento pero introduce una mejora en la búsqueda de camiones. En lugar de iterar sobre todos los cargados, se los mantiene dentro de una cola de prioridad, permitiendo que la lectura del menos cargado sea en $\mathcal{O}(1)$ y la inserción o reinserción tome una complejidad de $\mathcal{O}(\log n)$. La re inserción se efectúa cuando se carga un paquete en un camión no vacío. Es esos casos, la remoción se hace en tiempo logarítmico y luego se lo reinserta con la misma complejidad. En total, la re inserción efectúa dos operaciones logarítmicas consecutivas, siendo del orden $\mathcal{O}(\log n)$. La justificación de la complejidad temporal se logra gracias a las propiedades de la estructura de la cola prioritaria que almacena los elementos en un árbol binario «heapificado» [1, cap. 5.7]. La cota temporal inmediata está indicada por la propia documentación de la estructura `priority_queue` [6], parte de la STL de C++. La complejidad final termina resultando en un orden $\mathcal{O}(n \log n)$, dado que por cada paquete a lo sumo se efectúan dos operaciones logarítmicas.

2.1.5. Verificación mediante casos de prueba

Para verificar el correcto funcionamiento de nuestro programa tomamos una serie de instancias que consideramos representativas de casos extremos o sensibles y analizamos el comportamiento. Definimos los siguientes casos:

in1: 0 0	out1: 0
in2: 1 0	out2: 0
in3: 1 1 1	out3: 1 1
in4: 1 2 1 1	out4: 2 1 1
in5: 10 10 1 1 1 1 1 1 1 1 1 1	out5: 1 10
in6: 10 10 1 2 1 2 1 2 1 2 1 2	out6: 2 10 5
in7: 10 10 0 2 1 2 0 2 1 2 1 2	out7: 2 10 3
in8: 20 5 9 12 14 19 16	out8: 5 9 12 14 19 16
in9: 20 5 10 11 12 13 14	out9: 5 10 11 12 13 14

Los casos 1 y 2 muestran los escenarios en dónde no hay paquetes a cargar, con lo cual no se cargan camiones. Los casos 8 y 9 se refieren a las condiciones dadas para los peores casos, en dónde cada paquete pesa más de la mitad del límite de carga o bien, todo par de paquetes pesa más que el límite. El resto de los casos no son situaciones extremas pero sirven para analizar el comportamiento del algoritmo.

A su vez se realizaron pruebas aleatorias que además de verificar el comportamiento se utilizaron para mostrar gráficamente que los resultados empíricos se condicen con el análisis teórico. A continuación se pueden ver reflejados los resultados.

2.1.6. Medición empírica de la performance

Para poder realizar un muestreo significativo se tomaron de 500 a 100 muestras para cada X , dónde X representa la cantidad de paquetes. Se hizo variar a la cantidad X entre 1 y 20000. Para cada valor se tomaron dos muestreos, primeramente enfocandonos en los peores casos (en que la utilización de camiones resultaba máxima). Esta mdición se corresponde con *Figura 1*, y se eligió marcar a los distintos muestreos como puntos grises, que al superponerse permiten captar de forma efectiva el espectro de densidad de la distribución muestral. Sobre este mismo gráfico, se superpuso una línea blanca, representando el promedio de las mediciones obtenidas para cada X . La línea punteada, finalmente, es la cota teórica $\mathcal{O}(n \log n)$, multiplicada por una constante muy pequeña, del orden de $44e - 9$ ¹.

La primera eliminación de outliers se realizó calculando la desviación estándar, y descartando los valores que se encontraban a una distancia al promedio superior al doble de la misma. Para tomar las mediciones se utilizó un bucle cuya guarda comprobaba la cantidad de «mediciones buenas», tomando mediciones indefinidamente hasta alcanzar un nivel de muestreo confiable (establecido arbitrariamente, luego de sucesivas pruebas).

Asumimos aquí que una «buena» forma de descartar outliers de un conjunto muestral es asumir que los datos cumplen con la distribución probabilística de una función normal; entonces, se realiza el cálculo de la esperanza/promedio, y la distribución estándar, descartando posteriormente aquellos que se encuentran en un rango mayor a $2 * std$ alrededor de u (siendo u es la esperanza y std el desvío estándar).

Luego de consultar con diversos docentes, esta medición fue posteriormente descartada (aunque se había estimado que la cantidad y la calidad de los datos descartados en el proceso era despreciable). Se repitió la muestra con un mecanismo similar: las muestras fueron tomadas con un mecanismo iterativo: para cada X se tomaron N muestras, y luego se calcularon la esperanza y la $stdev$, al igual que antes, «descartando del conteo» (pero guardándolas en el muestreo) las muestras que se salieran del rango. Llegado un punto, se entra en un proceso iterativo en el que se recalculan sucesivamente la esperanza y $stdev$, hasta que la cantidad de «muestras aceptables» supera en gran medida a la cantidad de «muestras desviadas». Dicho de otra forma, se estableció un sistema de puntajes positivos/negativos, de forma tal que mientras las «muestras buenas» sumaban puntaje a nuestro «contador de muestreo», las «muestras malas» restaban, obligandonos a tomar mayor cantidad de muestras en los muestreos más «irregulares». Para dar un ejemplo, si nuestro muestreo objetivo era de 100 muestras buenas, y nos encontrábamos con 10 «muestras malas», antes de comenzar el muestreo de $X + 1$, tomábamos 10 muestras «válidas» más (esto significa que quizás terminábamos tomando 15 muestras más, suponiendo que de las nuevas muestras, 5 salieran «malas»).

Se repitió el proceso para todos los X , con la principal diferencia de que ahora se estaban guardando **todas** las muestras, independientemente si eran «malas» o «buenas», a diferencia del primer intento..

Luego de recolectar todos los datos se realizó el gráfico en donde a cada punto se le aplicó un valor alpha de 0.3 (transparencia), logrando un efecto de ¡«mayor densidad» de tonalidad/color en las zonas en que había más ocurrencia de muestras (es decir, en donde las muestras se imprimen una por sobre la otra). Esto se ve en *Figura 1*, y se interpreta la información de la siguiente forma:

¹Esta cota fue establecida arbitrariamente; con constantes mayores la cota teórica queda muy por encima, generando pérdida de visualización gráfica

- En gris se ven las muestras que están más allá de la desviación estándar. Mientras más alejadas estén, más pequeño debería ser el tamaño del punto que las representa. Para esto se hizo una función que variaba el tamaño del punto conforme aumentaba el desvío del mismo.
- En colores más oscuros, tendiendo al negro, se observan las muestras que están dentro del rango de muestras aceptadas como válidas.
- En blanco la función que representa el promedio, según lo detallado previamente.
- En línea punteada la cota teórica multiplicada por una constante muy chica.

Para la figura *Figura 2*, en cambio, debido al largo tiempo que había demorado el primer muestreo, se tomó una cantidad de valores de tiempo mucho más baja para cada valor de X (aproximadamente 20 valores, frente a los 500-100 de la figura 1). Se utilizó la misma técnica de «repetición de muestras malas», y se tomó además un segundo muestreo en donde la entrada fue generada mediante un algoritmo pseudoaleatorio que obteniendo los pesos de los paquetes a través de una función de probabilidad uniforme, generó una tanda de «testcases» ($20 \times 20000 = 400000$ casos de prueba distintos en total) que luego fue aplicada a nuestra versión logarítmica del algoritmo pascual. En este gráfico se representó la muestra de peor caso mediante un grosor de línea muy fino, y la muestra pseudoaleatoria mediante un trazo grueso. Se mantuvo la misma cota teórica que para el ejercicio anterior.

