COMPUTACIÓN

2012 EDITION

DIVERTIMENTOS INFORMÁTICOS FOR NEWBIES

COMPUTER SCIENCE FOR THE REST OF US!

DESCUBRIRÁS LO QUE DONALD KNUTH NO QUIERE QUE SEPAS.

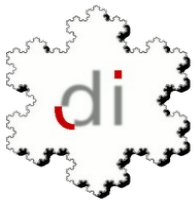
Alberto García Serrano

Divertimentos Informáticos

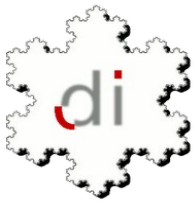
Informática recreativa para programadores inquietos

by

Alberto García Serrano <alberto.garcia AT agserrano.com>



Contenido



Introducción

Cuando comencé a escribir el blog Divertimentos informáticos (http://divertimentosinformaticos.blogspot.com.es/) lo hice por una necesidad propia de explorar y descubrir los entresijos tras la ciencia de la computación, y poco a poco se transformó en una aventura apasionante. Quizás buscaba volver a revivir la curiosidad que A.K. Dewdney me hizo sentir en mi juventud explorando sus "mundos del ordenador" en el libro "Aventuras Informáticas".

Mi intención no es, ni ha sido nunca, tratar de transmitir un conocimiento, que por otra lado, está recogido y bien documentado en multitud de libros. Más bien tratar de despertar la curiosidad y quitar hierro a temas de por sí bastante complejos. Lo que los ingleses llaman romper la superficie (break the surface).

Cada capítulo se corresponde con un post del blog, en el que se presenta un tema relacionado con la computación y el mundo de los ordenadores. Temas como Inteligencia Artificial, Criptografía, Algoritmia, etc. Los temas son tratados de forma accesible y lo más amena posible, en forma de diálogos entre personajes imaginarios.

Animado por la propuesta de algunos lectores de reunir los artículos en un sólo libro, me he decidido a crear este recopilatorio. Espero que lo disfrutes.

! Importante: El contenido de este libro está libremente disponible en http://divertimentosinformaticos.blogspot.com.es.

1. Razonando como un niño

Acababa de llegar a la oficina, y ahí estaba el señor Lego sentado en la mesa de al lado. A pesar de ser mi compañero de trabajo (es programador como yo) nos hablamos de usted y yo siempre lo llamo señor Lego (su nombre real es Pascal Lego). Estaba ensimismado tratando de resolver un rompecabezas para niños pequeños. Un rompecabezas lineal de unas cuatro piezas grandes.

¿Difícil señor Lego? - Dije irónicamente.

Mucho, me respondió. Anoche estaba enseñando a mi hijo como resolver operaciones matemáticas con el ordenador, y de pronto me espetó - "Este ordenador es tonto, ni siquiera es capaz de resolver mi rompecabezas" - Y aquí ando, viendo como podría hacer un programa para demostrar a mi hijo que el ordenador si puede resolverlo. Quizás tu puedas echarme una mano. La verdad es que tenía bastante trabajo acumulado, pero uno no es de piedra y siempre está dispuesto a enfrentarse a nuevos desafíos.

Esta bien, señor Lego. Primero hay que buscar una forma de representar el rompecabezas en el ordenador. Le propongo la siguiente representación: A cada pieza le asignaremos un número, y la representaremos en una lista o array de enteros. Consideraremos que dos piezas encajan si con consecutivas y están ordenadas de menor a mayor. Por ejemplo [1234] representaría el rompecabezas bien ordenado y en el que encajan todas las piezas. Si estuvieran así [2134] significaría que las dos primeras piezas no encajan (están bailadas) pero las dos últimas si encajan.

Sí, me parece bien -dijo el señor Lego con poca convicción.

Ahora tenemos que definir qué acciones vamos a poder realizar sobre sobre las piezas. Si le parece, podemos definir estas:

D - Intercambiar las dos piezas de la derecha.

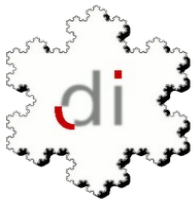
C - Intercambiar las dos piezas centrales.

I - Intercambiar las dos piezas de la izquierda.

De forma que si aplicamos la operación D a [2134] obtendríamos [1234].

Ya veo -dijo el señor Lego - entonces se trata de ir intercambiando piezas contiguas hasta conseguir que el rompecabezas esté ordenado.

Eso es -respondí. Ahora nos falta decidir qué estrategia seguimos para ir intercambiando las piezas. Si usted o yo quisiéramos ordenar las piezas

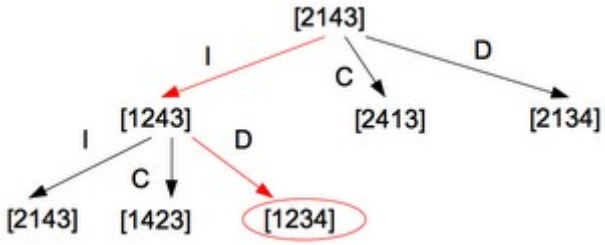


seguiríamos una serie de procesos mentales bastante complejos, pero el ordenador, afortunadamente, no es tan inteligente como nosotros. Si fueras tan tonto como un ordenador, ¿qué estrategia seguirías?

El señor Lego se quedó mirándome sin saber si yo esperaba realmente una respuesta o era una pregunta retórica que estaba a punto de contestar yo mismo.

Pues probaría cambios a ciegas hasta que finalmente obtuviera el rompecabezas ordenado.

Muy bien, señor Lego. Me parece una buena estrategia -contesté como si fuera un locutor de radio que acaba de dar un premio a un oyente. Para entendernos, en lo sucesivo, llamaremos *estado* a como queda el rompecabezas tras cada intercambio. Por lo tanto, partiremos de un *estado inicial*, que es como se encuentra el rompecabezas antes de empezar. Además iremos haciendo los cambios de forma sistemática para no dejarnos ninguno atrás. Lo mejor es verlo como un árbol donde se despliegan todas las posibilidades (estados). Cogí un bolígrafo e hice el siguiente esquema.



