Trabajo Práctico 2

Técnicas Algorítmicas Avanzadas

Viernes 9 de Mayo de 2014

Algoritmos y Estructuras de Datos III
Entrega de TP

Grupo

$(-1111111111111111111111111111111111111111111111111111111111111001_2)$

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`

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

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 de **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 pretenciones 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].

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

[2]Una explicación coloquial, obviando detalles puramente implementativos: arquitectura, lenguaje,

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.*

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.

etc.

[3]En 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

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
- LaTeX 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 óptina durante cadu 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 (c1, c2, ..., cn) 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,..,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 (p1: Ptos. Jug1, p2: 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

Algunas soluciones posibles para este ejemplo (la primera es una óptima):

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

Output		
1	7	0
izq	5	

Output		
3	2	5
izq	2	
der	2	
izq	2	

3.1.2. Planteamiento de resolución

Veamos las ideas desarrolladas en nuestra resolución. En un juego cualquiera de Robanúmeros se cumple que, siendo P el puntaje:

$$P_{mio} + P_{oponente} = \sum_{c \in CARTAS} c$$

Es decir que la suma de mi puntaje y el de mi oponente equivale a la suma total de cartas del juego.

Luego, maximizar la diferencia entre mi puntaje y el del oponente es lo mismo que maximizar mi puntaje. Si agudizamos el análisis, nos damos cuenta de que el puntaje que yo saco con ciertas cartas sobre la mesa es igual a la suma de las cartas sobre la mesa menos el puntaje que saca el oponente con las cartas que dejo sobre la mesa tras mi jugada. Es decir,

$$P_{mio}(CARTAS) = suma(CARTAS) - P_{oponente}(\text{cartas que quedan})$$

Como suma(CARTAS) está fijo, queremos minimizar el puntaje del oponente con las cartas que le quedan. Como la cantidad de subconjuntos de CARTAS que le puedo dejar es finita (es $O(n)$), los podemos recorrer y quedarnos con el que haga que el oponente saque la mínima cantidad de puntos.

La cantidad de puntos que saca el oponente con las cartas que le dejo debe calcularse con la misma función con que yo calculo mi puntaje, ya que asumimos que el oponente juega de manera óptima, es decir, tan bien como yo.

Luego obtenemos:

Proposición 3.1.2.1. *siendo `MPP` el Máximo Puntaje Posible, θ los subconjuntos de CARTAS que pueden quedar en la mesa tras una jugada:*

$$\texttt{MPP}(CARTAS) = Suma(CARTAS) - \min_{CARTAS' \in \theta} \texttt{MPP}(CARTAS')$$

Como caso base, $MaxPuntajePosible(\text{solo la carta c sobre la mesa}) = valor(c)$, pues la única jugada posible es tomar la carta.

Pseudocódigo

Algoritmo 1 Roba Cartas

Entrada:

$cantCartas \leftarrow$ DAMECANTCARTAS ▷ *integer*
$cartas \leftarrow$ DAMEARREGLOCARTAS ▷ *arreglo(integer)*

Salida:

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

Estructura Levante:
dirección ▷ *Bool*
cantidad ▷ *integer*

Estructura Jugada:
$mejorPuntaje$ ▷ *Integer*
$turnosHastaAhora$ ▷ *Integer*
levanteRealizado ▷ *Levante*

Variables Globales:
$matrizJugadas$ ▷ $Matriz < Jugada > tamaño : cantCartas + 1, cantCartas + 1$
$sumasParciales$ ▷ $Arreglo < Integer > tamaño : cantCartas + 1$

`Se generan las sumas parciales`

$sumasParciales_0 \leftarrow 0$ ▷ $\mathcal{O}(1)$
para cada i en $[0, cantCartas - 1]$ **hacer** ▷ $\mathcal{O}(n)$
  $sumasParciales_{i+1} \leftarrow cartas_i + sumasParciales_i$ ▷ $\mathcal{O}(1)$
fin para

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

para cada *posición* **en** $matrizJugadas$ **hacer** ▷ $\mathcal{O}(n^2)$
  $matrizJugadas_{posición} \leftarrow 0$ ▷ $\mathcal{O}(1)$
fin para