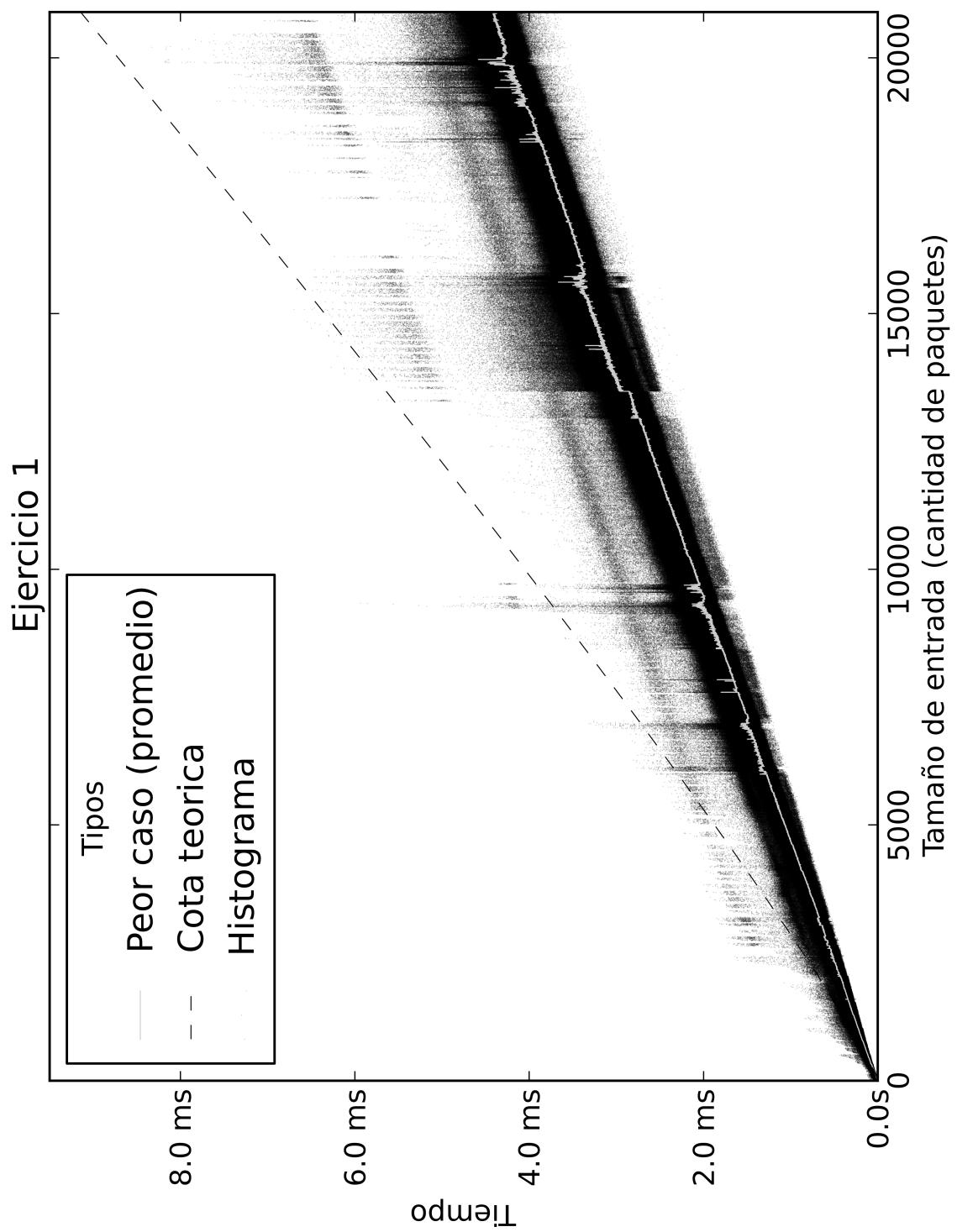


Figura 1: Muestreo del Ejercicio 1

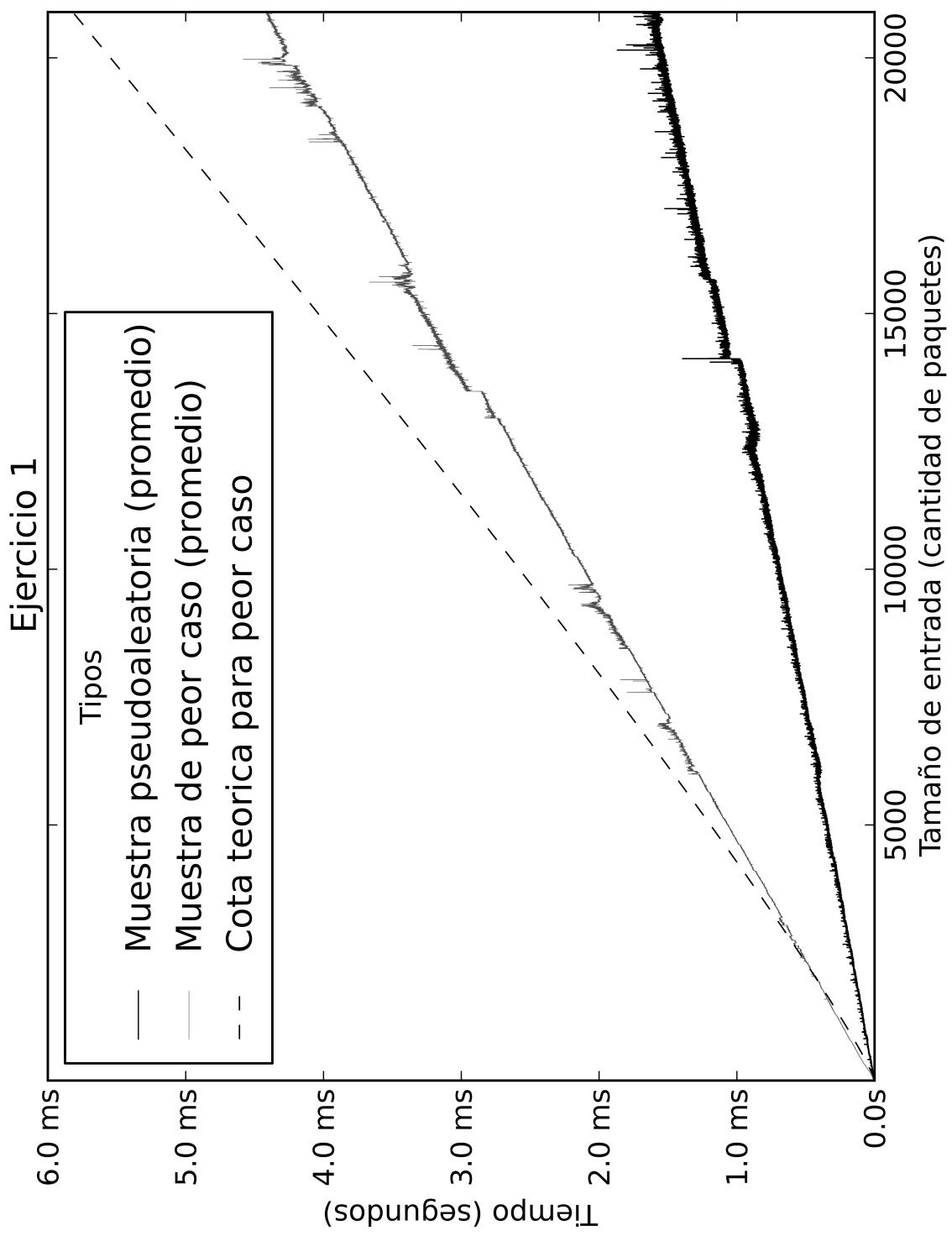


Figura 2: Segundo muestreo del Ejercicio 1

2.2. Problema 2: Profesores visitantes

2.2.1. Descripción

En este problema debemos maximizar la cantidad de cursos a dar a través del programa de Profesores Visitantes que organiza el Departamento de Computación de la facultad. Se invitó a un grupo de prestigiosos profesores extranjeros a que dicten cursos en la facultad. Como es natural, cada profesor que puede concurrir tiene una agenda bastante cargada, con lo cual son ellos quienes establecen la disponibilidad. Esto tiene la desventaja de que seguramente existirán cursos que se solapen en el tiempo, produciendo que si algún alumno quisiera participar en todos ellos, no lo va a poder hacer. Debemos garantizar que se ofrecerán la mayor cantidad de cursos sin que se solapen.

Los datos de los cursos vienen dados como pares $[inicio, fin]$ sin ningún tipo de orden. Una primera aproximación podría ser analizar el universo de soluciones y quedarse con la óptima. Este enfoque resultaría muy caro. Dado que el departamento solicita una cota menor que $\mathcal{O}(n^2)$, tendremos que mejorar el enfoque.

Este tipo de problemas suelen tener buenas soluciones mediante técnicas de algoritmos golosos. Algunas de las posibles estrategias serían: menor fecha de inicio, menor fecha de fin, menor duración. Luego vamos a demostrar que tomar el curso con menor fecha de fin cada vez es la estrategia que maximiza la cantidad de cursos a dictar con complejidad temporal en peor caso $\mathcal{O}(n \log n)$. Un ejemplo:

[1, 6] [4, 5] [2, 3] [7, 9] [9, 10] [10, 12]

Elección de cursos					
Menor fecha de inicio	[1, 6]	[7, 9]	[10, 12]	#3	
Menor fecha de fin	[2, 3]	[4, 5]	[7, 9] [10, 12]	#4	
Menor duración	[2, 3]	[4, 5]	[9, 10]	#3	

2.2.2. Hipótesis de resolución

Para resolver el problema se plantea un algoritmo que a cada momento elige el curso cuya fecha de finalización es menor que todos los demás cursos aún no seleccionados. Si el curso no se solapa con el anterior (con lo cual tampoco se solapa con los anteriores al anterior) entonces es el curso que mejor encaja en ese momento y se lo agrega a los cursos ya seleccionados. Si se solapa, se lo descarta. El procedimiento se repite hasta que no quedan cursos para analizar. En caso de haber dos cursos que finalicen en la misma fecha, se elige aquel que comience antes² (y obviamente no se solape) y en caso de persistir el empate, se elige aquel con menor «id», o sea, menor en orden de entrada.

Nuestro algoritmo procesa el «elegir en cada momento» ordenando los cursos por menor fecha de fin (desempatando como se describió) y luego se los recorre linealmente. Este cambio no modifica el comportamiento pero logra generar un mejor orden de complejidad temporal. Dados n cursos se los puede ordenar en $\mathcal{O}(\log n)$ utilizando *heapsort*. Para ello utilizamos la estructura `set` de la `stl` de C++ cuya complejidad cumple lo propuesto ya que internamente está implementado como un árbol binario de búsqueda [4]. Se puede ver el comportamiento en el *Algoritmo 2*.

²al estar ordenados por fecha de fin, la elección desempate según fecha de inicio es indistinta. Si eligiendo por menor fecha de inicio en el desempate hubiese otro curso más que «entre», entonces éste debiera tener menor fecha de fin, pues sino se solaparía. El desempate por menor fecha de inicio se basa en el hecho de que sea un curso con mayor duración, cubriendo de esta manera, no sólo la mayor cantidad de cursos a dictar sino también la mayor cantidad de días ocupados.

Algoritmo 2 Profesores Visitantes

Entrada:

cant \leftarrow CANTIDAD DE CURSOS OFRECIDOS ▷ integer
listaFechas \leftarrow LISTA DE FECHAS INICIO Y FIN ▷ std :: vector<integer>

Salida:

LISTA DE NÚMEROS DE CURSOS ▷ std :: list<integer>

```

1: conjuntoCursos  $\leftarrow$  CONJUNTO(cant) ▷ std :: set<Curso>

2: función PROCESAR CURSOS(cant, listaFechas) ▷ final  $\mathcal{O}(n \log n)$ 
3:   CARGAR CURSOS(cant, listaFechas) ▷  $\mathcal{O}(n \log n)$ 
4:   retornar SELECCIÓN DE CURSOS ▷  $\mathcal{O}(n)$ 
5: fin función ▷ final  $\mathcal{O}(n \log n)$ 

6: procedimiento CARGAR CURSOS(cant, listaFechas) ▷  $\mathcal{O}(1)$ 
7:   i  $\leftarrow$  0
8:   mientras i < 2 * cant hacer ▷ ciclo  $\mathcal{O}(n)$ 
9:     num  $\leftarrow$  i  $\div$  2
10:    inicio  $\leftarrow$  listaFechas[i]
11:    fin  $\leftarrow$  listaFechas[i + 1]
12:    curso  $\leftarrow$  NUEVO CURSO(num, inicio, fin) ▷  $\mathcal{O}(1)$ 
13:    INSERTAR ORDENADO(conjuntoCursos, curso) ▷ std::set::insert,  $\mathcal{O}(\log n)$ 
14:    i  $\leftarrow$  i + 2
15:   fin mientras ▷ final  $\mathcal{O}(n \log n)$ 
16: fin procedimiento ▷ final  $\mathcal{O}(n \log n)$ 

17: función SELECCIÓN DE CURSOS ▷  $\mathcal{O}(1)$ 
18:   listaNumeros  $\leftarrow$  NUEVA LISTA
19:   curso  $\leftarrow$  PRÓXIMO CURSO(conjuntoCursos) ▷  $\mathcal{O}(1)$ 
20:   num  $\leftarrow$  NUMERO(curso)
21:   fin  $\leftarrow$  FECHA DE FIN(curso)
22:   mientras proxCurso  $\leftarrow$  PRÓXIMO CURSO(conjuntoCursos) hacer ▷  $\mathcal{O}(1)$ 
23:     proxNum  $\leftarrow$  NUMERO(proxCurso) ▷  $\mathcal{O}(1)$ 
24:     proxInicio  $\leftarrow$  FECHA DE INICIO(proxCurso) ▷  $\mathcal{O}(1)$ 
25:     proxFin  $\leftarrow$  FECHA DE FIN(proxCurso) ▷  $\mathcal{O}(1)$ 
26:     si fin < proxInicio  $\wedge$  fin < proxFin entonces ▷  $\mathcal{O}(1)$ 
27:       INSERTAR(listaNumeros, num) ▷  $\mathcal{O}(1)$ 
28:       num  $\leftarrow$  proxNum
29:       fin  $\leftarrow$  proxFin
30:     fin si
31:   fin mientras ▷ ciclo  $\mathcal{O}(n - 1)$ 
32:   INSERTAR(listaNumeros, num) ▷ el último,  $\mathcal{O}(1)$ 
33:   retornar listaNumeros ▷ final  $\mathcal{O}(n)$ 
34: fin función ▷ final  $\mathcal{O}(n)$ 

```

2.2.3. Justificación formal de correctitud

Notación. $curso = \langle fechaInicio, fechaFin, ID \rangle$, tupla tal que $fechaInicio < fechaFin$ y ID es único.

Definición 2.2.3.1. Considerando a C como *el conjunto de todos los cursos provenientes de la entrada del problema*, notaremos como $F_p = \rho(C)$ al conjunto de todas las posibles soluciones, y a $F_v \subset F_p$ el subconjunto tal que

$$F_v = \{f : f \in F_p \wedge f \text{ representa a una solución válida}\}.$$

Definición 2.2.3.2. $\forall f \in F_v$, caracterizamos a $s(f)$ secuencia conformada por los cursos $c \in f$ de forma tal que se cumple que $\forall s_i(f), s_{i+1}(f)$

$$fechaInicio(s_i(f)) < fechaFinal(s_i(f)) < fechaInicio(s_{i+1}(f)) < fechaFinal(s_{i+1}(f))$$

Denotamos como $S(f)$ al conjunto de todas las posibles secuencias de la forma $s(f)$.

Nota. Como la consigna requiere que no existan cursos solapados, asumimos que para todo conjunto de cursos que pueda representar una **solución válida** existirá una y solo una secuencia ordenada según el criterio mencionado anteriormente, de forma tal que para cada curso que se encuentre en la posición $i + 1$ de la secuencia, la fecha de inicio es mayor a la fecha de fin del **curso** que se encuentre en la posición anterior (i). Demostrarlo resulta trivial ya que, dado que todos los **cursos** tienen su propia fecha de finalización posterior a su propia fecha de inicio, si asumiesemos que el criterio de orden estricto utilizado en la definición anterior no fuese satisfacible para alguna **solución válida**, entonces mediante dicha solución podríamos construir una secuencia en donde (al resultar imposible cumplir con el criterio de orden) debería existir al menos un par de cursos para el cual la fecha de inicio del primero de ellos sería menor a la fecha de finalización del otro, y en donde la fecha de inicio de este último sería menor a la fecha de finalización del primero, lo cual es absurdo porque los cursos no se pueden solapar.

