



Técnicas Algorítmicas Avanzadas

Algoritmos y Estructuras de Datos III
Entrega de TP

Integrante	LU	Correo electrónico
Barrios, Leandro E.	404/11	ezequiel.barrios@gmail.com
Benegas, Gonzalo	958/12	gsbenegas@gmail.com
Duarte, Miguel	904/11	miguelfeliped@gmail.com
Niikado, Marina	711/07	mariniik@yahoo.com.ar

<http://www.fcen.uba.ar>

Índice

1. Introducción	3
1.1. Objetivos	3
1.2. Pautas de trabajo	3
1.3. Metodología utilizada	3
2. Instrucciones de uso	5
2.1. Herramientas utilizadas	5
3. Desarrollo del TP	6
3.1. Problema 1: Robanúmeros	6
3.1.1. Descripción	6
3.1.2. Planteamiento de resolución	7
3.1.3. Justificación formal de correctitud	9
3.1.4. Cota de complejidad temporal	9
3.1.5. Verificación mediante casos de prueba	9
3.1.6. Medición empírica de la Performance	10
3.2. Problema 2: La centralita (de gas)	11
3.2.1. Descripción	11
3.2.2. Planteamiento de resolución	12
3.2.3. Justificación formal de correctitud	14
3.2.4. Cota de complejidad temporal	15
3.2.5. Verificación mediante casos de prueba	16
3.2.6. Medición empírica de la Performance	17
3.3. Problema 3: Saltos en La Matrix	18
3.3.1. Descripción	18
3.3.2. Planteamiento de resolución	18
3.3.3. Justificación formal de correctitud	19
3.3.4. Cota de complejidad temporal	19
3.3.5. Verificación mediante casos de prueba	19
3.3.6. Medición empírica de la Performance	21
4. Apéndices	22
4.1. Código Fuente (resumen)	22

1. Introducción

1.1. Objetivos

Mediante la realización de este trabajo práctico se pretende realizar un acercamiento al análisis e implementación de técnicas algorítmicas avanzadas para resolución de problemas, como así también a las estructuras que permiten su implementación.

En esta ocasión se hace énfasis en las técnicas que involucran el uso de *grafos*, con los distintos algoritmos que permiten recorrerlos, y los denominados *algoritmos dinámicos*.

1.2. Pautas de trabajo

Se brindan tres problemas, escritos en términos coloquiales, en donde para cada uno de ellos se requiere **encontrar un algoritmo** que brinde una **solución particular**, acotado por una determinada **complejidad temporal**. El algoritmo debe ser **implementado** en un lenguaje de programación a elección. Los datos son proporcionados y deben ser devueltos bajo formatos específicos de *input* y de *output*.

Posteriormente se deben realizar análisis teóricos y empíricos tanto de la **correctitud** como de la **complejidad temporal** para cada una de las soluciones propuestas.

1.3. Metodología utilizada

Para cada ejercicio, se brinda primeramente una **descripción** del problema planteado, a partir de la cual se realiza una **abstracción** hacia un **modelo formal**, que permite tener un **entendimiento preciso** de las pautas requeridas.

Se expone, cuando las hay, una **enumeración de las características** elementales del problema; estas son aquellas que permiten **encuadrarlo** dentro de una **familia de problemas** típicos.

Se desarrolla posteriormente un análisis del conjunto **universo de posibles soluciones** (o factibles), caracterizando matemáticamente el concepto de **solución correcta** y, en los casos en que se solicita **optimización**¹, se definen las condiciones que dan forma ya sea a todo el subconjunto de **soluciones óptimas** que se encuadran dentro de las pretensiones del problema, o a una **solución particular** dentro del mismo (la cual denominamos *mejor solución*).

Luego de caracterizar para todo conjunto posible de entradas «*cómo se compone el conjunto solución*» correspondiente, se desarrolla un **pseudocódigo** en el que se expone «*cómo llegar a ese conjunto*»².

Habiendo planteado la **hipótesis de resolución** se demuestra, de manera informal o mediante inferencias matemáticas según sea necesario, que el **algoritmo propuesto** realmente permite obtener la **solución correcta**³.

Después de demostrar la **correctitud de la solución**, y su **optimalidad** en caso de existir varias soluciones correctas, se realiza un análisis teórico de la **complejidad temporal** en donde se estima el comportamiento del algoritmo en términos de tiempo. Este análisis en particular se realiza con el objetivo de obtener una *cota superior asintótica*.

¹Es decir, que la solución pertenezca al *subconjunto de soluciones que maximicen o minimicen una determinada función*

²Una explicación coloquial, obviando detalles puramente implementativos: arquitectura, lenguaje, etc.

³En caso de existir más de una solución correcta, se demuestra que el algoritmo obtiene al menos una de ellas o, dicho de otro modo, para problemas de optimización, se demuestra que ninguna del resto de las soluciones correctas es mejor que la solución propuesta por nuestro algoritmo

Luego de calcular la **cota de complejidad temporal**, se realiza una **verificación** empírica, junto con una **exposición gráfica** de los resultados obtenidos, mediante la combinación de técnicas básicas de medición y análisis de datos.