Fíjese señor Lego, He puesto arriba el estado inicial, y para generar los siguientes estados aplico las tres operaciones que hemos definido antes (D, C e I). Indico las operaciones aplicadas en cada caso en las flechitas para que quede más claro. Esto es lo que llamaremos *expandir* un estado. Si comprobamos que no hemos obtenido una ordenación correcta (a la que llamaremos *nodo solución*) repetimos el procedimiento expandiendo cada uno de los tres nuevos nodos que se han generado (a los que llamaremos *nodos hijos*).

En rojo he marcado un posible camino que nos ofrece una solución. Pero, puede haber varios caminos que lleven a la solución del rompecabezas ¿no? -dijo el señor Lego.

Efectivamente, pero por ahora, nos bastará con cualquiera de las soluciones.

El Sr. Lego me miró desafiante y dijo: ¡Voy a probar a implemetarlo! Mientras salía corriendo a sentarse en su ordenador.

Pasó un buen rato, y cuando suponía que ya lo había dejado por imposible y que se estaría dedicando a mirar páginas en Internet, se levantó con unos listados en Python y me los soltó encima del teclado.

Reproduzco aquí ambos listados.

Arbol.py

```
1.  class Hoja:
2.      def __init__(self, datos, hijos=None):
3.          self.datos = datos
4.          self.hijos = None
5.          self.padre = None
6.          self.set_hijos(hijos)
7.
8.      def __str__(self):
9.          return str(self.datos)
10.
11.     def set_hijos(self, hijos):
12.         self.hijos=hijos
13.         if self.hijos != None:
14.             for h in self.hijos:
15.                 h.padre = self
16.
17.     def get_hijos(self):
18.         return self.hijos
19.
20.     def set_datos(self, datos):
21.         self.datos = datos
22.
23.     def get_datos(self):
24.         return self.datos
```

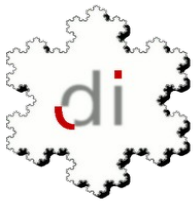
backtracking_i.py

```
1.  from arbol import Hoja
2.
3.  def buscar_solucion():
4.
5.      global solucionado
6.      global estado_inicial
7.      global solucion
8.      global hojas_expandidas
9.      global hojas_no_expandidas
10.     global nodo_solucion
11.
12.     while (not solucionado) and len(hojas_no_expandidas)!=0:
13.         nodo=hojas_no_expandidas[0]
14.         hojas_expandidas.append(hojas_no_expandidas.pop(0))
15.         if nodo.get_datos() == solucion:
16.             solucionado=True
17.             nodo_solucion = nodo
18.         else:
19.             # expandir nodos sucesores
20.             dato_nodo = nodo.get_datos()
```

```
21.
22.             # movimiento izquierdo
23.             hijo_izquierdo = Hoja([dato_nodo[1], dato_nodo[0],
    dato_nodo[2], dato_nodo[3]])
24.             hojas_no_expandidas.append(hijo_izquierdo)
25.
26.             # movimiento central
27.             hijo_central = Hoja([dato_nodo[0], dato_nodo[2], dato_nodo[1],
    dato_nodo[3]])
28.             hojas_no_expandidas.append(hijo_central)
29.
30.             # movimiento derecho
31.             hijo_derecho = Hoja([dato_nodo[0], dato_nodo[1], dato_nodo[3],
    dato_nodo[2]])
32.             hojas_no_expandidas.append(hijo_derecho)
33.
34.             nodo.set_hijos([hijo_izquierdo, hijo_central, hijo_derecho])
35.
36.
37.
38. if __name__ == "__main__":
39.     print "iterativa"
40.     estado_inicial=[4,2,3,1]
41.     solucion=[1,2,3,4]
42.     hojas_expandidas=[]
43.     hojas_no_expandidas=[]
44.     solucionado=False
45.     nodo_solucion = None
46.
47.     nodoInicial = Hoja(estado_inicial)
48.     hojas_no_expandidas.append(nodoInicial)
49.
50.     buscar_solucion()
51.
52.     # mostrar resultado (al reves)
53.     nodo=nodo_solucion
54.     while nodo.padre != None:
55.         print nodo
56.         nodo = nodo.padre
57.
58.     print nodo
```

Tras ejecutarlos, obtuvimos el siguiente resultado:

```
[1, 2, 3, 4]
[2, 1, 3, 4]
[2, 3, 1, 4]
[2, 3, 4, 1]
```



```
[2, 4, 3, 1]
[4, 2, 3, 1]
```

Mira como lo he resuelto -dijo el señor Lego con un entusiasmo al que no nos tenía acostumbrado. Es el resultado de la búsqueda por el árbol, pero al revés. Muestro así el resultado porque empiezo mostrando el nodo solución y recorro hacia atrás el árbol hasta el nodo raíz. Lo he dejado así para no complicar mucho el programa y que sea más fácil de seguir.

Increíble señor Lego, su programa funciona correctamente. Pero explíqueme como lo ha planteado -le dije mostrando interés.

Bien, lo que he hecho es crear una clase muy sencilla para el manejo de árboles y que luego utilizo en el programa principal.

Tenemos una lista llamada *hojas_no_expandidas* donde inicialmente almacenamos el estado inicial.

En su bucle principal, el programa realiza las siguientes acciones: A cada iteración del bucle *while* obtengo el primer nodo de *hojas_no_expandidas* y compruebo si ese nodo es un nodo solución. añadimos ese nodo a otra estructura (una lista) llamada *hojas_expandidas*, y evidentemente la retiro de *hojas_no_expandidas*.

Si el nodo es una solución, terminamos; si no, expandimos el nodo aplicando las tres operaciones (D, C, I) y añadimos los tres nodos nuevos a *hojas_no_expandidas*.