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

para i **en** $[0, cantCartas - 1]$ **hacer** ▷ $\mathcal{O}(n)$
  $matrizJugadas_{i,i}.mejorPuntaje \leftarrow cartas_i$ ▷ $\mathcal{O}(1)$
  $matrizJugadas_{i,i}.turnosHastaAhora \leftarrow 1$ ▷ $\mathcal{O}(1)$
  $matrizJugadas_{i,i}.levanteRealizado.dirección \leftarrow$ TRUE ▷ $\mathcal{O}(1)$
  $matrizJugadas_{i,i}.levanteRealizado.cantidad \leftarrow 1$ ▷ $\mathcal{O}(1)$
fin para

```
Se rellena el resto de la matriz
para tamSubConj en [2, cantCartas] hacer                                          ▷ O(n^3)
    principio ← 0                                                                 ▷ O(1)
    final ← tamSubConj                                                            ▷ O(1)
  Se miran todos los subjuegos posibles de cada tamaño.
    mientras final <= cantCartas hacer                                            ▷ O(n^2)
        sumaParcial ← sumasParciales_final − sumasParciales_principio             ▷ O(1)
        peorJugada                                                                ▷ Jugada, O(1)
        peorJugada.mejorPuntaje ← infinito                                        ▷ O(1)
        levanteCorrecto                                                           ▷ Levante, O(1)
        para subjuego en [subjuego posibles] hacer                                ▷ O(n)
            si matrizJugadas_{PRINCIPIO(subjuego),FINAL(subconunto)} < peorJugada entonces   ▷ O(1)
                peorJugada ← matrizJugadas_{PRINCIPIO(subjuego),FINAL(subjuego)}  ▷ O(1)
                levanteCorrecto ← LEVANTEPARALLEGARA(subjuego)                    ▷ O(1)
            fin si
        fin para
        nuevaJugada                                                               ▷ O(1)
        nuevaJugada.mejorPuntajePosible ← sumaParcial − peorJugada.mejorPuntaje   ▷ O(1)
        nuevaJugada.turnosHastaAhora ← peorJugada.turnosHastaAhora + 1            ▷ O(1)
        nuevaJugada.levanteRealizado ← levanteCorrecto                            ▷ O(1)
        matrizJugadas_{principio,final−1} ← nuevaJugada                           ▷ O(1)
        principio + +                                                             ▷ O(1)
        final + +                                                                 ▷ O(1)
    fin mientras
fin para

La mejor jugada del juego total está en la posición 0, cantCartas-1
mejorPuntaje ← matrizJugadas_{0,cantCartas−1}                                     ▷ O(1)
puntajeEnemigo ← sumasParciales_cantCartas − mejorPuntaje                         ▷ O(1)

Ahora se revisan las jugadas realizadas
fin ← cantCartas − 1                                                              ▷ O(1)
init ← 0                                                                          ▷ O(1)
turnos ← 0                                                                        ▷ O(1)
levantes ← NUEVALISTA()                                                           ▷ Lista < Levante > O(1)
mientras init <= fin hacer                                                        ▷ O(n)
    levanteActual ← matrizJugadas_{init,fin}.levanteRealizado                     ▷ O(1)
    AGREGAR(levantes, levanteActual)                                              ▷ O(1)
    si levanteActual.dirección = IZQ entonces                                     ▷ O(1)
        fin ← fin − levanteActual.cantidad                                        ▷ O(1)
    sino
        init ← init + levanteActual.cantidad                                      ▷ O(1)
    fin si
    turnos + +;                                                                   ▷ O(1)
fin mientras
retornar mejorPuntaje                                                             ▷ O(1)
retornar puntajeEnemigo                                                           ▷ O(1)
retornar turnos                                                                   ▷ O(1)
retornar levantes                                                                 ▷ O(1)
```

3.1.3. Justificación formal de correctitud

Se procede a demostrar que nustro algoritmo es correcto. Vamos a concentrarnos en probar que devuelve un puntaje máximo correcto, que obtenemos de $matrizJugadas[1][n]$.

Premisa inductiva

P(k): En la iteración k del ciclo principal se cumple

$$\forall (i,j) : j - i \leq k, \text{ sea C = cartas que venían en el orden de i a j}$$
$$matrizJugadas[i][j] = MaxPuntajePosible(C)$$

Hagamos inducción en k, necesitamos probar P(n).