2. Instrucciones de uso

2.1. Herramientas utilizadas

Para la realización de este trabajo se utilizaron un conjunto de herramientas, las cuales se enumeran a continuación:

- C++ como lenguaje de programación
 - gcc como compilador de C++
- python y bash para la realización de scripts
 - python para generar casos de prueba
 - bash para automatizar las mediciones
 - python/matplotlib para plotear los gráficos
- L^AT_EX para la redacción de este documento
- Se testeó bajo los siguientes Sistemas Operativos
 - Debian GNU/Linux
 - Ubuntu
 - FreeBSD, compilando a través de gmake
 - Windows, a través de cygwin

3. Desarrollo del TP

3.1. Problema 1: Robanúmeros

3.1.1. Descripción

En este problema se tiene que crear un algoritmo que juegue al *Robanúmeros* de forma tal que el jugador1 logre el mejor juego posible, y el jugador2 juegue de manera óptima durante cada turno que le toque. El algoritmo tiene que tener una complejidad temporal de peor caso de $\mathcal{O}(n^3)$, con n la cantidad de cartas iniciales.

Reglas del *Robanúmeros*:

- Comienzo del juego:
Se tiene una cantidad n ($n \in \mathbb{N}$) de cartas con valores enteros alineadas horizontalmente (c_1, c_2, \dots, c_n) sobre la mesa. Las cartas tienen que estar boca arriba.
- Turnos:
Participan 2 jugadores, cada uno va alternando un turno. (Total de turnos t : $1, \dots, n$).
- Elección de cartas:
En cada turno el jugador tiene que elegir un extremo, el izquierdo (izq) o el derecho (der), de la secuencia de cartas desde el que irá tomando de 1 a n de las cartas adyacentes que están en la mesa. La cantidad de cartas elegidas variará según le sea conveniente al jugador, pero por lo menos tiene que tomar una carta en su turno.
- Fin del juego:
El juego finaliza cuando no hay más cartas sobre la mesa. Se suman las cartas de cada jugador (p_1 : Ptos. Jug1, p_2 : Ptos. Jug2). Gana el que obtiene el mayor puntaje.

Ejemplo 3.1.1.1.

- Cartas iniciales:

2	-3	-2	5	5
---	----	----	---	---

- Turno1 (Jug1): Elige el extremo derecho y toma las 2 últimas cartas.

5	5
---	---

- Quedan sobre la mesa:

2	-3	-2
---	----	----

- Turno2 (Jug2): Elige el extremo izquierdo y toma 1 carta.

2

- Quedan sobre la mesa:

-3	-2
----	----

- Turno3 (Jug1): Elige el extremo derecho y toma 1 carta.

-2

- Quedan sobre la mesa:

-3

- Turno4 (Jug2): Sólo queda una carta, por lo que elige ésta. Es indistinto para este caso si el extremo elegido es el izquierdo o el derecho. -3
- Finaliza el juego porque no hay más cartas. Se suman los puntajes de cada jugador.

Ptos. Jug1	Ptos. Jug2
$5 + 5 + (-2) = 8$	$2 + (-3) = -1$

- Formato de entrada y salida:

Input: 5 2 -3 -2 5 5

Output: 4 8 -1
 der 2
 izq 1
 der 1
 izq 1

Ejemplo 3.1.1.2.

2 -1 6

El Jug1 toma todas las cartas porque de esta manera obtiene el mayor puntaje. Finaliza el juego en 1 turno porque no hay más cartas. Se suman los puntajes de cada jugador. Este tipo de caso también se daría si todas las cartas tuvieran números positivos, sólo llegaría a jugar el Jug1.

Ptos. Jug1	Ptos. Jug2
$2 + (-1) + 6 = 7$	0

3.1.2. Planteamiento de resolución

Para una mayor claridad, vamos a reducir el problema a encontrar la mayor cantidad de puntos que se pueden sacar con el juego de cartas dado.

Sea $f(i,j)$ = "maxima cantidad de puntos que se pueden sacar en el juego que consiste en las cartas que estaban desde la posición i hasta la j en el juego de cartas original".

Sea $\text{suma}(i,j)$ la suma de las cartas que estaban entre la posición i y j en el juego de cartas original.

Nuestro algoritmo se basa en que $f(i,j) = \text{— valor de la carta i si } i=j$

— $\text{suma}(i, j) - \min(f(i', j'))$ para todo $i'j'$ que representen un juego que le dejo al oponente usando una movida válida sino

[habra que justificar esto?]