La verdad es que no esperaba que el señor Lego fuera capaz de resolver el

problema, aunque la verdad es que no terminaban de convencerme todas aquellas variables globales, aunque supongo que lo dejó así por simplicidad y para que yo pudiera entender mejor su código.
No está nada mal, señor Lego. Acaba de implementar una *búsqueda en un árbol*, y usted ha hecho una implementación iterativa -le dije.

¿Iterativa? ¿es que se puede implementar de otra forma? -dijo simulando sorpresa.

De hecho, señor Lego, los algoritmos de busqueda suelen implementarse usando recursividad -dije.

El señor Lego sabía lo que era la recursividad de sus años de carrera, pero no tenía muy claro cómo podría usarla en este problema, así que me senté ante el ordenador y tecleé el código siguiente reutilizando su implementación de árbol.

backtracking_r.py

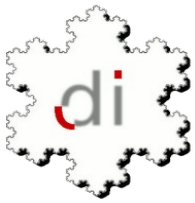
```
1.  from arbol import Hoja
2.
3.  def buscar_solucion(nodo_inicial, solucion, visitados):
4.      visitados.append(nodo_inicial.get_datos())
5.      if nodo_inicial.get_datos() == solucion:
6.          return nodo_inicial
7.      else:
8.          # expandir nodos sucesores (hijos)
9.          dato_nodo = nodo_inicial.get_datos()
10.         hijo_izquierdo = Hoja([dato_nodo[1], dato_nodo[0], dato_nodo[2],
    dato_nodo[3]])
11.         hijo_central = Hoja([dato_nodo[0], dato_nodo[2], dato_nodo[1],
    dato_nodo[3]])
12.         hijo_derecho = Hoja([dato_nodo[0], dato_nodo[1], dato_nodo[3],
    dato_nodo[2]])
13.         nodo_inicial.set_hijos([hijo_izquierdo, hijo_central,
    hijo_derecho])
```

```
14.
15.         for nodo_hijo in nodo_inicial.get_hijos():
16.             if not nodo_hijo.get_datos() in visitados:
17.                 sol = buscar_solucion(nodo_hijo, solucion, visitados)
18.                 if sol != None:
19.                     return sol
20.
21.         return None
22.
23.
24. if __name__ == "__main__":
25.     print "recursiva"
26.     estado_inicial=[4,2,3,1]
27.     solucion=[1,2,3,4]
28.     nodo_solucion = None
29.     visitados=[]
30.
31.     nodo_inicial = Hoja(estado_inicial)
32.
33.     nodo = buscar_solucion(nodo_inicial, solucion, visitados)
34.
35.     # mostrar resultado (al reves)
36.     while nodo.padre != None:
37.         print nodo
38.         nodo = nodo.padre
39.
40.     print nodo
```

Ahora -comencé a explicarle- la función *buscar_solucion* recibe tres parámetros: El nodo inicial, la solución que estamos buscando y una lista llamada *visitados* que contendrá aquellos nodos que vayamos visitando durante el recorrido del árbol.
Ya dentro de la función, si comprobamos que hemos llegado a una solución, saldremos devolviendo dicho nodo como resultado; si no, expandimos el nodo y volvemos a llamar recursivamente a la función *buscar_solucion* con cada uno de los nodos hijos generados. Si el nodo que estamos evaluando no es solución devolvemos *None* (valor nulo).

Ejecuté el programa y éste fue el resultado:

```
[1, 2, 3, 4]
[2, 1, 3, 4]
[2, 1, 4, 3]
[1, 2, 4, 3]
[1, 4, 2, 3]
[4, 1, 2, 3]
[4, 1, 3, 2]
[1, 4, 3, 2]
[1, 3, 4, 2]
[3, 1, 4, 2]
```



```
[3, 4, 1, 2]
[4, 3, 1, 2]
[4, 3, 2, 1]
[3, 4, 2, 1]
[3, 2, 4, 1]
[2, 3, 4, 1]
[2, 4, 3, 1]
[4, 2, 3, 1]
```

La sonrisa del señor Lego se hizo enorme al ver que su solución había resuelto el rompecabezas en menos movimientos.

Su solución no parece muy buena -dijo el señor Lego con desdén.

Tiene usted toda la razón. El algoritmo realiza algunos movimientos bastante tontos, por decirlo de alguna manera. Cuando resuelve usted el rompecabezas, hay ciertos movimientos que nunca haría ¿no es cierto? -pregunté.
Efectivamente, yo nunca haría el cambio [1, 4, 3, 2] ----> [4, 1, 3, 2]. No tiene sentido. Estamos empeorando la situación.

Es decir, señor Lego, que usted sabe, antes de hacer el cambio, que ese movimiento no le acerca más a la solución. Es un proceso que en Inteligencia Artificial se denomina Heurística y consiste precisamente en eso, en comprobar si el movimiento nos acerca más a la meta o no. De hecho, cuando el algoritmo de búsqueda encuentra un nodo que sabe que no nos llevará a un buen resultado no sigue explorando por ahí (se da la vuelta). A esta técnica se le denomina **Backtracking**.