Nos vamos a apoyar en la `Proposición 3.1.2.1` (página 8)

Caso base

P(1): Se corresponde al caso base de la proposición. $matrizJugadas[i][i]$ vale el valor de la carta i, según la operación en *Algoritmo 1, línea 9*

Paso inductivo

Supongo que vale P(k - 1), y quiero ver que vale P(k) $\forall\ 2 \leq k \leq n$.

Ahora, los subconjuntos de las cartas que venian en el orden de i a j que le puedo dejar al oponente son:

- Ninguna carta[4]
- Las cartas de «i a $j-1$», de «i a $j-2$», «...», de «i a i» [5]
- Las cartas de «$i+1$ a j», de «$i+2$ a j», «...», de «j a j» [6]

Ahora, todos estos subconjuntos de cartas cumplen que:
`su extremo derecho - su extremo izquierdo` $\leq k$, luego por HI su $puntajeMaximo$ ya está calculado en $matrizJugadas[extremoizquierdo][extremoderecho]$.

Nuestro algoritmo recorre la matriz en todas esas posiciones, quedándonse con el puntaje más bajo (el que saca el oponente), y guardandolo en $matrizJugadas[i][j]$. Luego por el resultado 1.1 contiene $MaxPuntajePosible$(cartas de i a j). En la «k-esima» iteración del ciclo principal, lo hace para todos los (i,j) tales que $i - j = k$. Luego se cumple P(k).

[4]Este caso está representado en $matrizJugadas[x][y]$ con $y < x$, inicializado al comienzo del algoritmo con 0, que representa el puntaje máximo que se puede sacar sin cartas sobre la mesa.

[5]Corresponde a sacar cartas del extremo derecho.

[6]Corresponde a sacar cartas del extremo izquierdo.

3.1.4. Cota de complejidad temporal

Todas las operaciones y ciclos del pseudocódigo están debidamente anotados. Faltaría justificar [FIXME] que el ciclo que comienza en la línea 22, "para subjuego en subjuegosPosibles" es verdaderamente O(n). El resto de los ciclos están explícitamente acotados por n, la cantidad de cartas. La cantidad de subjuegos, es decir subconjuntos de cartas, que le podemos dejar al oponente, son siempre cartas contiguas. Es decir las cartas que vinieron en el orden de i a j, con $1 \leq i \leq j \leq n$. Ésto es siempre menor o igual a $2 * n$, que es O(n).

3.1.5. Verificación mediante casos de prueba

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

Input

n	c_1	...	c_n

Output

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

[FIXME]

3.2. Problema 2: La central "ITA" (de gas)

(AKA: La central...ITA)

3.2.1. Descripción

Planteo del Problema

Existe una región del país (Italia) 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 <u>tuberías interconectadas</u> 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 <u>centralitas</u>**, 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 "**<u>riesgo</u>**" de una determinada distribución de tuberías a la **máxima de sus longitudes**.

El codicioso *Fontanero* Jefe, **Mario Toti Segale**[7], 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**.

[7]"No es más que un italojaponés americano... gordo y bigotudo" - descripción anónima de un empleado ofuscado.

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	Formato de salida
n k x_1 y_1 . . . x_n y_n	q m p_1 ... p_q v_1 w_1 . . . v_m w_m

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*[8] 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:

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

Nota. 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**, $p : E \to \mathbb{R}$, 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:

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

Función objetivo

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

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

[8] «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.

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".

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** $f(s_k) <= f(s_x)$, 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 1 Kruskal