Pseudocódigo

Algoritmo 1 Roba Cartas

Entrada:

cantCartas \leftarrow DAMECANTCARTAS \triangleright *integer*
cartas \leftarrow DAMEARREGLOCARTAS \triangleright *arreglo(integer)*

Salida:

MEJOR PUNTAJE PRIMER JUGADOR \triangleright *integer*
 MEJOR PUNTAJE SEGUNDO JUGADOR \triangleright *integer*
 CANTIDAD DE TURNOS QUE DURA EL PARTIDO \triangleright *integer*
 LISTA DE LEVANTES \triangleright *lista < integer >*

Estructura Levante:

dirección \triangleright *Bool*
cantidad \triangleright *integer*

Estructura Jugada:

mejorPuntaje \triangleright *Integer*
turnosHastaAhora \triangleright *Integer*
levanteRealizado \triangleright *Levante*

Variables Globales

matrizJugadas \triangleright *Matriz < Jugada > tamaño : cantCartas + 1, cantCartas + 1*
sumasParciales \triangleright *Arreglo < Integer > tamaño : cantCartas + 1*

Se generan las sumas parciales

1: *sumasParciales*₀ \leftarrow 0 \triangleright $\mathcal{O}(1)$
 2: **para** cada *i* en $[0, \text{cantCartas} - 1]$ **hacer** \triangleright $\mathcal{O}(n)$
 3: *sumasParciales*_{*i*+1} \leftarrow *cartas*_{*i*} + *sumasParciales*_{*i*} \triangleright $\mathcal{O}(1)$
 4: **fin para**

Se inicializa la matriz de jugadas con cero en todas sus posiciones.

5: **para** cada posición en *matrizJugadas* **hacer** \triangleright $\mathcal{O}(n^2)$
 6: *matrizJugadas*_{posición} \leftarrow 0 \triangleright $\mathcal{O}(1)$
 7: **fin para**

Se guardan primero la solución trivial. La de los subjugos de tamaño 1

8: **para** *i* en $[0, \text{cantCartas} - 1]$ **hacer** \triangleright $\mathcal{O}(n)$
 9: *matrizJugadas*_{*i*,*i*}.*mejorPuntaje* \leftarrow *cartas*_{*i*} \triangleright $\mathcal{O}(1)$
 10: *matrizJugadas*_{*i*,*i*}.*turnosHastaAhora* \leftarrow 1 \triangleright $\mathcal{O}(1)$
 11: *matrizJugadas*_{*i*,*i*}.*levanteRealizado.dirección* \leftarrow TRUE \triangleright $\mathcal{O}(1)$
 12: *matrizJugadas*_{*i*,*i*}.*levanteRealizado.cantidad* \leftarrow 1 \triangleright $\mathcal{O}(1)$
 13: **fin para**

Se rellena el resto de la matriz

14: **para** *tamSubConj* en $[2, \text{cantCartas}]$ **hacer** \triangleright $\mathcal{O}(n^2)$
 15: *principio* \leftarrow 0 \triangleright $\mathcal{O}(1)$
 16: *final* \leftarrow *tamSubConj* \triangleright $\mathcal{O}(1)$
 Se miran todos los subjugos posibles de cada tamaño.
 17: **mientras** *final* \leq *cantCartas* **hacer** \triangleright $\mathcal{O}(n)$
 18: *sumaParcial* \leftarrow *sumasParciales*_{*final*} - *sumasParciales*_{*principio*} \triangleright $\mathcal{O}(1)$
 19: *peorJugada* \triangleright *Jugada*, $\mathcal{O}(1)$
 20: *peorJugada.mejorPuntaje* \leftarrow *infinito* \triangleright $\mathcal{O}(1)$
 21: *levanteCorrecto* \triangleright *Levante*, $\mathcal{O}(1)$
 22: **para** subjuego en [subjuego posibles] **hacer**
 23: **si** *matrizJugadas*_{PRINCIPIO(subjuego),FINAL(subconjunto)} < *peorJugada* **entonces** \triangleright $\mathcal{O}(1)$
 24: *peorJugada* \leftarrow *matrizJugadas*_{PRINCIPIO(subjuego),FINAL(subconjunto)} \triangleright $\mathcal{O}(1)$
 25: *levanteCorrecto* \leftarrow LEVANTEPARALLEGARA(subjuego) \triangleright $\mathcal{O}(1)$
 26: **fin si**
 27: **fin para**
 28: *nuevaJugada* \triangleright $\mathcal{O}(1)$
 29: *nuevaJugada.mejorPuntajePosible* \leftarrow *sumaParcial* - *peorJugada.mejorPuntaje* \triangleright $\mathcal{O}(1)$
 30: *nuevaJugada.turnosHastaAhora* \leftarrow *peorJugada.turnosHastaAhora* + 1 \triangleright $\mathcal{O}(1)$
 31: *nuevaJugada.levanteRealizado* \leftarrow *levanteCorrecto* \triangleright $\mathcal{O}(1)$
 32: *matrizJugadas*_{*principio*,*final*-1} \leftarrow *nuevaJugada* \triangleright $\mathcal{O}(1)$
 33: *principio* ++ \triangleright $\mathcal{O}(1)$
 34: *final* ++ \triangleright $\mathcal{O}(1)$
 35: **fin mientras**
 36: **fin para**