Definición 2.2.3.3. Notamos $s(x) \in S(f)$ como la secuencia formada por los **cursos** de nuestra solución.

Proposición 2.2.3.1. $\forall i \in \mathbb{N}, s(f) \in S(f), \nexists s(f)[1 \dots i]$ subsecuencia de $s(f)$ formada por sus primeros i elementos tal que

$$\nexists s(x)[1 \dots i] \vee fechaFinal(s_i(x)) > fechaFinal(s_i(f))$$

Demostración 2.2.3.2.

Caso base Suponemos que existe un $s_1(f)$ subsecuencia de $s(f)$ tal que

$$fechaFinal(s_1(f)) < fechaFinal(s_1(x))$$

Sabemos que el criterio de selección de nuestra solución toma siempre desde el conjunto de cursos que todavía no han sido elegidos o descartados el curso con menor fecha de finalización. Entonces, llegamos a un absurdo, puesto que como el primer curso que está tomando nuestra solución va a ser necesariamente alguno de los que menor fecha tenga con respecto a todo el conjunto de cursos, es contradictorio plantear la existencia de otra solución cuyo primer curso tenga una fecha menor a esta.

Suponemos entonces que existe un $s_1(f)$ subsecuencia de $s(f)$ tal que $\nexists s(x)[1 \dots 1]$. Esto también es absurdo, porque si sabemos que existe al menos «**un curso**» (de forma tal que permite la existencia de la subsecuencia $s_1(f)$) en particular existe también «**un curso de menor fecha de finalización**», y por lo tanto existe $s(x)[1 \dots 1]$.

Paso inductivo $P(n) \Rightarrow P(n + 1)$

Suponemos que existe una subsecuencia $z = s(f)[1 \dots n + 1]$ de $s(f)$, de forma tal que la misma contiene $n + 1$ cursos de una **solucion válida**, y que además es **potencialmente mejor que nuestra solución**, es decir que se cumple que $\text{fechaFinal}(z_{n+1}) < \text{fechaFinal}(s_{n+1}(x))$, o bien que $\nexists s(x)[1..n + 1]$.

En particular, podemos afirmar que al ser z una subsecuencia de $s(f)$, la subsecuencia $g = z - z_{n+1}$ (puesto que es una subsecuencia de z) va a seguir siendo también una subsecuencia de $s(f)$, y debido a que para formarla le estamos quitando el último elemento a $z = s(f)[1 \dots n + 1]$, va a tener la forma $g = s(f)[1 \dots n + 1] - s_{n+1}(f) = s(f)[1 \dots n]$.

Por la **hipótesis inductiva**, sabemos que vale $\exists s(x)[1 \dots n]$, y que además no existe subsecuencia $s(f)[1 \dots n]$ de $s(f)$ tal que se cumpla $\text{fechaFinal}(s_n(f)) < \text{fechaFinal}(s_n(x))$. Esto es equivalente a afirmar que

$$(\forall s(f)[1 \dots n]) \text{fechaFinal}(s_n(x)) \leq \text{fechaFinal}(s_n(f))$$

(Esta última afirmación aplica también a g , ya que es subsecuencia de $s(f)$)

Finalmente, podemos concluir: por un lado que no es posible que se cumpla que dado $z \Rightarrow \nexists s^x[1 \dots n + 1]$, puesto que sabemos que $\exists s^x[1 \dots n]$, que $\exists g = z - z_{n+1}$, y que $\text{fechaFinal}(s_n(x)) \leq \text{fechaFinal}(g)$, nuestro algoritmo podría haber tomado el curso z_{n+1} , el cual no se solapa con g , y por consiguiente no se solapa con $s_n(x)$. Por otro lado, tampoco es posible que se cumpla que dado $z \Rightarrow \exists s(x)[1 \dots n + 1] \wedge \text{fechaFinal}(z_{n+1}) < \text{fechaFinal}(s_{n+1}(x))$, puesto que sabemos que $\exists s(x)[1..n]$, que $z = s_n(f) + z_{n+1}$, y que $\forall s(f)[1..n], \text{fechaFinal}(s_n(x)) \leq \text{fechaFinal}(s_n(f))$, y como **nuestra solución** toma siempre el de menor fecha de finalización, es absurdo plantear que z_{n+1} pudiese terminar antes que $s_{n+1}(x)$, puesto que en ese caso, como la fecha de finalización de $s_n(x)$ es menor a la de z_n , debería haberlo elegido. \square

Proposición 2.2.3.3. $s(x) \in S(f)$ (**nuestra solución**) es óptima.

Demostracion 2.2.3.4. Suponemos que existe una solución mejor a $s(x)$. En particular, esa solución va a ser válida, por lo que va a pertenecer a $S(f)$. Para ser mejor que nuestra solución, esta debe como condición necesaria pero no suficiente, cumplir las mismas condiciones expuestas en la proposición anterior, es decir, volcándola en una secuencia bajo el mismo criterio de ordenamiento ya expuesto, debería existir un i para el cual, o bien esta nueva solución tenga en su última posición una fecha de finalización menor a nuestra solución, o bien nuestra solución disponga tan solo de $i - 1$ elementos en su secuencia. Aplicando el mismo tipo de razonamiento que para el procedimiento anterior, se llega al absurdo, lo cual demuestra que no puede existir una solución mejor que la planteada.

2.2.4. Cota de complejidad temporal

El algoritmo ejecuta dos pasos. Primero recorre todos los cursos y los inserta con orden en el conjunto ordenado. La recorrida toma un tiempo linea $\mathcal{O}(n)$ dado que se recorren todos los elementos en orden de entrada, mientras que la inserción se hace en tiempo logarítmico al tratarse de una estructura implementada en base a *heapsort*. Este proceso tiene una complejidad de $\mathcal{O}(n \log n)$.

En segunda instancia se hace la selección de cursos, en donde se recorren nuevamente todos cursos tomando una complejidad de $\mathcal{O}(n)$. La complejidad final es de $\mathcal{O}(n \log n)$.

2.2.5. Verificación mediante casos de prueba

Para verificar el correcto funcionamiento de nuestro programa tomamos una serie de instancias que consideramos representativas de casos extremos o sensibles y analizamos el comportamiento. Definimos los siguientes casos:

in1: 2 1 1 1 1	out1: 1
in2: 5 1 2 2 3 3 4 4 5 5 6	out2: 1 3 5
in3: 4 1 2 3 4 5 6 7 8	out3: 1 2 3 4
in4: 4 1 10 2 3 4 5 6 7	out4: 2 3 4
in5: 6 1 6 4 5 2 3 7 9 9 10 10 12	out5: 3 2 4 6

La entrada 1 provee dos cursos con mismas fechas, con lo cual devuelve el primero (en este caso el desempate llega hasta menor «id»). En la entrada 2, cada curso se solapa con el siguiente, con lo cual toma uno sí y otro no. En la entrada 3 son todos cursos contiguos, entonces se seleccionan todos. La entrada 4 contiene cursos contiguos salvo el primero que se solapa con todos, entonces, a éste no lo tiene en cuenta y elige todos los demás. La entrada 6 es aquella que se usó como ejemplo en la descripción del problema. Se puede ver que la elección es consecuente con lo mostrado.