ComponenteConexa componentesConexas[n]

estructura ComponenteConexa:
  float distancias[n]
  list< $arista$ > aristas
  arista aristaMasCortaHacia[n]
  float distanciaMasCorta
  int indiceCCMasCerca

para cada i de 1 a n **hacer** ▷ $\mathcal{O}(n^2)$
    componentesConexas[i] = i-esimo pueblo ▷ $\mathcal{O}(1)$
    para cada j de 1 a n **hacer** ▷ $\mathcal{O}(n)$
        aristaMasCorta entre la CC i y la CC j = (i, j) ▷ $\mathcal{O}(1)$
        distancia entre componentesConexas[i] y la componente conexa j = distanciaEuclídea(pueblo i, pueblo j) ▷ $\mathcal{O}(1)$
        me voy fijando cual de estas distancias es mas corta y la guardo junto con el indice j ▷ $\mathcal{O}(1)$
    fin para
fin para

para cada i de 1 a n - k **hacer** ▷ $\mathcal{O}\big((n*(n-k)) = \mathcal{O}\big((n^2)\big)\big)$
    `// me fijo la distancia mas corta entre dos componentes`
    para cada i in 1 to n **hacer** ▷ $\mathcal{O}(n)$
        si componentes[i] ya no representa más una componente `continúo`
        me voy fijando qué componente tiene la menor 'menor distancia hacia otra componente'y la guardo en CCAUnir1, su componente mas cercana en CCAUnir2 ▷ $\mathcal{O}(1)$
    fin para

    `// uno CCAUnir1 y CCAUnir2`
    CCAUnir1.aristas = CCAUnir1.aristas ∪ CCAUnir2.aristas + aristaMasCorta entre CCAUnir1 y CCAUnir2 ▷ $\mathcal{O}(1)$
    marco CCAUnir2 como que ya no representa una componente conexa. (toda su información pasa a CCAUnir1) ▷ $\mathcal{O}(1)$

    `// actualizo distancias`
    para cada i de 1 a n **hacer** ▷ $\mathcal{O}(n)$
        si componentesConexas[i] ya no representa una componente conexa, `continúo` ▷ $\mathcal{O}(1)$
        distancia entre componentesConexas[i] y CCAUnir1 = min (distancia a CCAUnir1, distancia a CCAUnir2) ▷ $\mathcal{O}(1)$
        si CCAUnir2 esta mas cerca que CCAUnir1, aristaMasCortaHacia CCAUnir1 = aristaMasCortaHacia CCAUnir2 ▷ $\mathcal{O}(1)$
        con los mismos valores actualizo la distancia y arista mas corta desde CCAUnir1 hacia la CC i ▷ $\mathcal{O}(1)$
        voy guardando la distanciaMasCorta e indiceCCMasCerca de CCAUnir1 ▷ $\mathcal{O}(1)$
        voy viendo si tengo que actualizar la distanciaMasCorta e indiceCCMasCerca de la CC i ▷ $\mathcal{O}(1)$
    fin para
fin para
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 ▷ $\mathcal{O}(n)$

20 ▷ *Total* $\mathcal{O}\big((n^2)\big)$

3.2.3. Justificación formal de correctitud

Para demostrar la correctitud dividiremos el proceso en dos subprocesos independientes. Por un lado, demostraremos que el algoritmo expuesto cumple con los parámetros de un "Algoritmo de Kruskal". Por otro lado, demostraremos que un "Algoritmo de Kruskal", mediante su invariante de ciclo, cumple con las condiciones de nuestro problema.

Veamos que cumple Kruskal

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, ésto depende de que las distancias de una componente hacia otra estén 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 componte 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 $B \cup C$ = mín(distancia entre A y B, distancia entre A y C).

Veamos que Kruskal soluciona nuestro problema

Nota. Se cometerá un abuso de notación al indicar que se *"suma/resta una arista a una solución"*; lo que se está realizando es realmente agregar o quitar la arista del conjunto de aristas de la solución. Se denotará S_n como *"el grafo solución de n componentes conexas"*.

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

PREMISA INDUCTIVA
«P(i)»

La solución $S_{n-i} = (V, E_{n-i})$, subgrafo generador de K_n, obtenida luego de aplicar i veces Kruskal, la cual tiene «$n-i$» componentes conexas, minimiza la **función objetivo** f (3.2.2, pág. 18) cuando se la contrasta contra cualquier otra solución compuesta por «$n-i$» o menos componentes conexas.

CASO BASE
«P(1)»

Resulta trivial, ya que cualquier grafo de n nodos que contiene «$n-1$» componentes conexas contiene a lo sumo una sola arista, ya que por absurdo: dado cualquier grafo que cumpla las condiciones anteriores, al intentar agregar una segunda arista resulta inevitable unir dos de las «$n-1$» componentes conexas restantes, ya que todas son trivialmente maximales[9] en su cantidad de aristas internas, lo cual implica que el grafo dejaría de tener «$n-1$» componentes conexas. Y ya que Kruskal elige en cada iteración la menor arista, denotémosla particularmente e_1[10], el grafo $S_{n-1} = S_n + e_1$ resultante es mínimo.