La mejor jugada del juego total está en la posición 0, cantCartas-1	
37: $mejorPuntaje \leftarrow matrizJugadas_{0, cantCartas-1}$	$\triangleright \mathcal{O}(1)$
38: $puntajeEnemigo \leftarrow sumasParciales_{cantCartas} - mejorPuntaje$	$\triangleright \mathcal{O}(1)$
Ahora se revisan las jugadas realizadas	
39: $fin \leftarrow cantCartas - 1$	$\triangleright \mathcal{O}(1)$
40: $init \leftarrow 0$	$\triangleright \mathcal{O}(1)$
41: $turnos \leftarrow 0$	$\triangleright \mathcal{O}(1)$
42: $levantes \leftarrow NUEVALISTA()$	$\triangleright Lista < Levante > \mathcal{O}(1)$
43: mientras $init \leq fin$ hacer	$\triangleright \mathcal{O}(n)$
44: $levanteActual \leftarrow matrizJugadas_{init, fin}.levanteRealizado$	$\triangleright \mathcal{O}(1)$
45: AGREGAR($levantes, levanteActual$)	$\triangleright \mathcal{O}(1)$
46: si $levanteActual.direccion = IZQ$ entonces	$\triangleright \mathcal{O}(1)$
47: $fin \leftarrow fin - levanteActual.cantidad$	$\triangleright \mathcal{O}(1)$
48: sino	
49: $init \leftarrow init + levanteActual.cantidad$	$\triangleright \mathcal{O}(1)$
50: fin si	
51: $turnos++$	$\triangleright \mathcal{O}(1)$
52: fin mientras	
53: retornar $mejorPuntaje$	$\triangleright \mathcal{O}(1)$
54: retornar $puntajeEnemigo$	$\triangleright \mathcal{O}(1)$
55: retornar $turnos$	$\triangleright \mathcal{O}(1)$
56: retornar $levantes$	$\triangleright \mathcal{O}(1)$

3.1.3. Justificación formal de correctitud

á

3.1.4. Cota de complejidad temporal

á

3.1.5. Verificación mediante casos de prueba

A continuación presentamos distintas instancias que sirven para verificar que el programa funciona correctamente.

Input	Output		
n c_1 \dots c_n	t	$p1$	$p2$
	e_1	c_1	
	\vdots	\vdots	
	e_t	c_t	

- n : #cartas iniciales.
- c_i con $1 \leq i \leq n$: c_i valor de la carta i .
- t : #turnos del juego.
- $p1$: puntaje total Jug1.
- $p2$: puntaje total Jug2.
- e_i con $1 \leq i \leq t$: e_i extremo elegido por el jugador en el turno i (izq o der).
- c_i con $1 \leq i \leq t$: c_i #cartas tomadas por el jugador en el turno i .

Según los valores de p1 y p2, podemos separar en 3 casos posibles:

1. Caso Empate entre Jug1 y Jug2:

- Cartas con valor cero

Input
3 0 0 0

Output
1 0 0
izq 3

- Cartas con valores negativos

Input
3 -1 -2 -3

Output
2 -3 -3
izq 2
izq 1

2. Caso Perdedor Jug1:

- Cartas con valores negativos

Input
3 -2 -3 -1

Output
2 -4 -2
der 2
izq 1

3. Caso Ganador Jug1:

- Cartas con valores positivos

Input
3 1 2 3

Output
1 6 0
izq 3

- Cartas con valores negativos

Input
3 -5 -1 -3

Output
2 -4 -5
der 2
izq 1

- Cartas con valores positivos y negativos

Input
4 2 -8 -8 3

Output
4 -5 -6
der 1
izq 1
izq 1
izq 1

Ejecutamos el programa con los distintos ejemplos y se llegó a la solución esperada. Por lo tanto, podemos concluir que el comportamiento del programa es correcto.

3.1.6. Medición empírica de la Performance

3.2. Problema 2: La centralita (de gas)

3.2.1. Descripción

Planteo del Problema

Existe una región del país en la que *un grupo de pueblos no cuenta con red de gas natural*. Luego de una fuerte campaña política se lograron recaudar los fondos necesarios para *emprender una obra que provea del servicio a todos los pueblos de la región*.

El sistema consistirá en **una red de tuberías interconectadas de un pueblo al otro**, de forma tal que *la distribución de la misma asegure el abastecimiento* de cada uno de los pueblos, y en donde además **se seleccionarán determinados pueblos para establecer centralitas**, que serán las encargadas de *proveer gas hacia todo pueblo con el que cuenten con conexión*, ya sea mediante una tubería o a través de un camino de tuberías.

Debido a que el costo de las cañerías es significativamente menor que el de las centralitas, **el presupuesto final estará regido por la cantidad de centralitas** que se construyan (y viceversa, según quién lo mire). Por lo antes expuesto se conoce el cálculo que permite predecir, habiendo reunido una cantidad determinada de presupuesto, cual es la **cantidad máxima de centralitas** correspondientes que el mismo permite construir.

Se entiende como “**riesgo**” de una determinada distribución de tuberías a la **máxima de sus longitudes**.

El codicioso Fontanero Jefe, Mario Toti Segale⁴, desea conocer, dado un determinado “presupuesto máximo”, es decir, una “cantidad máxima de centralitas”, cuál es la distribución óptima de tuberías y centralitas de forma tal que el gasto sea menor al presupuestado. Para ello encomendó la tarea a su tímido hermano Luigi. Dado que Luigi es precavido y miedoso por naturaleza, decidió que el gasto no importaba realmente, siempre que fuera menor al presupuestado, y que una distribución óptima era más bien aquella en la que el riesgo resultase mínimo.