Me senté y modifiqué el programa de la siguiente manera.

backtraking_r_h.py

```
1. from arbol import Hoja
2.
3. def buscar_solucion(nodo_inicial, solucion, visitados):
4.     visitados.append(nodo_inicial.get_datos())
5.     if nodo_inicial.get_datos() == solucion:
6.         return nodo_inicial
7.     else:
8.         # expandir nodos sucesores (hijos)
9.         dato_nodo = nodo_inicial.get_datos()
10.        hijo_izquierdo = Hoja([dato_nodo[1], dato_nodo[0], dato_nodo[2],
   dato_nodo[3]])
11.        hijo_central = Hoja([dato_nodo[0], dato_nodo[2], dato_nodo[1],
   dato_nodo[3]])
12.        hijo_derecho = Hoja([dato_nodo[0], dato_nodo[1], dato_nodo[3],
   dato_nodo[2]])
13.        nodo_inicial.set_hijos([hijo_izquierdo, hijo_central,
   hijo_derecho])
14.
15.        for nodo_hijo in nodo_inicial.get_hijos():
16.            if (not nodo_hijo.get_datos() in visitados) and
   heuristica(nodo_inicial, nodo_hijo):
17.                sol = buscar_solucion(nodo_hijo, solucion, visitados)
18.                if sol != None:
19.                    return sol
20.
21.        return None
22.
23.
24. def heuristica(nodo_padre, nodo_hijo):
25.     calidad_padre=0
26.     calidad_hijo=0
27.     dato_padre = nodo_padre.get_datos()
28.     dato_hijo = nodo_hijo.get_datos()
29.     for n in range(1,len(dato_padre)):
30.         if (dato_padre[n]>dato_padre[n-1]):
31.             calidad_padre = calidad_padre + 1;
32.         if (dato_hijo[n]>dato_hijo[n-1]):
33.             calidad_hijo = calidad_hijo + 1;
34.
35.     if calidad_hijo>=calidad_padre:
36.         return True
37.     else:
38.         return False
39.
40.
41. if __name__ == "__main__":
42.     print "recursiva con heuristica"
43.     estado_inicial=[4,2,3,1]
44.     solucion=[1,2,3,4]
45.     nodo_solucion = None
46.     visitados=[]
```

```
47.
48.     nodo_inicial = Hoja(estado_inicial)
49.
50.     nodo = buscar_solucion(nodo_inicial, solucion, visitados)
51.
52.     # mostrar resultado (al reves)
53.     while nodo.padre != None:
54.         print nodo
55.         nodo = nodo.padre
56.
57.     print nodo
```

¿Ve usted? he añadido una función llamada *heurística* que comprueba si el nodo hijo nos aleja de la solución en vez de acercarnos a ella. Lo que hacemos es mirar cuántas piezas contiguas hay en el nodo padre y cuántas en el hijo. Si el hijo tiene menos piezas contiguas que el padre, podremos descastar este movimiento. Esta técnica tiene algunos inconvenientes no muy evidentes a simple vista, pero en general es una buena aproximación. Pero ya hablaremos de eso en otro momento.

Tras ejecutar el código obtuvimos el siguiente resultado:

```
[1, 2, 3, 4]
[2, 1, 3, 4]
[2, 3, 1, 4]
[2, 3, 4, 1]
[2, 4, 3, 1]
[4, 2, 3, 1]
```

Mucho mejor -dijo el señor Lego ya no tan sonriente. Pero si hay varias posibles soluciones, ¿cómo podemos saber que ésta es la mejor de todas?

Muy buena pregunta señor Lego. De hecho no podemos saberlo, a no ser que recorramos todo el árbol.

Me senté de nuevo y me puse a teclear los siguientes cambios.

backtracking_r_h_m.py

```
1. from arbol import Hoja
2.
3. def buscar_solucion(nodo_inicial, solucion, visitados, nivel):
4.     global mejor_solucion
5.
6.     visitados.append(nodo_inicial.get_datos())
7.     if nodo_inicial.get_datos() == solucion:
8.         if not mejor_solucion:
9.             mejor_solucion=[nivel,nodo_inicial]
10.        else:
11.            if mejor_solucion[0]>nivel:
12.                mejor_solucion=[nivel,nodo_inicial]
13.    else:
14.        # expandir nodos sucesores (hijos)
15.        dato_nodo = nodo_inicial.get_datos()
16.        hijo_izquierdo = Hoja([dato_nodo[1], dato_nodo[0], dato_nodo[2],
   dato_nodo[3]])
17.        hijo_central = Hoja([dato_nodo[0], dato_nodo[2], dato_nodo[1],
   dato_nodo[3]])
18.        hijo_derecho = Hoja([dato_nodo[0], dato_nodo[1], dato_nodo[3],
   dato_nodo[2]])
19.        nodo_inicial.set_hijos([hijo_izquierdo, hijo_central,
   hijo_derecho])
20.
21.        for nodo_hijo in nodo_inicial.get_hijos():
22.            if (not nodo_hijo.get_datos() in visitados) and
   heuristica(nodo_inicial, nodo_hijo):
23.                buscar_solucion(nodo_hijo, solucion, visitados, nivel+1)
24.
25.
26.
27. def heuristica(nodo_padre, nodo_hijo):
28.     calidad_padre=0
29.     calidad_hijo=0
30.     dato_padre = nodo_padre.get_datos()
31.     dato_hijo = nodo_hijo.get_datos()
32.     for n in range(1,len(dato_padre)):
33.         if (dato_padre[n]>dato_padre[n-1]):
34.             calidad_padre = calidad_padre + 1;
35.         if (dato_hijo[n]>dato_hijo[n-1]):
36.             calidad_hijo = calidad_hijo + 1;
37.
38.     if calidad_hijo>=calidad_padre:
39.         return True
40.     else:
41.         return False
42.
43.
44. if __name__ == "__main__":
45.     print "recursiva con heuristica (mejor solucion)"
46.     estado_inicial=[4,2,3,1]
```

```
47.     solucion=[1,2,3,4]
48.     nodo_solucion = None
49.     visitados=[]
50.     mejor_solucion=[]
51.
52.     nodo_inicial = Hoja(estado_inicial)
53.
54.     buscar_solucion(nodo_inicial, solucion, visitados, 0)
55.
56.     # mostrar resultado (al reves)
57.     nodo=mejor_solucion[1]
58.     while nodo.padre != None:
59.         print nodo
60.         nodo = nodo.padre
61.
62.     print nodo
```

Fíjese en el cambio señor Lego, ahora, aunque encontremos una solución, no salimos de la función con *return*, sino que la almacenamos y seguimos buscando. La próxima vez que encuentre una solución, la comparará con la que tiene almacenada y si es mejor, la sustituirá. Así cuando termine el recorrido del árbol, tendremos almacenada la mejor solución de todas.