Lema 3.2.3.1. *Dada una solución «A» cualquiera, de "x" componentes conexas, existe una solución «A′» con "y" componentes conexas tal que se cumple "y > x", la cual además es subgrafo generador[11] de «A», y surge de realizar una o más operaciones de "sacar una arista" sobre el conjunto de aristas de «A».*
Entonces, dada f (3.2.2, pág. 18), se cumple

$$f(A) >= f(A')$$

[9]Dado que son componentes triviales, excepto una, la cual contiene exactamente dos nodos y una arista.

[10]Notación: e_i representa la arista agregada en la "i-iteración". Se la denota e_1 por ser la primera iteración.

PASO INDUCTIVO

«P(i) $\rightarrow$ P(i+1)»

Hipótesis Inductiva: Suponemos que la premisa inductiva es válida para i.

Sea $S_{n-(i+1)} = (V, E_{n-(i+1)})$ 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**.

Vamos a demostrar por absurdo. Para ello, asumimos que **no** se cumple `p(i+1)`. Decir que no se cumple la **Premisa Inductiva** para `(i+1)` es equivalente a asumir que en este paso existe S^* **una solución estrictamente mejor a la nuestra**. En particular, a efectos de negar la **Premisa Inductiva**, podemos afirmar que esta otra solución contiene a lo sumo $n-(i+1)$ componentes conexas.

Por otro lado, por lema 3.2.3.1 podemos afirmar también que en caso de existir una solución S^* de **a lo sumo** $n-(i+1)$ componentes conexas, existe otra $S^*_{n-(i+1)} \subseteq S^*$ la cual contiene **exactamente** $n-(i+1)$ componentes conexas, que es subgrafo generador y, en particular, es mejor o igual que S^* cuando se la valúa en $f.objetivo$; lo que implica por transitividad que también es una solución estrictamente mejor a la nuestra. Reduciremos, pues, el análisis, a esta última $S^*_{n-(i+1)}$, a la cual denotaremos simplemente $S^* = (V, E^*)$ haciendo un abuso de notación.

Dado que S^* es mejor solución, esto equivale a afirmar que siendo p la función peso se cumple $p(e_{i+1}) > p(e^*)$ para toda arista $e^* \in E^*$.

Ahora bien, S^* está formado por un conjunto V de nodos y un conjunto E^* de aristas, y es fácil ver que dado que el conjunto de nodos **es el mismo que el de** S_{n-i}, «*si alguna arista $e^* \in E^*$ conectara en S^* dos componentes que en S_{n-i} están disjuntas, entonces llegaría a un absurdo*», ya que por el párrafo anterior tendría que $p(e^*) < p(e_{i+1})$, pero esto es un escenario imposible, ya que se contradeciría con la **invariante de ciclo de Kruskal**, ya que al no formar e^* un ciclo en S_{n-i}, y al ser menor que e_{i+1}, **Kruskal** debería haberla elegido en su lugar, ya que en cada paso selecciona la arista de peso mínimo que no forme ciclos.

Quiero mostrar, pues, que es inevitable que esta última arista exista. Para ello me voy a valer de que S_{n-i} tiene exactamente $n-i$ componentes conexas, mientras que S^* tiene $n-(i+1)$, es decir, una menos. Se demostrará por absurdo en el párrafo siguiente.