Requerimientos técnicos

El problema requiere encontrar una distribución insuperable de gaseoductos, recibiendo como **datos de entrada** la cantidad de ciudades (n), la cantidad máxima de núcleos gaseosos (k), y n pares de números enteros (x, y) representando las coordenadas euclídeas de cada una de las borneras (hay una por pueblo), y devolviendo como **datos de salida** la cantidad de núcleos gaseosos (q) y caños (m) construídos, junto con un listado en donde se detallen cada uno de los q pueblos (p) en donde se emplazará una central, y un segundo listado en donde a través de m pares de números enteros (v, w) se detallen cada una de las tuberías a construir. El algoritmo implementado debe respetar una complejidad temporal de peor caso de $\mathcal{O}(n^2)$.

Formato de los datos

Formato de entrada

$n \ k$
 $x_1 \ y_1$
 \cdot
 \cdot
 \cdot
 $x_n \ y_n$

Formato de salida

$q \ m$
 $p_1 \ \dots \ p_q$
 $v_1 \ w_1$
 \cdot
 \cdot
 \cdot
 $v_m \ w_m$

⁴“No es más que un italojaponés americano... gordo y bigotudo” - descripción anónima de un empleado.

Planteo del problema Existe una región del país en la que un grupo de pueblos no cuenta con red de gas natural.

3.2.2. Planteamiento de resolución

Modelado del problema

Este es un problema típico en que una **estructura de grafos** permite fácilmente realizar una visualización simple y práctica del escenario, como así también encaminar el análisis del universo de soluciones hacia un posible recorrido, construcción o *deconstrucción*⁵ de grafos.

En este caso en particular, representamos una **solución a una instancia** determinada del problema como un **subgrafo generador ponderado no dirigido**, en donde cada pueblo está representado por un vértice $v \in V$, y en donde cada cañería es una arista $e \in E_S$, siendo E_S subconjunto del conjunto de aristas del grafo completo de n vértices. Refiriendonos a este último detalle, y en un *abuso de notación*, afirmamos que S es subconjunto de K_n :

$$S = (V, E_S) \subseteq K_n = (V, E_K)$$

Aclaración: De aquí en adelante se utilizará la letra K para referirse tanto al grafo completo como a la cantidad de centralitas. Interpretar según el contexto.

Función peso

Establecemos además la **función peso**, $p : E \rightarrow \mathbb{R}$, siendo $e \in E$ una tupla $(inicio, fin)$, en donde $inicio$ y fin son dos vértices, como la distancia euclídea entre el pueblo representado por el vértice de inicio y el pueblo representado por el vértice de fin, es decir, el módulo del vector que resulta de la resta de ambos:

$$p(e) = dist(e.inicio, e.fin) = \| inicio - fin \| = \sqrt{(x_{fin} - x_{inicio})^2 + (y_{fin} - y_{inicio})^2}$$

Función objetivo

Se pide seleccionar una **solución óptima** a partir de un **conjunto de soluciones factibles**, estableciendo la **función objetivo** $f : S \rightarrow \mathbb{R}$ como aquella que dada una solución devuelve el peso de la mayor arista, la cual deberá ser minimizada:

$$f(s) = \max(\{p(e) : e \in E_s\})$$

Caracterización de la solución

Dado un conjunto de **soluciones factibles**, es evidente que las **soluciones óptimas** son un subconjunto del mismo. Dada cualquier solución, sin importar si esta es factible o no, y debido a que cada solución es un subgrafo generador de K_n , es posible determinar que la misma contendrá una cantidad de componentes conexas mayor o igual a 1, y menor o igual a n .

Dado que **se disponen de a lo sumo k centralitas**, diremos que una solución es factible cuando la misma se compone de **a lo sumo k componentes conexas**.

Abuso de notación: Representamos S_i como “una solución de i componentes conexas”.

⁵«Deshacer analíticamente los elementos que constituyen una estructura conceptual», es decir: desarmar, analizar, modificar y/o reconstruir una estructura, en este caso un grafo, la cual se encuentra ya previamente armada, ya sea de forma explícita o implícita.

Dada una solución factible $S_x = (V, E_x) \subseteq K_n$ (sin importar si esta es o no óptima), si la misma está compuesta por una cantidad x de **componentes conexas, estrictamente menor a k** , entonces podemos afirmar que existe una solución factible $S_k(V, E_k) \subseteq S_x \subseteq K_n$, de tal forma que el conjunto E_k se forma a partir de quitarle, ya sea una o sucesivas veces, la arista de mayor tamaño al conjunto E_x , repitiendo el proceso siempre y cuando la solución resultante de cada eliminación de arista esté compuesta por a lo sumo k componentes conexas.

En otras palabras, **dada una solución de menos de k componentes conexas**, siempre es posible encontrar a partir de la misma **otra solución de exactamente k componentes conexas**. **Decimos entonces que $f(s_k) \leq f(s_x)$** , y la demostración de esto es trivial ya que al depender f del peso de la máxima arista, es imposible que la valuación de la misma aumente al retirar una o más aristas.