2.2.6. Medición empírica de la performance

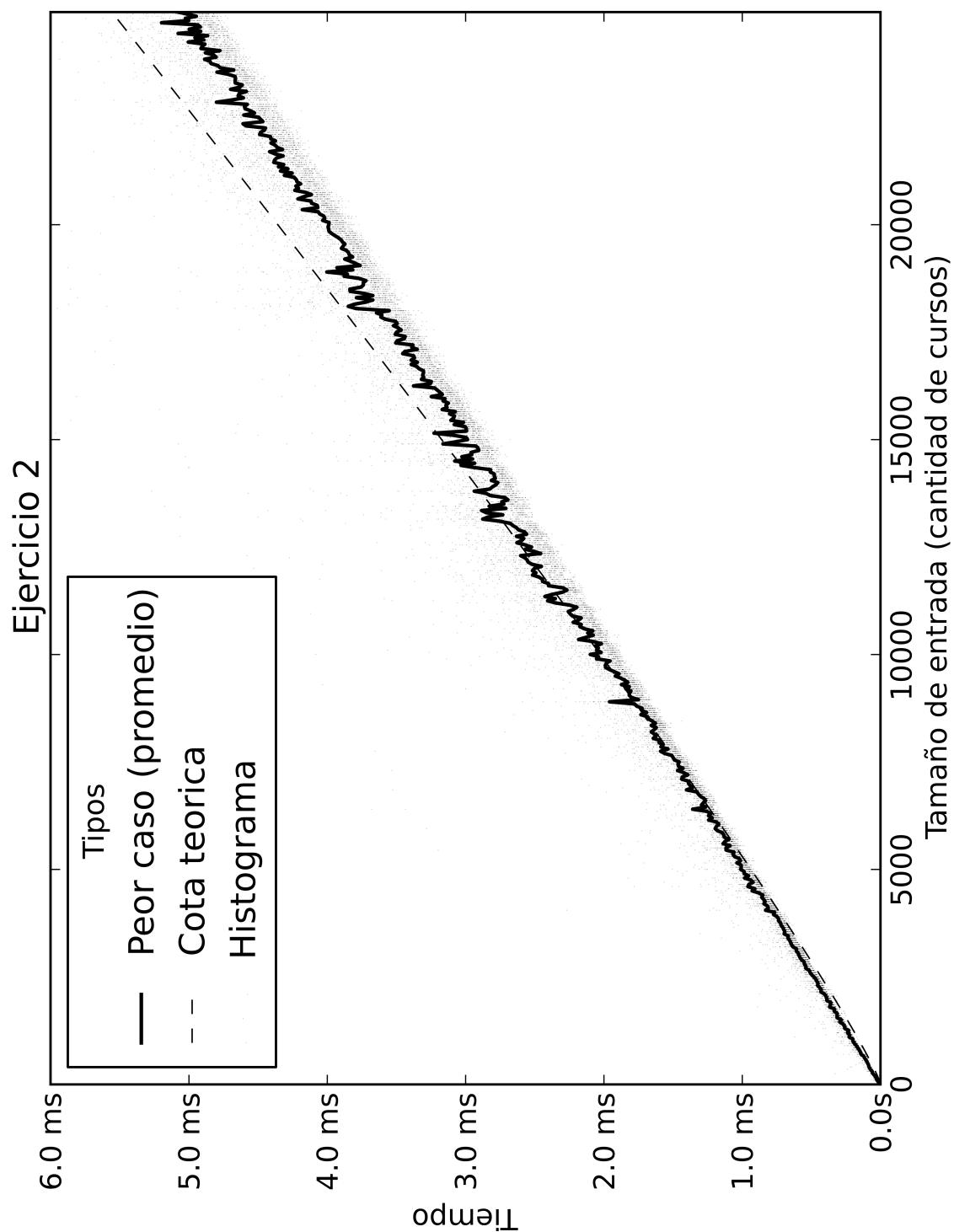


Figura 3: Primer muestreo del Ejercicio 2

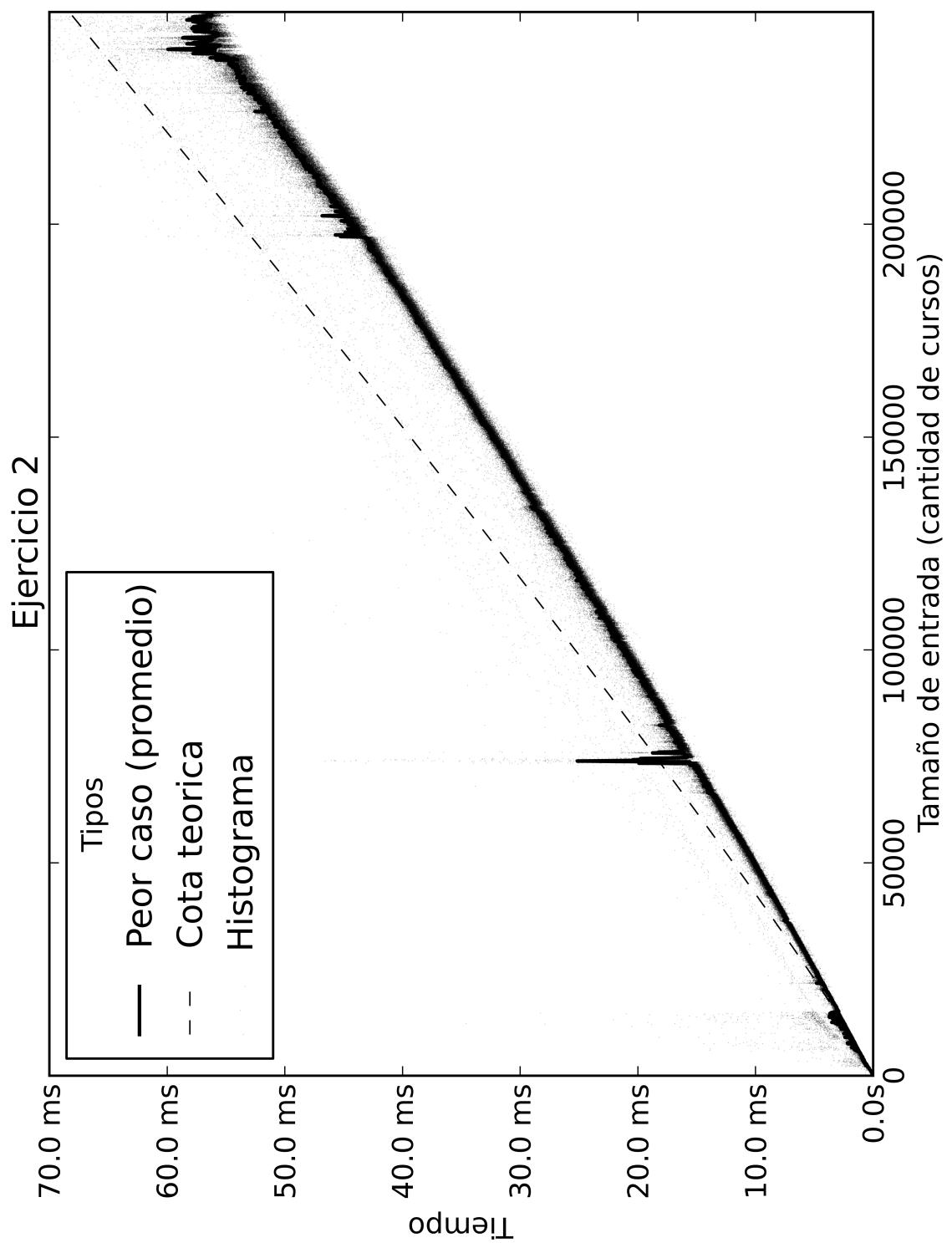


Figura 4: Segundo muestreo del Ejercicio 2

2.3. Problema 3: Una noche en el museo

2.3.1. Descripción

La optimización es una constante. Los escenarios anteriores buscaban optimizar el tiempo de respuesta del algoritmo que solucione el problema. En este caso, además de mantener la complejidad bajo cotas aceptables, es necesario optimizar la solución del problema. Queremos que nuestro algoritmo lo resuelva en un tiempo decente, pero la prioridad viene dada porque la resolución sea eficiente en cuanto a las demandas solicitadas. Hay que cuidar un museo, cuyo objetos son de gran valor. En estos casos podemos aceptar tardar un poco más siempre que aseguremos que la solución es efectivamente la que minimiza el problema.

Cada piso del museo se puede modelar como una matriz de $n \times m$, dónde cada elemento representa una porción equitativa y además contiene una propiedad. Llamaremos *grilla* a la matriz que modela el piso y *casilla* a cada elemento de la grilla. A priori, una celda puede estar libre, puede tener una pared o puede tener una *reliquia*. Nuestro trabajo es cuidar las reliquias. Para ello vamos a instalar sensores láser de modo tal que todas las celdas que no son una pared tengan protección. Contamos con dos láseres, bi-direccionales y cuatri-direccionales. Los «bi» sólo cubren en forma vertical u horizontal y los «cuatri» en ambos sentidos. La celda libre en donde se instala alguno de los láser queda automáticamente protegida. Además, el rayo emitido desde el láser protege toda su fila y/o columna hasta que se encuentra con una pared. Toda celda libre que no tenga sensor láser debe tener al menos un rayo láser. Las reliquias deben ser protegidas con exactamente dos rayos láser, siempre y cuando éstos provengan uno desde una «fila» y el otro desde una «columna». No se pueden instalar sensores en la misma fila o columna salvo que una pared se interponga entre ellos. Toda celda debe quedar cubierta o bien por una pared (de las existentes, no se pueden agregar) o bien por detección láser (sensor, rayo, doble rayo).

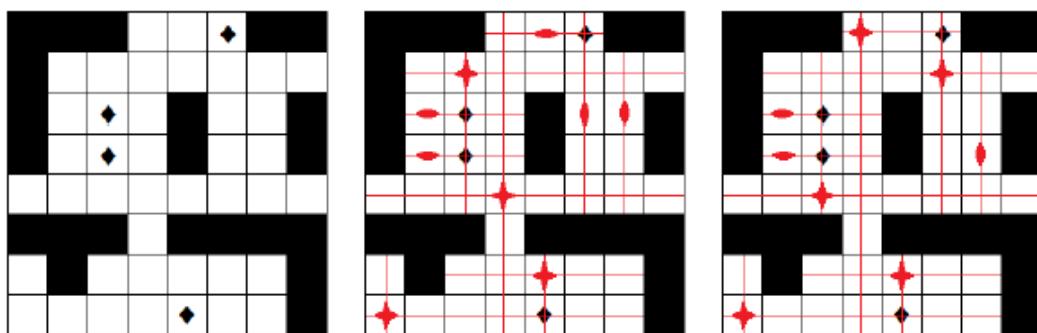


Figura 5: Una instancia del Problema 3 y dos soluciones distintas

El algoritmo debe realizar una ejecución con la técnica de *backtracking* devolviendo sólo la solución óptima (o alguna de ellas) en caso de existir. O sea, en cada momento se preserva sólo la solución óptima parcial y se descartan las demás. Esto tiene como provechoso el hecho de que al necesitar gastar la menor cantidad de dinero, si en medio de una rama de posible solución ya superamos en costo a la posible solución óptima parcial guardada, entonces podemos *podar* esta rama desde la bifurcación próxima anterior sin necesidad de completarla. De todas formas no se puede garantizar que se encuentre una solución. Pero si existe, vamos a demotrar que es óptima.

2.3.2. Hipótesis de resolución

Para que podamos asegurar el museo, se reciben las dimensiones del piso (n, m) y su matriz asociada, donde cada celda (o elemento) representa $1m^2$ del piso. Al momento de recibir la grilla, cada celda puede tener sólo uno de los siguientes estados: PARED, LIBRE, RELIQUIA. Para diferenciarlos, la cátedra asigna un valor a cada estado inicial, siendo 0, 1, 2 respectivamente.

Toda celda con estado LIBRE o con una reliquia debe quedar protegida, ya sea colocando un sensor sobre la misma o que el haz de láser emitido por un sensor cruce la celda. Una solución al problema va tener uno y solo uno de los siguientes estados: SENSORH, SENSORV, SENSORC, LASERH, LASERV y LASERC. En la siguiente tabla se describen cada uno de los estados:

Estado celda	Descripción estado
SENSORH	En la celda hay un sensor bi-direccional que emite un haz de láser a lo largo de la fila.
SENSORV	En la celda hay un sensor bi-direccional que emite un haz de láser a lo largo de la columna.
SENSORC	En la celda hay un sensor cuatri-direccional que emite haz de láser a lo largo de la fila y otro a lo largo de la columna de la celda.
LASERH	La celda esta protegida por un haz de láser horizontal emitido desde otra celda de la fila.
LASERV	La celda esta protegida por un haz de láser horizontal emitido desde otra celda de la columna.
LASERC	La celda esta protegida por dos haces de láser, uno horizontal emitido por un sensor en la fila y otro vertical emitido por un sensor en la columna.

El enunciado pide que las celdas que contengan reliquias estén protegidas por 2 haces de láser. Luego, toda solución válida va tener el estado LASERC en las celdas que inicialmente tenían el estado RELIQUIA. Ésta es la primer cota que podemos aplicar a fin de reducir el procesamiento.

Dada una fila o columna, las celdas que las componen no pueden tener una combinación arbitraria de los estados mencionados. Por ejemplo, una fila no puede tener dos celdas con el estado SENSORH. Este tipo de restricción impuesta por el enunciado nos permite aplicar podas, dado que sabemos a priori que ciertas combinaciones de estados son imposibles.

En la siguiente tabla se muestran que combinaciones de estados son posibles en las filas y columnas de la grilla.

Los estados válidos para una celda van a estar dados por la intersección entre los estados posibles de la fila y la columna a la que cada celda pertenezca. La información contenida en la tabla no es suficiente para podar correctamente. Por ejemplo, si nos basamos sólo en esta tabla podríamos tomar como solución una grilla donde una columna en la cual todas sus celdas se encuentren en el estado LASERV. Esta solución es incorrecta dado que no hay sensor que emita el haz de láser, contradiciendo lo descripto en la tabla de estados. Esto nos permite aplicar otra poda. Al saber de antemano que toda fila en la cual alguna de sus celdas tiene el estado LASERC o LASERH, se requiere que alguna celda dentro de la fila se encuentre en estado SENSORC o LASERH. Análogamente para las columnas, cuando alguna de sus celdas tiene el estado LASERC o LASERV se requiere que una celda dentro de la columna tenga el estado SENSORC o LASERV.

Estado celda	Fila puede contener celdas con estado	Columna puede contener celdas con estado
SENSORH	LASERC, LASERH	LASERH, SENSORH
SENSORV	LASERV, SENSORV	LASERV, LASERC
SENSORC	LASERC, LASERH	LASERC, LASERV
LASERH	LASERC, LASERC, SENSORC, LASERH, SENSORH	LASERH, SENSORH
LASERC	LASERC, LASERC, SENSORC, LASERH, SENSORH	LASERC, SENSORC, LASERV, SENSORV
LASERV	LASERV, SENSORV	LASERC, SENSORC, LASERV, SENSORV

Cuadro 1: Combinaciones posibles de estados

Por el momento, en todo lo analizado previamente no se tuvo en cuenta las celdas con estado inicial PARED.

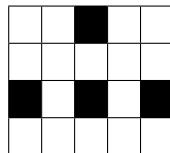


Figura 6: Dibujo de grilla con paredes

Por ejemplo, en grilla de la *Figura 6* se puede ver que la primer fila se encuentra dividida en 2 por una pared, lo cual nos da 2 sub-filas totalmente independientes entre sí. Con esto queremos decir que la selección de estado de las celdas de la sub-fila 1 no implica una reducción en los posibles estados que puedan tomar las celdas de la sub-fila 2. En este caso las podas pensadas anteriormente pierden cierta efectividad.

Luego, en la tercer fila se puede ver la misma situación con la particularidad que la longitud de una de las sub-filas es de 1 celda. Es evidente que dicha celda no va poder tomar ningún estado que implique que hay un haz de láser horizontal sobre la celda, o sea los estados LASERC y LASERV, dado que no hay celda en dicha sub-fila que emita el haz. Detectar sub-filas y sub-columnas de longitud 1 nos permitirá probar en dichas celdas solo los casos posibles y correctos.

Es posible que algunas grillas no admitan solución. Por ejemplo una grilla que contenga toda un fila/columna (o sub-fila/columna) donde todas las celdas que la componen tengan como estado inicial RELIQUIA. Una situación parecida se presenta a cuando una sub-fila/columna de 1 celda contiene una celda con estado inicial RELIQUIA. Es importante poder detectar este tipo de situación lo antes posible y evitar malgastar tiempo de cómputo.