Proposición 3.2.3.2. *Decir que* "no pueden existir en S^* aristas tales que en S_{n-i} conecten dos componentes distintas" *es equivalente a admitir que por cada componente conexa en S^* debe existir una en S_{n-i} de forma tal que la misma contenga a todos sus nodos.*

Demostracion 3.2.3.3. Esto es así ya que **de lo contrario**, si existiese una componente C^* en S^* tal que sus nodos no estuviesen contenidos en los nodos de alguna componente de S_{n-i}, existirían en particular dos conjuntos de nodos $C_A^* \subseteq C^*$ y $C_B^* \subseteq C^* - C_A^*$ distintos de vacío, para los que se cumple que C_A^* está contenido en alguna componente de S_{n-i}, y C_B^* está contenido en alguna otra componente (distinta) de S_{n-i}. Que existe C_A^* contenido en alguna componente de S_{n-i} es fácil de ver, ya que en particular un subgrafo de C^* formado por un nodo está contenido trivialmente en la componente que lo contiene en S_{n-i}. Ahora, si tomamos C_A^* un conjunto de nodos maximal[12] contenido en alguna componente de S_{n-i}, y SC_A a la componente que los contiene, esto implica que para todo nodo $v_B \in C_B^*$, $v_B \notin SC_A$. Ahora, dado que $C_A^*, C_B^* \subseteq C^*$ componente conexa, existe un camino desde todo nodo de C_A^* hacia todo nodo de C_B^*, y en particular existe una arista de frontera, es decir, una arista $e^* = (v_A^*, v_B^*)$ en donde un extremo pertenezca a un nodo $v_A^* \in C_A^*$ y el otro a un nodo $v_B^* \in C_B^*$.

Dado que los nodos de C_B^* no pertenecen a C_A^*, y siendo que C_A^* **es maximal**, pudimos afirmar que los nodos de C_B^* **no** pertenecen a la componente conexa SC_A o, lo que es igual, que pertenecen a otra componente conexa en S_{n-i}. En particular, entonces, v_B^* tampoco pertenece a SC_A. Finalmente, la arista e^* mencionada anteriormente estaría conectando dos componentes disjuntas si se la agregase a S_{n-i}, en donde específicamente una de ellas sería SC_A, y la otra sería la componente de S_{n-i} tal que contiene al nodo v_b^*.

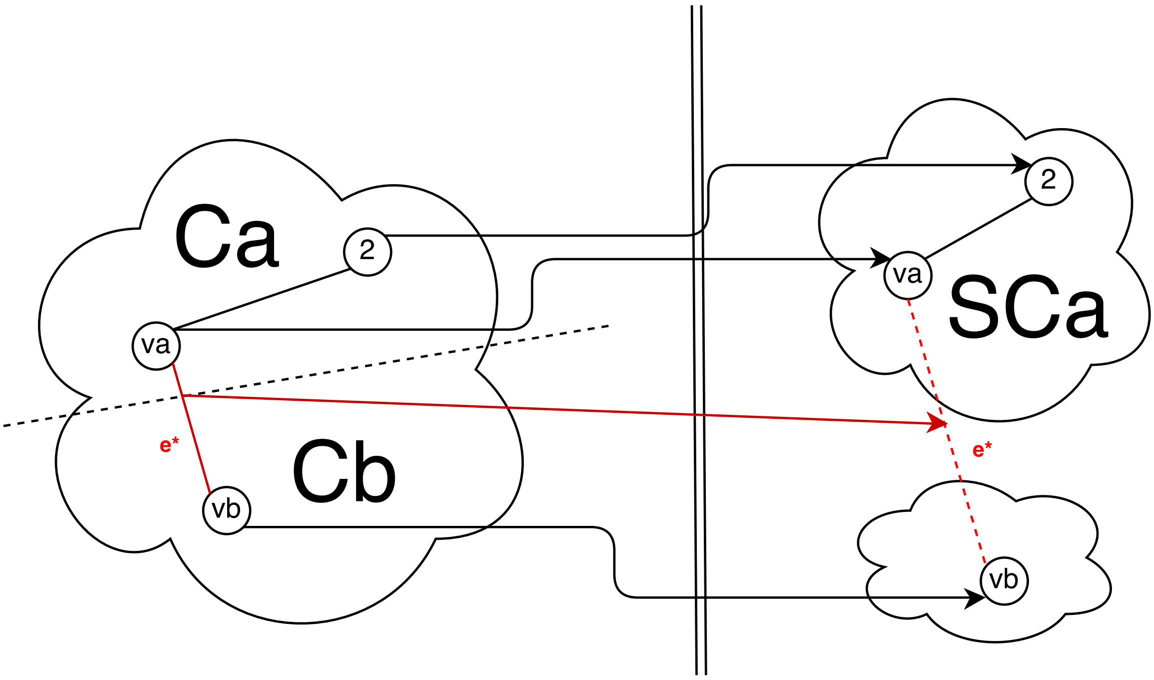

Figura 1: **Ejemplo visual de la demostración anterior**

[12]Es decir, un conjunto de al menos un nodo formado por nodos de C^* tal que esté contenido en S_{n-i} y que ningún otro nodo perteneciente a C^* le pueda ser agregado.