De esta forma, y a efectos de simplificar el análisis, podemos acotar el conjunto de soluciones factibles por el de **todas las soluciones de exactamente k componentes conexas**.

Idea de resolución

Se utilizará de forma parcial el **Algoritmo de Kruskal**, es decir que partiendo de un grafo solución inicial S_n , conformado por n subgrafos triviales, y siendo k el número de centralitas, el ciclo será frenado al llegar a la « $n - k''$ iteracion» o, dicho de otro modo, al formar un subgrafo de k componentes conexas. Decimos, entonces, que el bosque resultante pertenece al conjunto de soluciones óptimas.

Pseudocódigo

Algoritmo 2 Kruskal

```

1: ComponenteConexa componentesConexas[n]
2:
3: estructura ComponenteConexa:
4:   float distancias[n]
5:   list< arista > aristas
6:   arista aristaMasCortaHacia[n]
7:   float distanciaMasCorta
8:   int indiceCCMasCerca
9:
10: para cada i in 1 to n hacer                                ▷  $\mathcal{O}(n^2)$ 
    componentesConexas[i] = i-esimo pueblo                    ▷  $\mathcal{O}(1)$ 
11:   para cada j in 1 to n hacer                                ▷  $\mathcal{O}(n)$ 
12:     aristaMasCorta entre la CC i y la CC j = (i, j)        ▷  $\mathcal{O}(1)$ 
13:     distancia entre componentesConexas[i] y la componente conexa j = distanciaEuclidea(pueblo i,
    pueblo j)                                                  ▷  $\mathcal{O}(1)$ 
14:     me voy fijando cual de estas distancias es mas corta y la guardo junto con el indice j    ▷  $\mathcal{O}(1)$ 
15:   fin para
16: fin para
17:
18: para cada i in 1 to n - k hacer                                ▷  $\mathcal{O}((n * (n - k)) = \mathcal{O}(n^2))$ 
    me fijo la distancia mas corta entre dos componentes
19:   para cada i in 1 to n hacer                                ▷  $\mathcal{O}(n)$ 
    observacion: si componentes[i] ya no representa más una componente continuo
20:     me voy fijando qué componente tiene la menor 'menor distancia hacia otra componente'y la guardo
    en CCAUnir1, su componente mas cercana en CCAUnir2          ▷  $\mathcal{O}(1)$ 
21:   fin para
22:
    uno CCAUnir1 y CCAUnir2
23:   CCAUnir1.aristas = CCAUnir1.aristas union CCAUnir2.aristas + aristaMasCorta entre CCAUnir1 y
    CCAUnir2                                                    ▷  $\mathcal{O}(1)$ 
24:   marco CCAUnir2 como que ya no representa una componente conexa. toda su información pasa a
    CCAUnir1                                                    ▷  $\mathcal{O}(1)$ 
25:
    actualizo distancias
26:   para cada i in 1 to n hacer                                ▷  $\mathcal{O}(n)$ 
27:     si componentesConexas[i] ya no representa una componente conexa, continuo                ▷  $\mathcal{O}(1)$ 
28:     distancia entre componentesConexas[i] y CCAUnir1 = min (distancia a CCAUnir1, distancia a
    CCAUnir2)                                                    ▷  $\mathcal{O}(1)$ 
29:     si CCAUnir2 esta mas cerca que CCAUnir1, aristaMasCortaHacia CCAUnir1 = aristaMasCortaHacia CCAUnir2
    O(1)                                                         ▷
30:     y lo mismo actualizo para la distancia y arista mas corta desde CCAUnir1 hacia la CC i    ▷  $\mathcal{O}(1)$ 
31:     voy guardando la distanciaMasCorta e indiceCCMasCerca de CCAUnir1                      ▷  $\mathcal{O}(1)$ 
32:     voy viendo si tengo que actualizar la distanciaMasCorta e indiceCCMasCerca de la CC i    ▷  $\mathcal{O}(1)$ 
33:   fin para
34: fin para
35: al final recorro componentesConexas fijandome qué índices representan componentes conexas, en estos
    índices de pueblo coloco una central y las tuberías son la unión de las las aristas de las CC representadas
    por estos índices                                          ▷  $\mathcal{O}(n)$ 
36:                                                         ▷ Total  $\mathcal{O}(n^2)$ 

```

3.2.3. Justificación formal de correctitud

[asegurarse de que el pseudocodigo este mas arriba que esto]

Veamos que nuestra implementación es una correcta versión del algoritmo de Kruskal. En cada iteración, Kruskal agrega a sus aristas la arista con menor peso entre las que no forman ciclo con las que ya tiene. Es decir, que una dos nodos que no estaban conectados por ningún camino, o lo que es lo mismo, que pertenezcan a distintas componentes conexas.

Veamos que nuestro algoritmo elige la misma arista que Kruskal. Iteramos sobre todas las componentes y elegimos las dos que tienen la menor distancia hacia otra componente. Ahora veamos que las distancias de una componente hacia otra están bien calculadas.