Pero, ¿cómo sabemos si una solución es mejor que otra? -preguntó el señor Lego.

Para nuestro caso hemos seguido un criterio sencillo. Fíjese que cada vez que llamamos recursivamente a la función *buscar_solucion* descendemos un nivel en el árbol, es decir, estamos a un salto más de distancia del nodo raíz. Para saber en qué nivel del árbol nos encontramos hemos añadido un nuevo parámetro a la función *buscar_solucion* llamado nivel. A cada llamada a la función le sumamos uno.
Por lo tanto, la mejor solución será aquella que esté más cerca del nodo raíz, porque es la que tiene menos movimientos. Es decir, la que tenga un nivel más bajo.

Ejecutamos el programa y obtuvimos:

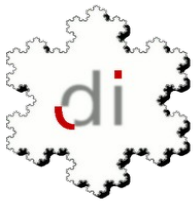
```
[1, 2, 3, 4]
[2, 1, 3, 4]
[2, 3, 1, 4]
[2, 3, 4, 1]
[2, 4, 3, 1]
[4, 2, 3, 1]
```

Fíjese que, en este caso, no se ha mejorado con respecto al algoritmo

anterior. Aunque en un árbol más grande seguramente sí que habría habido diferencias.

Entonces es mejor usar siempre esta última implementación ¿no? -Preguntó el señor Lego.

Para nada. Si el árbol fuera más grande el tiempo necesario para recorrerlo entero crece exponencialmente hasta valores de años, siglos o incluso milenios con los ordenadores actuales (y futuros). Es decir, son problemas computacionalmente intratables. Sólo puede usarse en árboles pequeños. Aunque si quiere, otro día hablaremos sobre estrategias para enfrentarse a ese tipo de problemas intratables. Por hoy ya está bien.



2. El juego de la vida

Aquellos puntitos negros no paran de encenderse y apagarse en la pantalla. Me quedo mirándolos durante un buen rato, casi hipnotizado. Se encienden y apagan formando curiosos patrones. Algunos patrones parecen repetirse indefinidamente y otros desaparecen tras un rato. También se van creando nuevas estructuras. Todo ese baile de puntitos me tiene intrigado.

¿Qué te parece, Alberto? -Me pregunta mi buen amigo Descollante. Es bonito, ¿pero qué es? -Pregunto. Se trata del Juego de la Vida -me responde Descollante.

Se trata de un autómata celular inventado por John Conway, un matemático de la Univesidad de Cambridge. Ayer estuve leyendo un antiguo artículo de Martin Gardner en Scientific American llamado "The fantastic combinations of John Conway's new solitaire game life". En él se describe un curioso juego de cero jugadores en el que una gran cantidad de células nacen y mueren según las condiciones de su entorno inmediato. En cada generación de células unas sobreviven, otras mueren y algunas nacen según unas simples reglas.

Ya veo -dije sin tener muy claro qué finalidad tenía todo aquello. ¿Y cuáles son esas reglas?

Son muy sencillas. En cada generación se evalúa cada célula:
Si una célula está viva en el instante t, morirá en el instante t+1 si menos de dos o más de tres de sus vecinas están vivas en el instante t.

Por el contrario, una célula cobrará vida en el instante t+1 si exactamente tres de sus ocho vecinas están vivas en el instante t.

Pues no parecen unas reglas muy complejas -le dije. Y veo que no has tardado en implementarlo.

Sí, estoy probando una serie de patrones que he encontrado por Internet y que son muy interesantes. Mira, acabo de construir una lanzadera de naves espaciales.

Me pareció algo bastante curioso. ¿Supongo que habrá más patrones interesantes? -pregunté

Claro, mira. Estos son otros que estoy probando.

Ver esos patrones hizo aumentar mi curiosidad, así que nos pusimos a probarlos todos. Así llevávamos un buen rato cuando empezó a hablarme de que el Juego de la Vida era algo más que un simple juego. Mi amigo

comenzó a contarme cómo se había demostrado que era posible crear un ordenador a partir de puertas lógicas NAND simuladas dentro de la rejilla del juego. De hecho, era posible construir una Máquina de Turing en su interior.

Bueno Alberto, me dijo mi amigo Descollante. Te reto a buscar nuevos patrones interesantes.

Como nunca he podido resistirme a un reto informático, en cuanto llegué a casa me puse a teclear un pequeño programa en Java para representar el juego. Os dejo aquí una versión simplificada que funciona en modo texto, a modo de referencia por si queréis aventuraros a implementar una versión en modo gráfico y con más funciones. En ella se muestra un ejemplo de patrón estable que a partir de cierta generación se mantiene indefinidamente cambiando entre dos estados.

```
1.  public class JuegoVida {
2.      static int[][] antes = new int[12][12];
3.      static int[][] despues = new int[12][12];
4.      public static void main(String[] args) {
5.          int i,j;
6.          String linea;
7.          antes[5][5]=1;
8.          antes[5][7]=1;
9.          antes[6][6]=1;
10.         antes[7][6]=1;
11.         antes[8][6]=1;
12.         while(true) {
13.             for (i=1 ; i<=10 ; i++) {
14.                 for (j=1 ; j<=10 ; j++) {
15.                     despues[i][j] = nuevoEstado(i,j);
16.                 }
17.             }
18.             // mostrar celulas
19.             for (i=1 ; i<=10 ; i++) {
20.                 linea="";
21.                 for (j=1 ; j<=10 ; j++) {
22.                     if (antes[i][j] == 1)
23.                         linea += "o";
24.                     else
```