La primer táctica de solución que se consideró para resolver el problema consistía en modificar el estado de las celdas con estado inicial LIBRE por los diferentes estados de sensor. Luego modificar las celdas de fila y/o columna sobre las cuales proyectara el sensor en cuestión, con el estado de haz emitido por él mismo. Luego escanear la grilla a fin de validar que cumple que dicha solución parcial cumpliera con los requerimientos. De cumplir con los requerimientos se procedería a completar la próxima celda con estado LIBRE, nuevamente con los diferentes sensores y se repetiría el mismo procedimiento hasta que no haya celdas con estado inicial LIBRE. De no cumplir con los requerimientos se volvería al estado anterior, para luego probar

con otro tipo de sensor. Esta táctica, a pesar de ser correcta, requería escanear la grilla cada vez que se probaba un nuevo estado en una celda, lo cual resultaba computacionalmente costoso, por lo cual decidimos buscar otra alternativa de solución.

La segunda táctica que se consideró, que fue la que finalmente utilizamos, consiste en tener en todo momento conocimiento de los estados posibles que pudiesen tener celdas de la grilla. La grilla se completa en orden, de a 1 celda LIBRE a la vez, modificando los estados disponibles para la fila/columna de la celda según las especificaciones de la tabla 1. A diferencia de la táctica anterior, no solo completamos las celdas con sensores, también es posible completar una celda con alguno de los estados de láser. De llegar a procesar todas las celdas con estado LIBRE sabremos que dicha solución cumple con los requerimientos del problema, dado que se completó aplicando las restricciones solicitadas por el enunciado.

En el caso que la solución parcial no llegue a ser una solución, eventualmente se llegará a procesar alguna celda para la cual no hay estados posibles, con lo cual se deberá retroceder a las celdas previas y continuar probando con combinaciones de estados diferentes. Esta táctica nos permite evitar escanear la grilla cada vez que se modifica el estado de la grilla, aunque requiere mayor consumo de memoria.

Las opciones de las celdas se calcularán haciendo una intersección entre los estados disponibles de la fila y columna de cada celda. Se guardan y se mantendrán actualizados los estados posibles de cada fila/columna de la grilla, los mismos estarán codificados en una máscara según se muestra en la *Figura 7*.

Req	LV	LH	LC	SV	SH	SC	0
-----	----	----	----	----	----	----	---

Figura 7: Mascara de estados disponibles para columna/fila

El primer bit comenzando desde la derecha no se utiliza y siempre se mantiene en 0. Los 6 campos siguientes, de tamaño de 1 bit, toman el valor 1 cuando el estado está permitido para la celda/columna y 0 cuando no.

SC	Estado SENSORC
SH	Estado SENSORH
SV	Estado SENSORV
LC	Estado LASERC
LH	Estado LASERH
LV	Estado LASERV
Req	Estado Requerimiento Fila/Columna (2 bits)
	Puede tomar los siguientes valores
	00 - No Requiere Sensor
	01 - Requiere Sensor
	02 - Tiene Sensor

Como se mencionó antes, una celda con estado *Pared* genera múltiples sub-filas/columnas independientes, por lo que será necesario guardar el cambio de estado que implica cada pared.

Primero se analizará la grilla, generando las opciones iniciales para cada fila/columna. De detectar que el problema no tiene solución se terminará la ejecución inmediatamente. Luego, dentro del espectro de soluciones posibles resultantes de este análisis probaremos todas las combinaciones de estados, quedándonos con la solución de menor costo. A fin de mejorar el tiempo de ejecución se agrega una poda más, que consiste en dejar de procesar toda solución parcial que sea mayor o igual a la mejor solución que se haya encontrado hasta el momento. En el *Algoritmo 3* se puede observar lo descripto.

Algoritmo 3 Asegurar Museo

Entrada:

$n, m \leftarrow \text{INT}$ \triangleright dimensiones de la matriz
 $G \leftarrow \text{MATRIZ DE } n \times m$ \triangleright grilla de celdas

Salida:

CANTIDAD DE SENSORES \triangleright si es -1 , no existe solución
COSTO DE LA SOLUCIÓN
LISTA DE SENSORES \triangleright sensor = (tipo, fila, columna)

```

1: opcionesFilas  $\leftarrow$  NUEVO VECTOR( $n$ )  $\triangleright$  posibilidades en cada celda
2: opcionesCol  $\leftarrow$  NUEVO VECTOR( $m$ )
3: opciones  $\leftarrow$  NUEVO PAR(opcionesFilas, opcionesCol)
4: celdasProcesar  $\leftarrow$  NUEVA LISTA
5: solucion  $\leftarrow$  NUEVA LISTA DE SENSORES
6: solucionOptima  $\leftarrow$  NUEVA LISTA DE SENSORES

7: CARGAR( $n, m, G$ )
8: si ANALIZAR( $G, opciones, celdasProcesar$ ) = TIENE SOLUCIÓN entonces  $\triangleright \mathcal{O}(n)$ 
9:     BACKTRACK( $G, opciones, celdasProcesar, solucion, solucionOptima$ )  $\triangleright \mathcal{O}(n * 6^n)$ 
10:    retornar solucion
11: fin si

12: función ANALIZAR( $G, opciones, celdasProcesar$ )
13:     FilaValida  $\leftarrow$  NUEVO VECTOR( $n$ )  $\triangleright$  Máscaras que verifican validez de filas
14:     ColValida  $\leftarrow$  NUEVO VECTOR( $m$ )  $\triangleright$  Máscaras que verificar validez de columnas
15:     opcionesFilas  $\leftarrow$  PRIMERO(opciones)
16:     opcionesCol  $\leftarrow$  SEGUNDO(opciones)
17:     para  $i$  desde 1 hasta  $n$  hacer
18:         para  $j$  desde 1 hasta  $m$  hacer
19:             OR BINARIO(FilaValida[ $i$ ],  $G[i][j]$ )
20:             OR BINARIO(ColValida[ $j$ ],  $G[i][j]$ )
21:             mascara  $\leftarrow$  ANALIZARCELDA( $G, i, j, celdasProcesar$ )
22:             si mascara es CONFIGURACIÓN INVALIDA entonces
23:                 retornar No TIENE SOLUCIÓN
24:             fin si
25:             ACTUALIZAR OPCIONES FILA(opcionesFila,  $i$ , mascara)
26:                          $\triangleright$  Actualiza opciones Fila o asigna opciones a Pared
27:             ACTUALIZAR OPCIONES COLUMNA(opcionesCol,  $j$ , mascara)
28:                          $\triangleright$  Actualiza opciones Columna o asigna opciones a Pared
29:         fin para
30:     fin para
31:     para  $i$  desde 1 hasta  $n$  hacer
32:         si FilaValida[ $i$ ] es FILA INVALIDA entonces
33:             retornar No TIENE SOLUCIÓN
34:         fin si
35:     fin para
36:     para  $j$  desde 1 hasta  $m$  hacer
37:         si ColValida[ $j$ ] es COL INVALIDA entonces
38:             retornar No TIENE SOLUCIÓN
39:         fin si
40:     fin para
41:     retornar TIENE SOLUCIÓN
42: fin función

```

Algoritmo 4 Asegurar Museo (continuación)

```

41: procedimiento BACKTRACK( $G$ ,  $opciones$ ,  $celdasProcesar$ ,  $solucion$ ,  $solucionOptima$ )
42:    $snapshot \leftarrow$  NUEVA TUPLA( $Vector$ ,  $Vector$ ,  $Opcion$ )
43:    $opcionesFilas \leftarrow$  PRIMERO( $opciones$ )
44:    $opcionesCol \leftarrow$  SEGUNDO( $opciones$ )
45:   si  $celdasProcesar$  es VACIO entonces
46:      $celda \leftarrow$  OBTENER PRIMER ELEMENTO( $celdasProcesar$ )
47:      $mascaraFila \leftarrow$  OBTENER MASCARA FILA( $celda$ ,  $opcionesFilas$ )
48:      $mascaraColumna \leftarrow$  OBTENER MASCARA COLUMNA( $celda$ ,  $opcionesCol$ )
49:      $mascaraRequerimiento \leftarrow$  REQUIERE SENSOR( $mascaraFila$ ,  $mascaraColumna$ ,  $celda$ )
50:      $mascaraCelda \leftarrow$  AND BINARIO( $mascaraFila$ ,  $mascaraColumna$ ,  $mascaraRequerimiento$ )
51:      $snapshot \leftarrow$  GUARDAR ESTADO( $opcionesFilas$ ,  $opcionesCol$ ,  $celda$ )
52:     para cada  $opcion$  en OPCIONES hacer
53:        $opValida \leftarrow$  ES VALIDA( $mascaraCelda$ ,  $opcion$ )
54:        $precioOp \leftarrow$  PRECIO( $opcion$ )
55:        $precioSol \leftarrow$  PRECIO( $solucion$ )
56:        $precioSolOptima \leftarrow$  PRECIO( $solucionOptima$ )
57:       si  $opValida \wedge (precioSol + precioOp < precioSolOptima)$  entonces
58:         ACTUALIZAR ESTADO( $opcion$ ,  $opcionesFilas$ ,  $opcionesCol$ )
59:         BACKTRACK( $G$ ,  $opciones$ ,  $celdasProcesar$ ,  $solucion$ ,  $solucionOptima$ )
60:         ACTUALIZAR ESTADO( $snapshot$ ,  $opcion$ ,  $opcionesFilas$ ,  $opcionesCol$ )
61:       fin si
62:     fin para
63:   sino
64:     si PRECIO( $solucion$ ) < PRECIO( $solucionOptima$ ) entonces
65:        $solucionOptima \leftarrow solucion$  ▷  $\mathcal{O}(n)$ 
66:     fin si
67:   fin si
68: fin procedimiento

69: función ANALIZAR CELDA( $G$ ,  $i$ ,  $j$ ,  $celdasProcesar$ )
70:    $mascara \leftarrow$  CONFIGURACIÓN INVÁLIDA
71:   si  $celdasProcesar$  es LIBRE entonces
72:     AGREGAR AL FINAL DE LISTA( $libre$ )
73:      $mascara \leftarrow$  MASCARA NO HAY CAMBIOS
74:   sino si  $celdasProcesar$  es RELIQUIA entonces
75:      $G[i][j] \leftarrow LASERC$ 
76:     si ESTA RODEADA POR PAREDES( $G$ ,  $i$ ,  $j$ ) es FALSO entonces
77:        $mascara \leftarrow$  MÁSCARA LASER C
78:     fin si
79:   sino si  $celdasProcesar$  es PARED entonces
80:      $mascaraFila \leftarrow$  GENERAR MÁSCARA SEGÚN LONGITUD SUB FILA( $G$ ,  $j$ )
81:      $mascaraCol \leftarrow$  GENERAR MÁSCARA SEGÚN LONGITUD SUB COL( $G$ ,  $j$ )
82:      $mascara \leftarrow$  OR BINARIO( $mascaraFila$ ,  $mascaraCol$ )
83:   fin si
84:   retornar  $mascara$ 
85: fin función

```

2.3.3. Justificación formal de correctitud

Como se describió en las secciones anteriores, en todo momento se tiene total conocimiento de todos los estados posibles (y correctos) que puede tener cada celda. Para ello se escanea la grilla a fin de recopilar las condiciones iniciales de la grilla. Al escanear la grilla también se verifica que la misma tenga solución, en el caso de detectar que no tiene solución se interrumpe el algoritmo. Cada vez que se «completa» una celda, se actualizan las opciones disponibles para la fila y/o columna de la celda, asegurando que próximas llamadas solo se utilizarán estados correctos. Esto, sumado se a que se modifica el estado de las celdas de manera ordenada, nos permite asegurar que de llegar a una solución, es decir, llegar a un estado donde no queden celdas con estado *Libre*, la solución es válida. Antes de completar una celda con alguna de las opciones disponibles se guarda el estado, esto permite que al retornar el procedimiento BACKTRACK se restaure el estado previo para poder probar con el resto de las opciones. Toda nueva solución encontrada se compara contra la mejor solución encontrada hasta el momento. De superarla, se guarda la nueva solución como óptima. Esto nos asegura que al terminar el algoritmo, en caso de encontrar una solución, será óptima.