En la etapa de inicialización, cuando tenemos n componentes conexas triviales, la distancia entre cualquier par de ellas es la distancia euclídea entre sus únicos nodos. Ahora la distancia entre dos componentes conexas no triviales, es, afín a la noción de distancia en conjuntos, la distancia más corta entre un nodo de una componente conexa y un nodo de la otra. Supongamos que conocemos la distancia de la componente conexa A hacia la B y la C . Luego la distancia entre A y $B \cup C$ es la distancia entre un nodo de A y un nodo de B o C , es decir el mínimo de la distancia mínima entre un nodo de A y un nodo de B , y la distancia mínima entre un nodo de B y un nodo de C . Se concluye esta relación: distancia entre A y $B \cup C = \min(\text{distancia entre } A \text{ y } B, \text{ distancia entre } A \text{ y } C)$.

[aca poner algun titulo separador]

Queremos demostrar que, partiendo de un grafo inicial S_n , compuesto por n componentes triviales, el resultante de aplicar k iteraciones de Kruskal es una solución óptima. Para ello, aplicaremos inducción.

Aviso: Se cometerá un abuso de notación al indicar que se “suma/resta una arista a una solución”; lo que se está efectuando es realmente agregar o quitar la arista del conjunto de aristas de la solución.

Hipótesis Inductiva

$P(i)$: La solución S_{n-i} , obtenida a luego de aplicar i veces Kruskal, minimiza la **función objetivo** f (3.2.2, pág. 12) frente a cualquier otra solución compuesta por i componentes conexas.

Caso Inicial

$P(1)$: Es trivial, ya que cualquier grafo de n nodos que contiene $n - 1$ componentes conexas, contiene a lo sumo una sola arista. [No sé si es tan trivial eso..] Y ya que Kruskal elige en cada iteración la menor arista, denotémosla e_1 , el grafo $S_{n-1} = S_n + e_1$ resultante es mínimo.

Paso Inductivo

$P(i) \rightarrow P(i+1)$: Sea $S_{n-(i+1)}$ la solución obtenida en el paso $(i + 1)$ de **Kruskal**, podemos reescribir la misma de la forma $S_{n-(i+1)} = S_{n-i} + e_{i+1}$, en donde e_{i+1} es la arista agregada en este paso, y en donde S_{n-i} es una solución óptima según la **Hipótesis Inductiva**.

[Después sigo...]

3.2.4. Cota de complejidad temporal

La cota temporal según anotado en el pseudocódigo es $O(n^2)$. Todos los ciclos, excepto el último, son del estilo *for*, con la cantidad de iteraciones bien definida.

Faltaría nomás ver la cantidad de iteraciones que hace el ciclo que reconstruye de las componentes conexas, el grafo resultante con sus centrales y tuberías. Se recorre el arreglo de ComponentesConexas de tamaño n . Para cada componente conexa representada por una posición del arreglo, voy agregando las aristas de la componente a las aristas del grafo. Son exactamente $n - k$ aristas. Luego entre todas las $n - k$ aristas que recorro, agrego $n - k$ aristas en $O(1)$. Luego la complejidad de este ciclo es $O(n)$.

3.2.5. Verificación mediante casos de prueba

A continuación presentamos distintas instancias que sirven para verificar que el programa funciona correctamente.

Según la distribución de los pueblos en el mapa, y la relación entre cantidad total de pueblos y la cantidad máxima de centrales, podemos separar el conjunto de soluciones en 2 grandes casos:

- Todas las tuberías tienen la misma longitud:
 - Caso $\# \text{pueblos} \leq \# \text{centrales}$:
En estos casos se coloca en todos los pueblos una central y se va a tener un riesgo mínimo porque no hay tuberías (longitud de tuberías: 0).

Input		Output	
3	4	3	0
1	1	1	
2	2	2	
3	3	3	

- Caso $\# \text{pueblos} > \# \text{centrales}$:
Todas las tuberías tienen longitud 1 en este ejemplo.

Input		Output	
6	2	2	4
1	1	1	
2	1	4	
1	2	1	2
3	3	3	1
3	2	4	5
4	2	6	5

- Las tuberías tienen longitudes diferentes:

- Caso $\# \text{pueblos} > \# \text{centrales}$:
Longitudes de las tuberías: 0, 1, 2.

Input		Output	
6	2	2	4
1	1	1	
1	2	3	
2	5	1	2
3	1	4	1
3	3	4	6
4	1	5	4

Misma distribución de los pueblos pero sólo teniendo una central:

Longitudes de las tuberías: 1, 2, $\sqrt{5}$.

Input		Output	
6	1	1	5
1	1	1	
1	2	1	2
2	5	4	1
3	1	4	6
3	3	5	4
4	1	3	5

Ejecutamos el programa con los distintos ejemplos y se llegó a la solución esperada. Por lo tanto, podemos concluir que el comportamiento del programa es correcto.

3.2.6. Medición empírica de la Performance

3.3. Problema 3: Saltos en La Matrix

3.3.1. Descripción

3.3.2. Planteamiento de resolución

Para una mayor claridad, vamos a describir nuestro algoritmo reduciendo el problema a encontrar la distancia hasta la casilla destino. Encontrar la secuencia de saltos es algo secundario y está detallado en el código fuente.