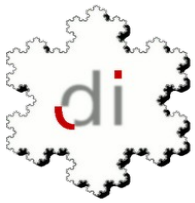
```
                        linea += ".";
                    }
                    System.out.println(linea);
                }
                // copiar celulas
                for (i=1 ; i<=10 ; i++) {
                    for (j=1 ; j<=10 ; j++) {
                        antes[i][j] = despues[i][j];
                    }
                }
            System.out.println("--------------------");
            try {
                Thread.sleep(500);
            } catch (Exception e) {}
            }
    }
    private static int nuevoEstado(int i, int j) {
        int vecinos;
        vecinos = antes[i-1][j-1]+antes[i][j-1]+antes[i+1][j-1];
        vecinos += antes[i-1][j]+antes[i+1][j];
        vecinos += antes[i-1][j+1]+antes[i][j+1]+antes[i+1][j+1];
        if ((vecinos == 3) && (antes[i][j] == 0))
            return 1;
        else if (((vecinos == 2) || (vecinos == 3)) && (antes[i][j]==1))
            return 1;
        else
            return 0;
    }
}
```

3. Caos infinito

Como en cualquier empresa en la que la mayoría de trabajadores son mileuristas, la hora de la comida es un ir y venir de tupperwares que se desplazan por la sala de descanso en un acompasado paseo que va del microondas a la mesa. Mi compañera Sofía traía una suerte de verdura extraña cocida, que la verdad, no entraba demasiado por los ojos. No pude más que preguntarle qué era eso que se disponía a comer. Es un romanescu. Un híbrido entre coliflor y brécol. ¿a que mola? - dijo alegremente.

Desde luego tenía una forma curiosa. Antes de atreverme a preguntarle lo que llevaba a una persona más o menos normal a engullir tan extraño alimento, Sofía se adelantó y empezó a explicarme que le encantaba esa verdura porque le recordaba a un fractal.

Uno de mis mayores problemas es que soy incapaz de dejar de demostrar con el gesto de mi cara todo aquello que pasa por mi cabeza, así que en aquél momento, Sofía podría haber sido capaz intuir mi debate interno para dirimir quién era más raro, el romenescu o ella. Afortunadamente, Sofía



malinterpretó mi cara y pensó que eso de los fractales debía sonarme a chino, cosa que por otro lado era cierta.

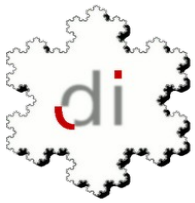
Un fractal es, por definición, un conjunto cuya dimensión de Hausdorff-Besicovitch es estrictamente mayor que su dimensión topológica - Espetó Sofía, así, sin anestesia.

De nuevo, mi cara debió ser un poema, porque empezó a reir y comenzó a hablarme de que los objetos fractales no tenían dimensión entera como una recta (1 dimensión), un plano (2 dimensiones) o un cubo (3 dimensiones). Sino que tienen dimensiones fraccionarias, como la curva de Koch que tiene dimensión 1'2618.

En ese momento ni me atrevía a mirar a Sofía a la cara. Me estaba dando un discurso que era incapaz de descifrar.

Imagina que queremos medir la longitud de la Costa del Sol, en el Sur de España- continuó Sofía. Podrías coger Google Maps e ir trazando líneas de 1Km por el perímetro de la costa y luego sumar el número de líneas. Esto que me estaba contando ahora sí que podía seguirlo, así que me animé a decirle que sí, pero que no sería una medida muy exacta. Efectivamente. Hay irregularidades más pequeñas de 1Km que no podríamos medir, pero supongamos que ahora lo hacemos con una regla de 1m. Pues ahora si que sería más exacto- contesté. Pero aún así hay irregularidades más pequeñas que 1m.

Correcto- dijo Sofía con voz cantarina. Y si bajamos a la escala del Centímetro pasará igual, y así sucesivamente hasta... bueno, hasta el infinito, si es que algo puede ser infinitamente pequeño. Esas



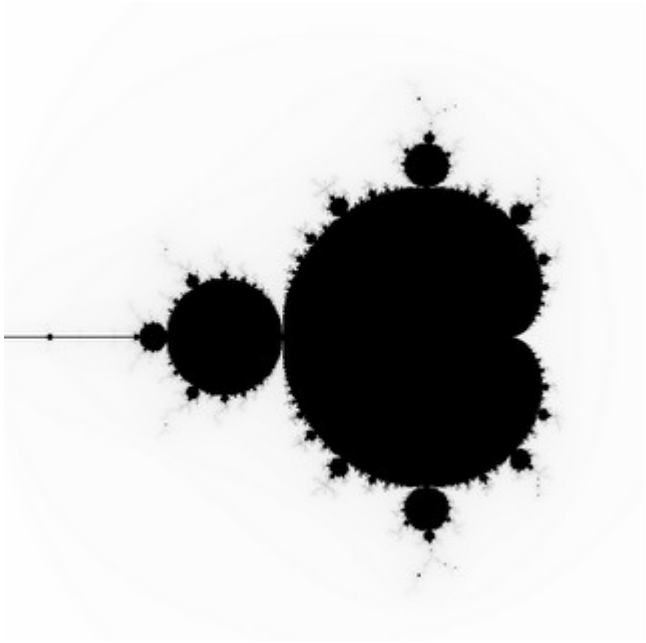
irregularidades nos impiden medir certeramente la costa usando las tres dimensiones del espacio euclideo, y es por ello por lo que tenemos que recurrir a la geometría fractal para describir multitud de elementos en la naturaleza.

Tal y como le pasa a este romenescu- Prosiguió. Si te fijas, tiene unas estructuras en forma helicoidal que se van repitiendo a escalas más pequeñas. Es como si un tallo de romenescu contuviera a si vez más tallos iguales, pero más pequeños, y estos a su vez tuvieran otros más pequeños, y así sucesivamente, sin importar la escala a la que fuéramos capaces de observar. Es lo que se llama autosimilitud.

La idea de algo que se repetía infinitamente según disminuíamos de escala me daba un poco de vértigo, pero a la vez llamaba mi atención.