En el gráfico se puede apreciar el fenómeno expuesto en el párrafo anterior, en donde a modo de ejemplo se están obviando nodos y componentes que no son imprescindibles a la demostración (se asume que adentro de las "nubes", cada una de las cuales representa una componente conexa, hay una cantidad **indefinida** de nodos, no reflejada en el gráfico, y se asume que dentro de cada "nube" todos sus nodos son conexos).

Finalmente...

Dado que todas las soluciones de este problema tienen la misma cantidad de nodos, ya que son subgrafos generadores de K_n, ya que la cantidad de nodos contenidos en las $n-(i+1)$ componentes de S^* suman n, y teniendo en cuenta la proposición anterior, se deduce que las $n-(i+1)$ componentes de S^* están contenidas en **a lo sumo** $n-(i+1)$ componentes de S_{n-i} o, lo que es lo mismo, que existe cierto conjunto de **a lo sumo** $n-(i+1)$ componentes de S_{n-i} ***cuyos nodos suman n***, y puesto que existen $n-i$ componentes en S_{n-i} eso significaría que entre todas ellas sumarían como mínimo $n+[(n-i)-(n-(i+1))]=n+1$ nodos, lo cual es **ABSURDO**.

Entonces, ya que derivado de suponer que no se cumplía $p(i) \rightarrow p(i+1)$ llegamos a un absurdo, esto en conjunto con la validez del *caso base* implica que la premisa vale para todo i, en particular para el caso $i=n-k$ y, por todo lo ya expuesto, esto último demuestra que `Kruskal` es solución al problema.

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 claramente definida.

Faltaría entonces ver la cantidad de iteraciones que hace el ciclo que reconstruye, a partir 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. Recorro a lo sumo n componentes conexas, y entre todas agrego n - k aristas en O(1).
>
> Finalmente, 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 #pueblos $\leq$ #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	
3	4
1	1
2	2
3	3

Output	
3	0
1	
2	
3	

- Caso #pueblos > #centrales:
  Todas las tuberías tienen longitud 1 en este ejemplo.

Input	
6	2
1	1
2	1
1	2
3	3
3	2
4	2

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

Las tuberías tienen longitudes diferentes

- Caso #pueblos > #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

[FIXME]

(,.,)(é.è)(,.,)

3.3. Problema 3: Saltos en La Matrix

3.3.1. Descripción

Se tiene un campo cuadrado de $n \times n$ celdas, cada una de la cuales tiene un resorte propulsor con una potencia que va de 1 a p. Es decir que es posible ir saltando de una celda cualquier otra mediante estos resortes. El juego consiste en que cada participante llegue a la celda *destino* partiendo desde la celda *origen*, realizando la menor cantidad de saltos posibles. Para moverse de una de una celda a otra, sólo se permiten movimientos hacia el norte, sur, este u oeste. La potencia que tenga el resorte en una celda, es la que limitará la distancia del salto que se puede realizar desde una celda a otra. Además, el participante contará con una cantidad k de pontencia extra, la cual podrá utilizar durante el juego, distribuyéndola como le sea conveniente en cada salto. Esta potencia extra, le permitirá realizar saltos de mayor distancia. La complejidad temporal de peor caso del algoritmo deberá ser de $\mathcal{O}(n^3 \times k)$.

Formato de entrada y salida:

Input					
n	f_0	c_0	f_d	c_d	k
p_{11}	$\dots$	p_{1n}			
$\vdots$		$\vdots$			
p_{n1}		p_{nn}			

Output		
S		
f_1	c_1	e_1
$\vdots$	$\vdots$	$\vdots$
f_s	c_s	e_s

- n #*filas* y #*columnas*
- (f_0, c_0) celda *origen*
- (f_d, c_d) celda *destino*
- k #*unidades* de potencia extra
- p_{ij} potencia máxima del resorte de la celda (i,j), $1 < i, j < n$
- S #*saltos* de solución óptima
- (f_i, c_i) celda a la que se realiza el salto, $1 < i < S$
- e_i #*unidades* de potencia extra usados en saldo i, $1 < i < S$, $0 < e_i < k$

Ejemplo 3.3.1.1.