2.3.4. Cota de complejidad temporal

El *Algoritmo 3* consiste, en el caso de que la grilla tenga solución, en la ejecución secuencial de los procedimientos ANALIZAR GRILLA y BACTRACK.

Ambos procedimientos utilizan estructuras provistas por la biblioteca estándar de C++. Se utilizan vectores, `std::vector`, para guardar las máscaras de cada fila/columna, y para la grilla se utiliza un vector de vectores. Por lo documentado en la biblioteca estándar de C++, `std::vector` nos garantiza acceso aleatorio con orden constante. Para almacenar la lista de celdas a procesar se utiliza `std::list`, sobre dicha estructura solo se realizan operaciones de inserción/borrado al inicio y final de la lista, siendo estas operaciones de orden constante según lo documentado en la biblioteca estándar de C++.

El procedimiento ANALIZAR GRILLA recorre la grilla actualizando estados de celdas y modificando opciones de la fila y/o columna de la celda analizadas. Las operaciones que se ejecutan por celda se pueden considerar constantes. Al análisis de la grilla es $\mathcal{O}(n)$. Luego se recorren las opciones resultantes de cada fila y columna para verificar la validez de la grilla, con una complejidad $\mathcal{O}(\#filas)$ y $\mathcal{O}(\#columnas)$ respectivamente. El orden del procedimiento será $\mathcal{O}(\text{MÁX}(n, \#fila, \#columna))$. Como n es mayor que $\#filas$ y $\#columnas$ para toda grilla, ANALIZAR GRILLA resulta $\mathcal{O}(n)$.

Ahora analizamos la complejidad de BACTRACK. Sea $T(i)$ el tiempo que demora el procedimiento BACTRACK con un entrada de tamaño i . Para el caso donde $i = 0$, se verifica si la solución encontrada y guardada en el vector *solucion* tiene un costo menor que a la solución guardada. De tener un menor precio, se copian los valores de la solución actual al vector *solucion_optima*. Guardar la nueva solución implica eliminar los k elementos del vector *solucion_optima* y luego hacer inserción de los k' sensores de la nueva solución. Por documentación de la biblioteca estándar de C++, sabemos que en el peor de los casos la operación de borrar k elementos es $\mathcal{O}(k)$ [5]. La inserción de un elemento al final del vector es constante, luego la inserción de k' elementos $\in \mathcal{O}(k')$. Dado que no puede haber más sensores que celdas, $k \leq n$ y $k' \leq n$, resultando $\mathcal{O}(n)$ para el caso $i = 0$. Para los casos $n \geq 1$, se harán en el peor de los casos con 6 llamadas recursivas (una por cada opción disponible) de duración $\mathcal{T}(n - 1)$ cada una. Considerando el resto de las operaciones, que consisten en comparaciones, sumas y operaciones lógicas de enteros de 32 bits, de complejidad constante, obtenemos la siguiente ecuación de recurrencia:

$$T(i) = \begin{cases} n & \text{si } i = 0 \\ 6 * T(i - 1) & \text{si } i > 1 \end{cases}$$

Calculamos $T(n)$ para algunos valores de n :

$$T(0) = n$$

$$T(1) = 6 * T(0) = n * 6$$

$$T(2) = 6 * T(1) = 6 * 6 * T(0) = n * 6^2$$

$$T(3) = 6 * T(2) = 6 * 6 * T(1) = 6 * 6 * 6 * T(0) = n * 6^3$$

Se puede conjeturar de los casos anteriores que $T(i) = n * 6^i$. Probamos la conjetura utilizando inducción en i .

Caso Base $i = 0$. Por definición $T(0) = k$. Luego, $T(0) = n * 6^0 = n * 1 = i$. Verdadero para el caso base.

Paso Inductivo HI: $T(i) = n * 6^i$. Queremos probar $T(i) \Rightarrow T(i + 1)$.

$$T(i + 1) = 6 * T(i) \stackrel{\text{HI}}{=} 6 * (n * 6^i) = n * 6^{i+1}$$

Con lo cual $T(i)$ vale $\forall i$. Entonces $T(i) \in \mathcal{O}(n * 6^i)$. Para el caso de una grilla con n celdas libres $T(n) \in \mathcal{O}(n * 6^n)$. \square

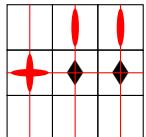
Como ANALIZAR GRILLA $\in \mathcal{O}(n)$, BACTRACK $\in \mathcal{O}(n * 6^n)$ y además $\forall n$, $n * 6^n > n$, podemos concluir que *Algoritmo 3* $\in \mathcal{O}(n * 6^n)$.

2.3.5. Verificación mediante casos de prueba

Para verificar el correcto funcionamiento del algoritmo se generaron grillas pequeñas posicionando paredes en cada una de ellas de tal manera de probar que cada una de las podas funcionasen correctamente y que no se llegue a soluciones incorrectas. A continuación se listan las pruebas mas representativas.

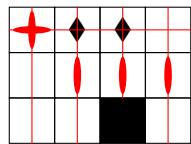
Caso de Prueba 1

Entrada	Salida
-----	-----
3 3	3 14000
1 1 1	3 1 2
1 2 2	3 1 3
1 1 1	1 2 1

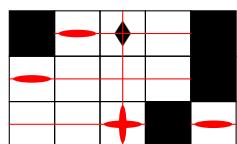


Caso de Prueba 2

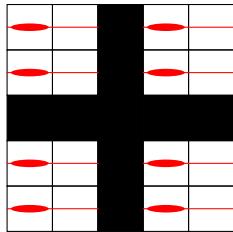
Entrada	Salida
-----	-----
3 4	4 18000
1 2 2 1	1 1 1
1 1 1 1	3 2 2
1 1 0 1	3 2 3
	3 2 4

**Caso de Prueba 3**

Entrada	Salida
-----	-----
3 5	
0 1 2 1 0	4 18000
1 1 1 1 0	2 1 2
1 1 1 1 0	2 2 1
1 1 1 0 1	1 3 3
	2 3 5

**Caso de Prueba 4**

Entrada	Salida
-----	-----
5 5	8 32000
1 1 0 1 1	2 1 1
1 1 0 1 1	2 1 4
0 0 0 0 0	2 2 1
1 1 0 1 1	2 2 4
1 1 0 1 1	2 4 1
1 1 0 1 1	2 4 4
	2 5 1
	2 5 4



Grillas sin solución

Por último se probó el algoritmo en las siguientes grillas sin solución. El algoritmo detectó la imposibilidad de resolver las grillas devolviendo -1 como es requerido por el enunciado.

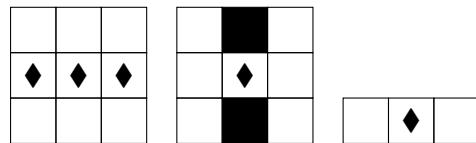


Figura 8: Ejemplos de grilla sin solución

2.3.6. Medición empírica de la performance

A fin de verificar la cota teórica empíricamente, se realizaron múltiples pruebas sobre grillas con todas sus celdas libres. En la *Figura 9* podemos observar como, en promedio, las mediciones tomadas para grillas con todas sus celdas libre está por debajo de la cota teórica. Además se realizaron mediciones sobre grillas cuadradas con diferentes porcentajes de paredes y 7 % de celdas conteniendo reliquias, las misma distribuidas al azar. La *Figura 10* muestra que los tiempos obtenidos son siempre menores a los obtenidos para grillas con todas sus celdas libres.