Apenas si pude terminar de comerme mis spaguetis cuando ya había pasado la hora de la comida. Ya de vuelta en las mesas Sofía continuó con la disertación (Sofía se sienta junto a mí, justo en el lado contrario de donde se sienta el Sr. Lego).

El primero en hablar de fractales fué Benoît Mandelbrot- continuó. En 1975 acuñó el término fractal y creó el famoso conjunto de Mandelbrot.
Sofía giró su monitor y me mostró la siguiente imagen.



La imagen me resultó curiosa. Así que me me animé a preguntar si ese objeto fractal podría aumentarse e ir encontrando figuras similares a ella misma.

¡Claro!- respondió. Se puede explorar el conjunto de Mandelbrot y encontrarás paisajes muy hermosos, lugares inhóspitos y, por supuesto, réplicas en miniatura del conjunto de Mandelbrot.

De hecho, es muy sencillo generarlo. Tú mismo podrías hacer un programa que te permita explorar la infinitud del conjunto.

Empezaba a picarme la curiosidad, así que me armé de valor y seguí escuchando...

Bueno, el algoritmo es sencillo. Basta con saber sumar y multiplicar número complejos. ¿recuerdas cómo se hacía?

Antes de darme la oportunidad de responser prosiguió explicándome que un número complejo está compuesto de una parte real y otra imaginaria y que tienen la forma a+bi (por ejemplo 5+3i) donde el primer número es un

número real, y el segundo es otro número multiplicado por la unidad imaginaria, que equivale a la raíz cuadrada de -1.
También me explicó que para sumar dos números complejos había que sumar por separado la parte real y la imaginaria. Por ejemplo (3+7i) + (5+3i) = (8+10i).

Sobre la multiplicación, me explicó que debía hacerse como si se tratara de la multiplicación de dos polinomios normales, pero con la particularidad de que i^2 es igual a -1. Por lo tanto (a+bi)*(c+di) = ac+adi+bci+bdi^2 = (ac-bd) +(ad+bc)i.
En realidad, podemos ver un número complejo como las dos coordenadas de un plano, en el que la parte real indica el valor de la abscisa y la parte imaginaria el valor de la ordenada.

Sofía me mostró un pequeño programa en Java que había escrito para realizar la imagen que acababa de mostrarme.

```
1.  import java.awt.*;
2.  import javax.swing.*;
3.
4.  class JMandelbrot extends JFrame {
5.
6.      final int RES=600;   // Resolución
7.      final int ITERACIONES=255;   // iteraciones
8.
9.      // Porción del conjunto de Mandebrot a mostrar
10.     final double XPOS=-2.0;
11.     final double YPOS=-1.5;
12.     final double LADO= 3;
13.
14.     public JMandelbrot() {
15.         super("Mandelbrot");
16.         setBounds(0,0,RES,RES);
17.         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18.         Container con=this.getContentPane();
19.         con.setBackground(Color.white);
20.         GCanvas canvas=new GCanvas(RES, XPOS, YPOS, LADO, ITERACIONES);
21.         con.add(canvas);
22.         setVisible(true);
```

```
23.     }
24.     public static void main(String[] args) {
25.         new JMandelbrot();
26.     }
27. }
28.
29. class GCanvas extends Canvas {
30.
31.     int res, iteraciones;
32.     double xpos, ypos, lado, delta;
33.
34.     public GCanvas(int res, double xpos, double ypos, double lado, int
    iteraciones) {
35.         this.res=res;
36.         this.xpos=xpos;
37.         this.ypos=ypos;
38.         this.lado=lado;
39.         this.delta = lado/res;
40.         this.iteraciones=iteraciones;
41.     }
42.
43.     double c_real, c_imaginaria, z_real, z_imaginaria, zr, zi, modulo;
44.     int cont, color;
45.     public void paint(Graphics g) {
46.         for (int i=0; i<res; i++) {
47.             for (int j=0; j<res; j++) {
48.                 // cálculo del número complejo C correspondiente
49.                 // al pixel actual.
50.                 c_real=xpos+(i*delta);
51.                 c_imaginaria=ypos+(j*delta);
52.
53.                 // inicializamos Z
54.                 z_real=0.0;
55.                 z_imaginaria=0.0;
56.                 cont=0;
57.
58.                 // calcular si el punto pertenece al conjunto de Mandelbrot
59.                 do {
60.                     zr=((z_real*z_real)-(z_imaginaria*z_imaginaria))
    +c_real;
61.                     zi=2*(z_real*z_imaginaria)+c_imaginaria;
62.                     z_real=zr;
63.                     z_imaginaria=zi;
64.                     cont++;
65.                     modulo=Math.sqrt((z_real*z_real)+
    (z_imaginaria*z_imaginaria));
66.                 } while (cont<=iteraciones && modulo<2);
67.
68.                 // transformar el valor a escala de grises.
69.                 if (cont<iteraciones) {
70.                     color=((iteraciones-cont)*255)/iteraciones;
71.                 } else {
72.                     color=0;
73.                 }
```

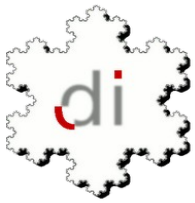
```
74.
75.                     // dibujar el pixel
76.                     g.setColor(new Color(color, color, color));
77.                     g.drawLine(i, j, i, j);
78.                 }
79.             }
80.         }
81. }
```

Ayudándose del listado del programa comenzó a explicarme el algoritmo que generaba aquellas imágenes tan intrigantes.

Verás, lo primero que hacemos es especificar qué parte del conjunto queremos explorar. Lo hacemos indicando el área con las variables *xpos* e *ypos*. Seguidamente, con la variable *lado* indicamos que tamaño del lado del cuadrado que tendrá el área a mostrar. Como el área que queremos explorar puede tener cualquier tamaño arbitrario, hay que adaptarlo a la resolución de la imagen (indicada por la constante *RES*). Para ello calculamos la variable *delta* dividiendo *lado* entre la resolución de la imagen, obteniendo así la distancia que media entre dos pixeles adyacentes.