Input					
3	1	1	2	3	2
1	1	1			
1	2	1			
1	1	1			

La siguientes son algunas soluciones posibles (las 2 primeras serían las óptimas):

Output		
2		
2	1	0
2	3	1

Output		
2		
1	3	1
2	3	0

Output		
3		
1	2	0
1	3	0
2	3	0

Output		
3		
3	1	1
3	3	1
2	3	0

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

Figura 2: Moerfo, antes de saltar (...a la siguiente celda)

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 algoritmo utilizando la técnica de Breadth-First Search.

Pseudocódigo

Algoritmo 2 La Centralita

```
// Inicializo
para (i = 1..n, j = 1..n, l = 0..k) hacer                                   ▷ O(n^2 * k)
    distancia hasta el casillero[i][j], sobrando l unidades extra de potencia = ∞
fin para

distancia hasta el casillero origen, sobrando k unidades extra de potencia = 0

cola< (int, int, int) > colaBFS
colaBFS.push(origen, k)

mientras (colaBFS no esté vacía) hacer                                      ▷ O(n^3 * k)
    actual = colaBFS.pop()
    para cada casillero (x,y) al que puedo llegar desde actual hacer
        l = unidades extra que quedan tras ir a (x,y)

        // si no recorrí ya ese casillero, quedando esas unidades extra
        si distancia a (x, y), quedando l unidades extra es menor a ∞ entonces
            la sobreescribo como: 'distancia desde actual'+ 1
            si es el casillero de destino entonces
                break
            fin si
            colaBFS.push((x,y), unidades extra que quedan)
        fin si
    fin para
fin mientras

retornar distancia al destino
```

3.3.3. Justificación formal de correctitud

[no se puede asumir que el lector conozca bfs] En nuestro algoritmo recorremos primero los nodos a distancia 0, luego a distancia 1, y así sucesivamente. Para cada nodo vamos guardando su distancia desde la posicion de origen. Inicializamos la distancia hasta la posicion 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.distancia pasos, y luego saltando al adyacente $'a'$ en un paso más. Luego, podemos decir que a.distancia $\leq v$.distancia $+ 1$. Si no había recorrido a previamente, v.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.
[no es conexo, no podes llegar de k=3 a k=5]

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` recorremos a lo sumo $n^2 * k$ nodos, ya que no recorremos 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 ≠ 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 ≠ 0

- Potencia máxima del resorte igual para todas las celdas:

Input

3	1	1	3	3	5
1	1	1			
1	1	1			
1	1	1			

Output

2		
3	1	1
3	3	1

- Potencia máxima del resorte distinta para las celdas:

Input

3	1	1	3	3	1
1	1	2			
1	3	1			
1	1	1			

Output

2		
1	3	1
3	3	0

Input

4	1	1	4	4	3
1	1	3	1		
3	1	1	2		
1	1	1	1		
2	1	1	1		

Output

2		
4	1	2
4	4	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

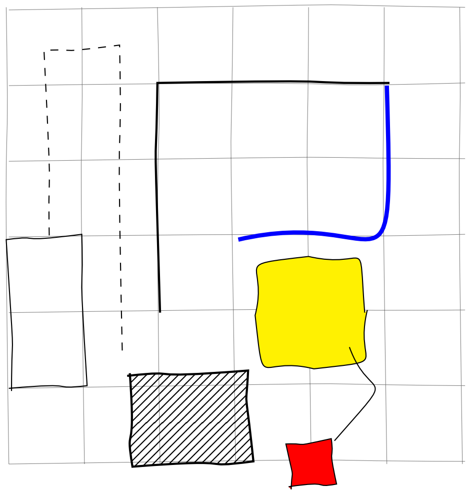

Figura 3: Tiempo en función del radio

[FIXME]

4. Apéndices

4.1. Código Fuente (resumen)