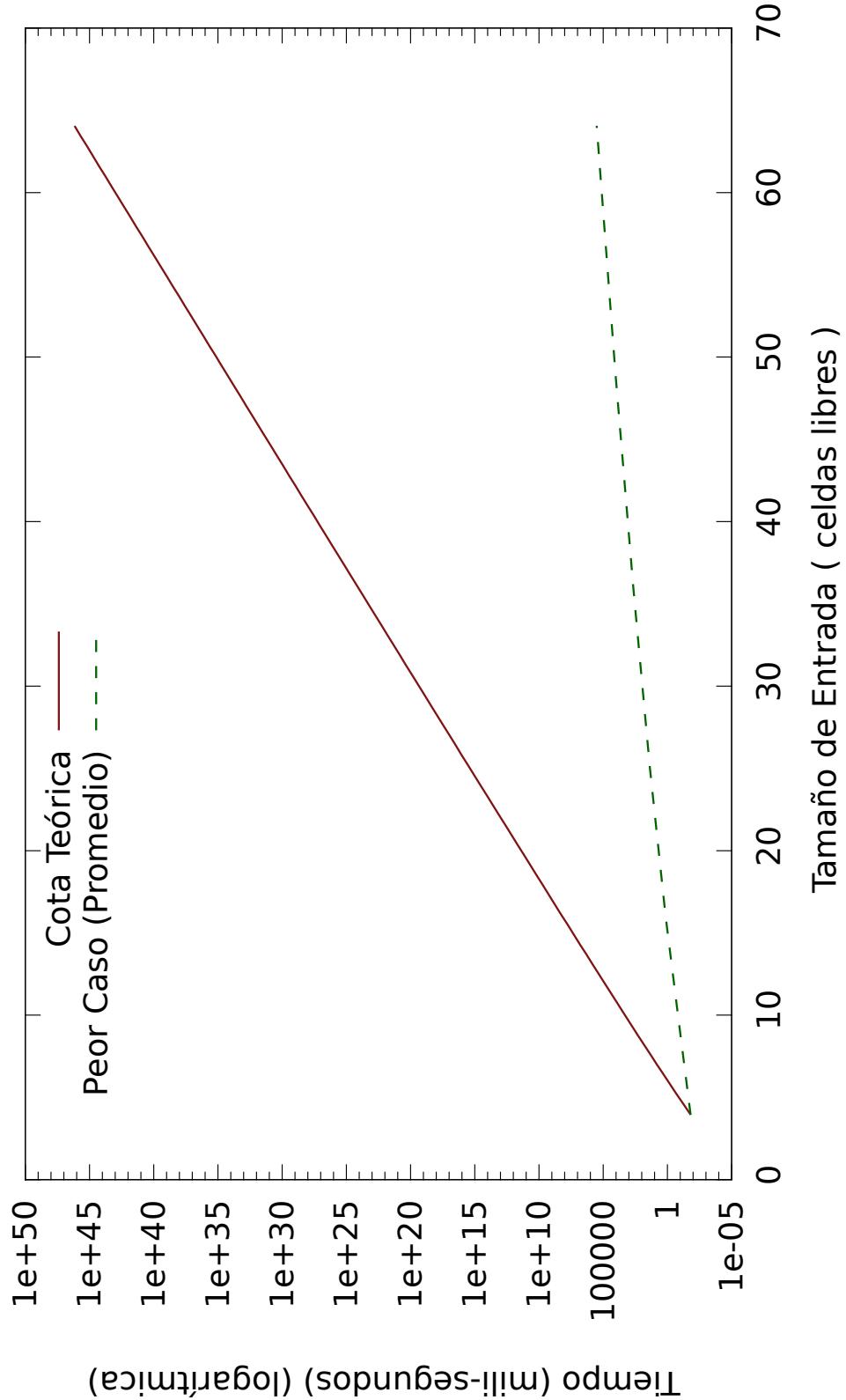


Figura 9: Muestreo del Ejercicio 3

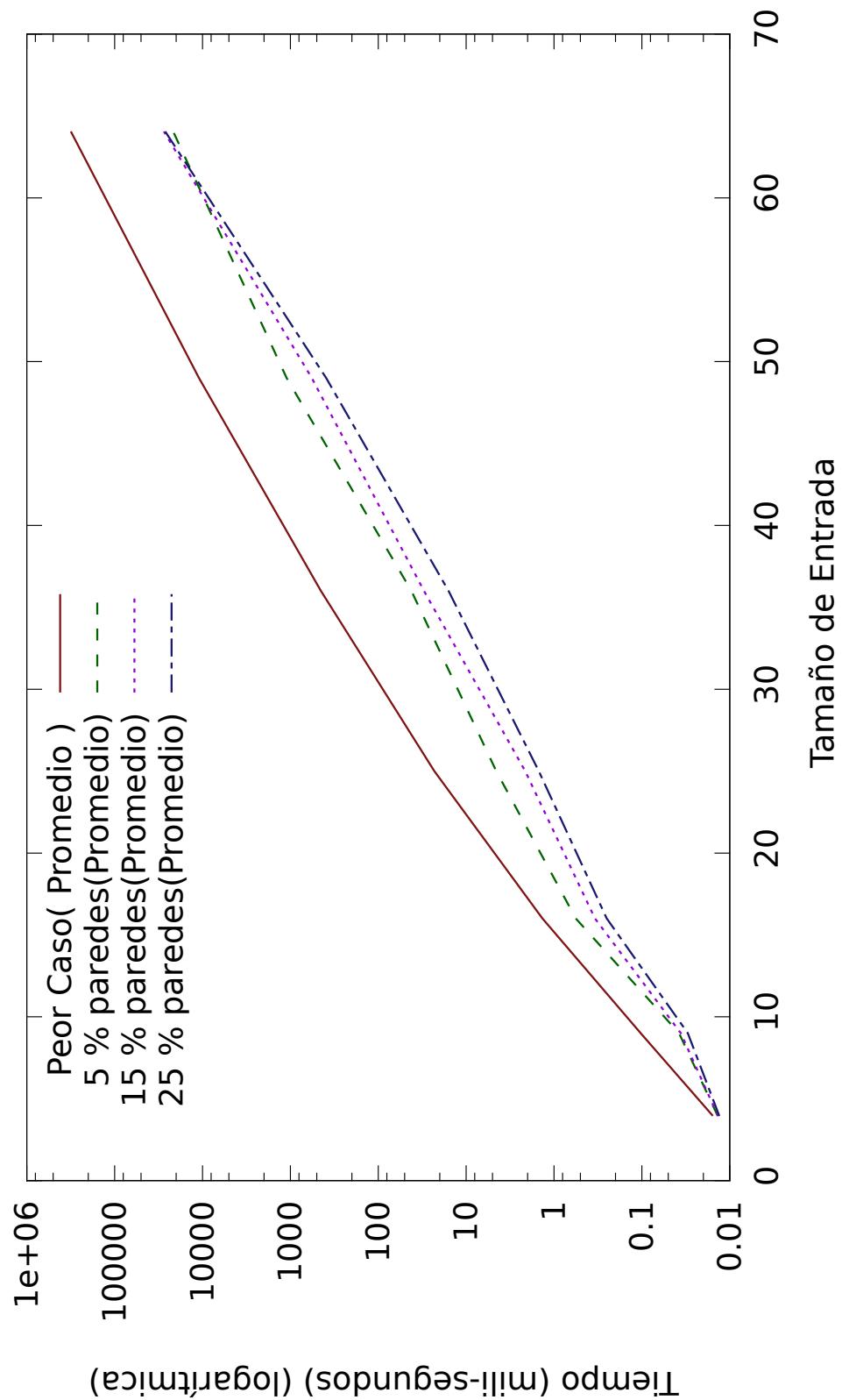


Figura 10: Muestreo del Ejercicio 3 - Peor caso contra Casos aleatorios

3. Apéndices

3.1. Código Fuente (resumen)

3.1.1. Problema 1: Pascual y el correo

Listing 1: ej1.cpp

```

1 #include "ej1.h"
2 int main( int argc, char** argv ){
3     ParserDeParametros parser( argc, argv );
4     if( ! parser.parametrosSonValidos() ) {
5         parser.imprimirAyuda();
6         return -1;
7     }
8     Ejercicio1 ejercicio = Ejercicio1();
9     while( ejercicio.obtenerInput( parser.input() ) ){
10         ejercicio.resolver();
11         ejercicio.imprimirOutput( parser.output() );
12     }
13     return 0;
14 }
```

Listing 2: Ejercicio1.cpp

```

1 void Ejercicio1::obtenerInput(istream &input){
2     char testcase[16];
3     input >> testcase;                                // Parametro L
4     if(testcase[0] != TESTCASE_NULL){
5         _cargaMaxima = max(atoi(testcase),0);
6         input >> _cantidadTotalDePaquetes;           // Parametro n
7         _inicializarEstructurasInternas();             // Reserva espacio
8         for(int i=0;i<_cantidadTotalDePaquetes;i++){
9             input >> _listaDePesosDePaquetes[i];       // Parametros p1..pn
10        } else _cargaMaxima = 0;
11        _cantidadDeCamionesOcupados = 0;
12    }
13
14 bool Ejercicio1::inputNoEsNulo( void ){
15     return _cargaMaxima > 0;
16 }
17
18 void Ejercicio1::resolver( void ){
19     for(int i=0; i<_cantidadTotalDePaquetes; i++){
20         int pesoPaquete = _listaDePesosDePaquetes[i];
21         if( !_hayLugarParaCargar( pesoPaquete ) )
22             _nuevoCamion ( pesoPaquete );
23         else
24             _agregaCarga ( pesoPaquete );
25     }
26 }
27
28 void Ejercicio1::imprimirOutput( ostream &output ){
29     output << _cantidadDeCamionesOcupados;
30     for(int i=0; i<_cantidadDeCamionesOcupados; i++)
31         output << " " << _listaDeCamiones[i].carga;
32     output << endl;
33 }
```

3.1.2. Problema 2: Profesores visitantes

Listing 3: ej2.cpp

```

1 #include "ej2.h"
2 int main( int argc, char** argv ){
3     ParserDeParametros parser( argc, argv );
4     if( ! parser.parametrosSonValidos() ) {
5         parser.imprimirAyuda();
6         return -1;
7     }
8     Ejercicio2 ejercicio = Ejercicio2();
9     while( ejercicio.obtenerInput( parser.input() ) ){
10         ejercicio.resolver();
11         ejercicio.imprimirOutput( parser.output() );
12     }
13     return 0;
14 }
```

Listing 4: Ejercicio2.cpp

```

1 /* Inicializo con una instancia 'vacía' del ejercicio */
2 Ejercicio2::Ejercicio2(){ _cantidadTotalDeCursos = 0; }
3
4 bool Ejercicio2::obtenerInput(istream &input){
5     char testcase[BASE10_INT_MAXSIZE];
6     input >> testcase;                                     // Parametro n
7     if(testcase[0] != TESTCASE_NULL){
8         _cantidadTotalDeCursos = max(atoi(testcase),0); // int n > 0 o invalido
9         _inicializarEstructurasInternas();                // Reserva espacio
10        for(uint i=0;i<_cantidadTotalDeCursos;i++){
11            uint inicio, fin;                            // Parametros ik fk
12            input >> inicio >> fin;
13            _listaDeCursos[i]=Curso(inicio, fin, i+1);
14        }
15    } else return false; // Testcase == TESTCASE_NULL
16    return true;
17 }
18
19 void Ejercicio2::resolver( void ){
20     for(uint i=0; i<_cantidadTotalDeCursos; i++){
21         _listaOrdenadaDeCursos.insert( _listaDeCursos[i] );
22     }
23     set<Curso>::iterator it = _listaOrdenadaDeCursos.begin();
24     uint fin = it->fechaDefin, numeroDeCurso = it->numeroDeCurso;
25     _cantidadDeCursosQueSeDictaran = 1;
26     while( it != _listaOrdenadaDeCursos.end() ){
27         if(it->fechaDeInicio > fin && it->fechaDefin > fin){
28             _listaDeCursosQueSeDictaran[_cantidadDeCursosQueSeDictaran-1] =
29                 numeroDeCurso;
30             fin = it->fechaDefin;
31             numeroDeCurso = it->numeroDeCurso;
32             _cantidadDeCursosQueSeDictaran++;
33         }
34         it++;
35     }
36     _listaDeCursosQueSeDictaran[_cantidadDeCursosQueSeDictaran-1] =
37         numeroDeCurso;
38 }
39
40 void Ejercicio2::imprimirOutput( ostream &output ){
```

TP1: TÉCNICAS ALGORITMICAS

```
39 ||    int i=0;
40 ||    for( ;i<(int)_cantidadDeCursosQueSeDictaran-1; i++)
41 ||        cout << _listaDeCursosQueSeDictaran[i] << " ";
42 ||    cout << _listaDeCursosQueSeDictaran[i] << endl;
43 || }
```

3.1.3. Problema 3: Una noche en el museo

Listing 5: ej3.cpp : main

```

1 #include "ej3.h"
2 int main(int argc, char **argv)
3 {
4     int filas, columnas;
5     LCoord celdas_a_procesar;
6     LSensor sensores_solucion;
7     LSensor sensores_solucion_parcial;
8     Solucion solucion(sensores_solucion, 0);
9     Solucion solucion_parcial(sensores_solucion_parcial, 0);
10    VectorMascaras mascaras_filas(16, 0);
11    VectorMascaras mascaras_columnas(16, 0);
12
13    // Paser de Input & Init
14    cin >> filas;
15    cin >> columnas;
16    Grilla grilla(filas, Vec(columnas, 0));
17    VectorOpciones opciones_filas(filas, MASK_OPCIONES_INICIALES);
18    VectorOpciones opciones_columnas(columnas, MASK_OPCIONES_INICIALES);
19    cargar(grilla);
20
21    if( analizar(grilla, celdas_a_procesar,
22                  opciones_filas, opciones_columnas)
23    ){
24        resolver(grilla, celdas_a_procesar, solucion_parcial,
25                  opciones_filas, opciones_columnas, mascaras_filas,
26                  mascaras_columnas, solucion);
27    }
28
29    // Imprimir resultado
30    if( solucion.second == 0 ) cout << "-1" << endl;
31    else {
32        cout << solucion.first.size() << " " << solucion.second << endl;
33        for( int j = 0; j < solucion.first.size();j++ ) {
34            cout << solucion.first[j].second << " ";
35            cout << solucion.first[j].first.first+1 << " ";
36            cout << solucion.first[j].first.second+1 << endl;
37        }
38    }
39    return 0;
40 }
```