Vamos a pensar el problema como un grafo, siendo cada nodo una posición del tablero con una cantidad de unidades de potencia extra restantes. Los adyacentes a cada nodo son las casillas (y las unidades extra que quedan para cada caso) a las que puedo llegar usando mi resorte y mis unidades extra.

Se implementó un Breadth-first search.

Recorremos primero los nodos a distancia 0, luego a distancia 1, y así sucesivamente. Para cada nodo vamos guardando su distancia desde la posición de origen.

Inicializamos la distancia hasta la posición de origen, contando con k unidades extra de potencia, con 0, y el resto de las distancias en INF.

Para cada nodo ' v ' que recorremos, sabemos que podemos llegar a sus adyacentes saltando hasta v en $v.\text{distancia}$ pasos, y luego saltando al adyacente ' a ' en un paso más.

Luego, podemos decir que $a.\text{distancia} \leq v.\text{distancia} + 1$. Si no había recorrido a previamente, $v.\text{distancia} + 1$ refleja la distancia del camino más corto hasta a . Si ya había recorrido ' a ', su distancia ya está calculada y es menor o igual a la de v . En algún momento llegamos a la casilla destino, ya que el grafo es conexo, y podemos devolver su distancia.

Pseudocódigo

Algoritmo 3 La Centralita

```

Inicializo
1: para ( $i = 1..n, j = 1..n, l = 0..k$ ) hacer  $\triangleright \mathcal{O}(n^2 * k)$ 
2:   distancia desde el casillero[i][j], sobrando l unidades extra de potencia = INF
3: fin para
4:
5: distancia hasta el casillero origen, sobrando k unidades extra de potencia = 0
6:
7: cola < (int, int, int) > colaBFS
8: colaBFS.push(origen, k)
9:
10: mientras (colaBFS no esté vacía) hacer  $\triangleright \mathcal{O}(n^3 * k)$ 
11:   actual = colaBFS.pop()
12:   para cada casillero (x,y) al que puedo llegar desde actual hacer
13:     l = unidades extra que quedan tras ir a (x,y)
14:
15:     si no recorri ya ese casillero, quedando esas unidades extra
16:     si distancia a (x, y), quedando l unidades extra es menor a INF entonces
17:       la pongo en 'distancia desde actual' + 1
18:       si es el casillero de destino entonces
19:         break
20:       fin si
21:       colaBFS.push((x,y), unidades extra que quedan)
22:   fin para
23: fin mientras
24:
25: retornar distancia al destino

```

3.3.3. Justificación formal de correctitud**3.3.4. Cota de complejidad temporal**

Como se pudo observar en el pseudocódigo (sección Planteamiento de Resolución), inicializar las distancias lleva $\mathcal{O}(n^2 * k)$.

En el ciclo **mientras** recorreremos a lo sumo $n^2 * k$ nodos, ya que no recorreremos dos veces el mismo nodo.

Para cada nodo miramos todos sus adyacentes, que son a lo sumo todas las casillas en la misma fila, o todas las casillas en la misma columna, cada casilla con una cantidad de unidades de potencia extra única.

En peor caso miramos $2 * n$ nodos, es decir $\mathcal{O}(n)$ nodos.

Luego la complejidad del ciclo es $\mathcal{O}(n^2 * k) * \mathcal{O}(n) = \mathcal{O}(n^3 * k)$.

3.3.5. Verificación mediante casos de prueba

A continuación presentamos distintas instancias que sirven para verificar que el programa funciona correctamente.

- Caso celda origen = celda destino

Input	Output
2 1 1 1 1 0	0
1 1	
1 1	

- Caso celda origen \neq celda destino

- Potencia extra = 0

- Potencia máxima del resorte igual para todas las celdas:

Input	Output
3 1 1 3 3 0	4
1 1 1	2 1 0
1 1 1	3 1 0
1 1 1	3 2 0
	3 3 0

Input	Output
3 1 1 3 3 0	2
5 5 5	3 1 0
5 5 5	3 3 0
5 5 5	

- Potencia máxima del resorte distinta para las celdas:

Input	Output
3 1 1 3 3 0	3
1 1 2	1 2 0
1 3 1	1 3 0
1 1 1	3 3 0

Input	Output
4 1 1 4 4 0	3
1 1 3 1	2 1 0
3 1 1 2	2 4 0
1 1 1 1	4 4 0
2 1 1 1	

- Potencia extra \neq 0

- Potencia máxima del resorte igual para todas las celdas:

Input	Output
3 1 1 3 3 5	2
1 1 1	3 1 1
1 1 1	3 3 1
1 1 1	

- Potencia máxima del resorte distinta para las celdas:

Input						Output		
3	1	1	3	3	1	2		
1	1	2				1	3	1
1	3	1				3	3	0
1	1	1						

Input						Output		
4	1	1	4	4	3	2		
1	1	3	1			4	1	2
3	1	1	2			4	4	1
1	1	1	1					
2	1	1	1					

Ejecutamos el programa con los distintos ejemplos y se llegó a la solución esperada. Por lo tanto, podemos concluir que el comportamiento del programa es correcto.

3.3.6. Medición empírica de la Performance

4. Apéndices

4.1. Código Fuente (resumen)