Listing 6: ej3.cpp : analizar

```

1 bool analizar(Grilla &grilla,
2                 LCoord &celdas_a_procesar,
3                 VectorOpciones &opciones_filas,
4                 VectorOpciones &opciones_columnas)
5 {
6     int filas = grilla.size();
7     int columnas = grilla[0].size();
8     vector<uint8_t> columna_valida(columnas, 0);
9     vector<uint8_t> fila_valida(filas, 0);
10    VCoord pared_previa_columna(columnas, celdas_a_procesar.end());
11    LCoord::iterator pared_previa_fila = celdas_a_procesar.end();
12    VCoord ultima_celda_libre_columna(columnas, celdas_a_procesar.end());
13    LCoord::iterator ultima_celda_libre_fila = celdas_a_procesar.end();
14    LCoord::iterator itr_celdas_a_procesar;
```

```

15 // Se recorre grilla
16 for( int i = 0; i < filas ;i++){
17     for( int j = 0; j < columnas ; j++){
18         if( grilla[i][j] == CELDA_IMPORTANTE ){
19             grilla[i][j] = LASER_CRUZ;
20             columna_valida[j] = columna_valida[j] | TIENE_CELDAS_IMPORTANTES;
21             fila_valida[i] = fila_valida[i] | TIENE_CELDAS_IMPORTANTES;
22             // Dado que en la fila y columna hay una celda
23             // de estado LaserC, tengo que modificar opciones.
24             marcarFilaComoImportante(grilla, celdas_a_procesar, i,
25                                         opciones_filas, pared_previa_fila);
26             marcarColumnaComoImportante(grilla, celdas_a_procesar, j,
27                                         opciones_columnas, pared_previa_columna[j]);
28             // Si la Reliquia esta rodeada por paredes,
29             // la grilla no tiene solucion.
30             if(estaRodeadaPorParedes(grilla, i, j)){
31                 return false;
32             }
33
34         } else if( grilla[i][j] == CELDA_LIBRE ){
35             grilla[i][j] = VACIA;
36             columna_valida[j] = columna_valida[j] | TIENE_CELDAS_VACIAS;
37             fila_valida[i] = fila_valida[i] | TIENE_CELDAS_VACIAS;
38             celdas_a_procesar.push_back(Coord(i, j));
39             itr_celdas_a_procesar = celdas_a_procesar.end();
40             itr_celdas_a_procesar--;
41             ultima_celda_libre_fila = itr_celdas_a_procesar;
42             ultima_celda_libre_columna[j] = itr_celdas_a_procesar;
43
44         } else if( grilla[i][j] == PARED ){
45             grilla[i][j] = grilla[i][j] | ( MASK_PARED << 4);
46             columna_valida[j] = columna_valida[j] | TIENE_PAREDES;
47             fila_valida[i] = fila_valida[i] | TIENE_PAREDES;
48             celdas_a_procesar.push_back(Coord(i, j)); // Iterador al ultimo
49             itr_celdas_a_procesar = celdas_a_procesar.end();
50             itr_celdas_a_procesar--;
51
52             //Analizo longitud de sub-fila y sub-columna y modiflico opciones
53             procesarFinDeSegmento(grilla, celdas_a_procesar,
54                                     ultima_celda_libre_fila, ultima_celda_libre_columna[j]);
55             analizarLargoSegmentoHorizontal(grilla, celdas_a_procesar, i, j,
56                                             opciones_filas, pared_previa_fila);
57             analizarLargoSegmentoVertical(grilla, celdas_a_procesar, i, j,
58                                             opciones_columnas, pared_previa_columna[j]);
59             pared_previa_fila = itr_celdas_a_procesar;
60             pared_previa_columna[j] = itr_celdas_a_procesar;
61         }
62     }
63 }
64
65 // Aplico Mascara MASK_ULTIMA_CELDA_LIBRE_FILA
66 // a la ultima celda libre de la fila
67 if( ultima_celda_libre_fila != celdas_a_procesar.end() ){
68     grilla[(*ultima_celda_libre_fila).first]
69     [(*ultima_celda_libre_fila).second]
70     = ( grilla[(*ultima_celda_libre_fila).first]
71         [(*ultima_celda_libre_fila).second] )
72     | ( MASK_ULTIMA_CELDA_LIBRE_FILA );
73     ultima_celda_libre_fila = celdas_a_procesar.end();
74 }
75     pared_previa_fila = celdas_a_procesar.end();
76 }
77

```

```

78 // Aplico mascara MASK_ULTIMA_CELDA_LIBRE_COL
79 // a la ultima celda de cada columna
80 for( int j = 0; j<ultima_celda_libre_columna.size();j++){
81     if( ultima_celda_libre_columna[j] != celdas_a_procesar.end() ){
82         grilla[(*ultima_celda_libre_columna[j]).first]
83             [(*ultima_celda_libre_columna[j]).second]
84             = ( grilla[(*ultima_celda_libre_columna[j]).first]
85                 [(*ultima_celda_libre_columna[j]).second] )
86             | ( MASK_ULTIMA_CELDA_LIBRE_COL );
87         ultima_celda_libre_columna[j] = celdas_a_procesar.end();
88     }
89 }
90
91 // Analizamos mascaras de estados para ver si las filas son validas,
92 // es decir, si tienen solucion
93 for( int i = 0; i < filas;i++){
94     if( fila_valida[i] == 2 || fila_valida[i] == 3 ) return false;
95 }
96 // Analizamos mascaras de estados para ver si las columnas son validas,
97 // es decir, si tienen solucion
98 for( int i = 0; i < columnas;i++){
99     if( columna_valida[i] == 2 || columna_valida[i] == 3 ) return false;
100 }
101
102 // Detectamos Sub-columnas de 1 Celda de longitud en la
103 // ultima fila de la grilla
104 if( filas > 1 ){
105     for( int i = 0;i < columnas;i++){
106         if( esVacia(grilla[filas-1][i]) == true
107             && esPared(grilla[filas-2][i]) == true ){
108             grilla[filas-2][i]
109             = grilla[filas-2][i] & ( MASK_PARED_INICIO_COL_1_CELDA << 4 );
110         }
111     }
112 } else {
113     for( int i = 0; i < columnas; i++){
114         opciones_columnas[i]
115             = opciones_columnas[i] & ( MASK_PARED_INICIO_COL_1_CELDA >> 9 );
116         if( grilla[0][i] == LASER_CRUZ ) return false;
117     }
118 }
119
120 // Detectamos Sub-filas de 1 Celda de longitud en la
121 // ultima columna de la grilla
122 if( columnas > 1 ){
123     for( int i = 0;i < filas;i++){
124         if( esVacia(grilla[i][columnas-1])
125             && esPared(grilla[i][columnas-2]) == true ){
126             grilla[i][columnas-2]
127             = grilla[i][columnas-2] & ( MASK_PARED_INICIO_FILA_1_CELDA << 4 );
128         }
129     }
130 } else {
131     for( int i = 0; i < filas; i++){
132         opciones_filas[i]
133             = opciones_filas[i] & MASK_PARED_INICIO_FILA_1_CELDA;
134         if( grilla[i][0] == LASER_CRUZ ) return false;
135     }
136 }
137
138 return true;
139 }
```

Listing 7: ej3.cpp : resolver

```

1 void resolver(Grilla &g,
2                 LCoord &celdas_a_procesar,
3                 Solucion &solucion_parcial,
4                 VectorOpciones &opciones_filas,
5                 VectorOpciones &opciones_columnas,
6                 VectorMascaras &mascaras_filas,
7                 VectorMascaras &mascaras_columnas,
8                 Solucion &solucion)
9 {
10    LCoord::iterator itr = celdas_a_procesar.begin();
11    uint32_t opciones_originales_fila;
12    uint32_t opciones_originales_columna;
13    int fila, columna;
14    int tipo = 0;
15    int valor_original;
16
17    if( itr != celdas_a_procesar.end() ){
18        //Calculo Interseccion Opciones Sensores entre Fila y columna
19        Coord celda = (*itr);
20        fila = celda.first;
21        celdas_a_procesar.pop_front();
22        columna = celda.second;
23        //Guardo estados de opciones/requerimientos para fila/columna
24        opciones_originales_fila = opciones_filas[fila];
25        opciones_originales_columna = opciones_columnas[columna];
26        valor_original = g[fila][columna];
27        if( esPared(g[fila][columna]) ){
28            opciones_filas[fila] = ( g[fila][columna] >> 4 ) & 0x1FF;
29            opciones_columnas[columna] = ( g[fila][columna] >> 13 ) & 0x1FF;
30            resolver(g, celdas_a_procesar, solucion_parcial, opciones_filas,
31                      opciones_columnas, mascaras_filas, mascaras_columnas, solucion);
32            opciones_filas[fila] = opciones_originales_fila;
33            opciones_columnas[columna] = opciones_originales_columna;
34        } else {
35            // En el caso de que no sea una pared, seguro es una celda libre VACIA
36            uint32_t requerimiento_fila = opciones_filas[fila] >> 7;
37            uint32_t requerimiento_columna = opciones_columnas[columna] >> 7;
38            uint32_t opciones_celdas =
39                ( opciones_originales_fila & opciones_originales_columna )
40                & puedeContenerLaser(g, g[fila][columna], requerimiento_fila,
41                                      requerimiento_columna);
42            uint32_t precio = 0;
43            for(int i = 1; i<= 6;i++){
44                if( ( opciones_celdas >> i ) & 1 ){
45                    if( esSensor(i) )
46                        solucion_parcial.first.push_back(Sensor(celda, i));
47                    g[fila][columna] = i;
48                    opciones_filas[fila] =
49                        ( opciones_filas[fila] & mascaras_filas[i] & 0x7E )
50                        | nuevoRequerimientoFila(i, requerimiento_fila);
51                    opciones_columnas[columna] =
52                        ( opciones_columnas[columna] & mascaras_columnas[i] & 0x7E )
53                        | nuevoRequerimientoCol(i, requerimiento_columna);
54                    precio = obtenerPrecio(i);
55                    if( solucion_parcial.second + precio < solucion.second
56                        || solucion.second == 0){
57                        // Si solucion parcial sigue por debajo de la mejor
58                        // solucion encontrada, sumo precio sensor a solucion_parcial
59                        solucion_parcial.second += precio;
60

```

```

61     //llamo recursivamente
62     resolver
63     (g, celdas_a_procesar, solucion_parcial, opciones_filas,
64      opciones_columnas, mascaraas_filas, mascaraas_columnas,solucion);
65     //restauro precio
66     solucion_parcial.second -= precio;
67 }
68 //Restauro estado previo a la seleccion de la opcion
69 if( esSensor(i))
70     solucion_parcial.first.pop_back();
71 g[fila][columna] = valor_original;
72 opciones_filas[fila] = opciones_originales_fila;
73 opciones_columnas[columna] = opciones_originales_columna;
74 }
75 }
76 }
77 celdas_a_procesar.push_front(celda);
78 } else {
79 // Si ya no quedan celdas a procesar, tenemos una solucion
80
81 // Si la nueva solucion es menos costosa que la actual, reemplazamos
82 if( solucion_parcial.second < solucion.second || solucion.second == 0){
83     solucion.second = solucion_parcial.second;
84     solucion.first.clear();
85     for( int j = 0; j < solucion_parcial.first.size();j++ )
86         solucion.first.push_back(solucion_parcial.first[j]);
87 }
88 }
89 return;
90 }
```

3.2. Informe de Modificaciones

Problema 1: Conejito de Pascual y el correo

- Se corrigió el concepto del algoritmo de Pascual, mostrando que no es óptimo.
- Se agregaron ejemplos de mejor y peor caso en la descripción del problema.
- En la hipótesis de resolución se corrigió el hecho de que Pascual cumple las condiciones de algoritmo goloso pero su subestructura no es óptima, con lo cual no logra obtener la solución que minimice. También se corrigió la mal usada estructura «heapsort» por una cola de prioridad.
- Nota: en el *Algoritmo 1*, línea 26, la variable *camionesOcupados* además de como contador, también se usa como índice, comenzando desde 0.
- Se agregó un segundo gráfico en donde se comparan tres funciones: cota estimada, peores casos y casos de pesos pseudoaleatorios.

Problema 2: Profesores visitantes del más allá

- Se rehizo por completo la explicación de correctitud.
- Se agregaron los casos de prueba

Problema 3: Una noche en el museo con Ben Stiller

- Se agregó calculo de complejidad teórica
- Se agregaron representaciones gráficas de las soluciones en la sección «Verificación mediante casos de prueba», a fin de facilitar la interpretación de la salida.
- Se extendió la justificación de correctitud.
- Se agregaron gráficos de pruebas empíricas de la performance.
- Se corrigió tabla de estados en la sección «Hipótesis de resolución», faltaban enumerar estados posibles.

Generales

- Se corrigió el tiempo verbal utilizado en los objetivos.
- Se realizaron correcciones menores estructurales entre los distintos párrafos y oraciones, moviendo contenido de lugar, adaptando palabras, reemplazando algunas frases o palabras por sinónimos, para enriquecer el vocabulario y amenizar la lectura, sin alterar el contenido esencial.
- Se movieron los códigos fuentes que se encontraban dentro de el contenido de cada problema hacia la sección de apéndices.

3.3. Bibliografía

Referencias

- [1] G. Brassard, P. Bratley, *Fundamental of Algorithmics*, Prentice Hall, 1996, Chapter 6.2, «General characteristics of greedy algorithms», Chapter 9.6, «Backtracking»,
- [2] Cormen, Leiserson, Rivest *Introduction to Algorithms*, 2001, Chapter 16 «Greedy Algorithms».
- [3] <http://www.cplusplus.com/reference/stl/>
- [4] <http://www.cplusplus.com/reference/set/set/>
- [5] <http://www.cplusplus.com/reference/vector/vector/>
- [6] http://www.cplusplus.com/reference/queue/priority_queue/
- [7] http://en.wikipedia.org/wiki/Greedy_algorithm#Specifics
- [8] http://en.wikipedia.org/wiki/Backtracking#Usage_considerations