Finalmente, entramos en harina y buscamos qué puntos del plano que estamos explorando pertenecen o no al conjunto de Mandelbrot. Para ello evaluamos cada uno de los pixeles de la imagen mediante dos bucles anidados donde la variable *i* representa el eje de abscisas y *j* el eje de ordenadas. Suponiendo que nos encontramos en la fila *i*, columna *j*, los pasos que realizaremos son los siguientes:

- Calcular el número complejo *c* que representa el pixel *i,j*. La parte real se obtiene sumando a *xpos* la columna actual multiplicada por *delta*. La parte imaginaria se obtiene igual, pero multiplicando *j* por *delta*. Es decir:



c_real=xpos+(i*delta);

c_imaginaria=ypos+(j*delta)

- Asignar a una variable compleja *z* el valor 0+0i, y crear una variable contador (*cont*) también con el valor 0.

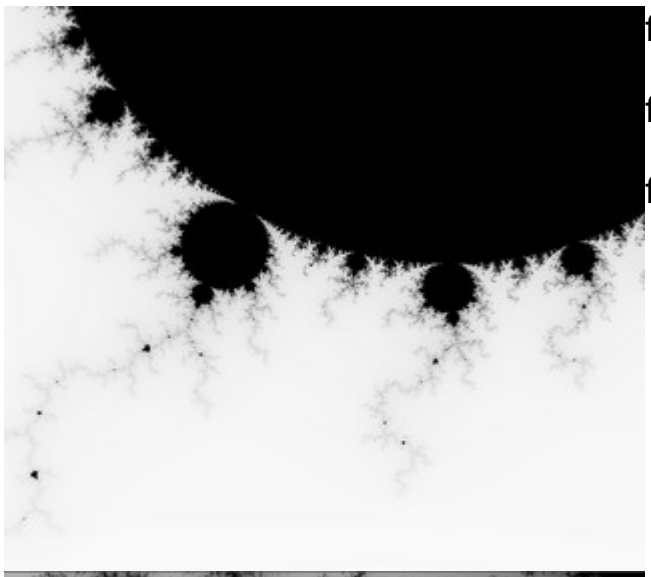
Repetir las siguientes dos operaciones hasta que el módulo de *z* sea mayor que 2 o bien que la variable cont exceda un valor prefijado (en nuestro caso almacenado en la variable *iteraciones*).

z = z^2 + c

cont = cont +1

Si terminamos y el contador es mayor que el valor prefijado, quiere decir que ese punto pertenece al conjunto de Mandelbrot, ya que no diverge (se mantiene acotado con un módulo inferior a 2 después de varias iteraciones). Este punto lo colorearemos de negro.
Si no, quiere decir que el punto no pertenece al conjunto de Mandelbrot y lo pondremos de blanco o le asignaremos un color en función del valor de la variable cont. En nuestro caso hemos asignado un tono de gris. Mientras más grande sea el valor prefijado (variable iteraciones) más pequeñas serán las zonas a las que podamos acceder (y más tiempo tardará el conjunto en generarse).

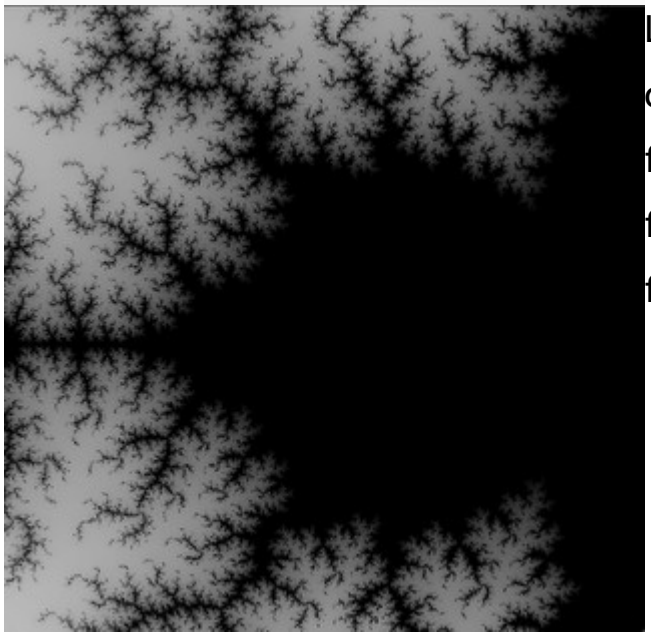
Como curiosidad, Sofía me mostró un par de parajes curiosos. Para ver el primero asignamos los siguientes valores a las constantes:



final double XPOS=-1.275;

final double YPOS=0.0916;

final double LADO= 0.4;



La segunda tiene las siguientes coordenadas:

final double XPOS=-1.17286;

final double YPOS=0.24133;

final double LADO= 0.003;

Sea como fuere, ahora empezaba a ver al romanescu con mejores ojos. Quizás, quién sabe, hasta tenga buen sabor.

4. Tráfico, cervezas y autómatas celulares

Mi amigo Descollante es probablemente la persona más cuadriculada que existe. Si quedas a una hora con él puedes estar seguro que estará en el lugar y hora acordado pase lo que pase, así que con toda segurdad, Descollante ya estaba esperándome. Yo no soy de los que suele llegar tarde, pero ¿qué podía hacer? estaba encerrado en un terrible caravana de esas impredecibles.

Había quedado con mi amigo para tomar unas cervezas y charlar sobre un nuevo proyecto que íbamos a emprender. Una aplicación que iba a revolucionar el mundo del software (evidentemente esto es sólo un sarcasmo, ya que todo quedará seguramente, en agua de borrajas, como todos los proyectos que emprendemos). El caso, es que allí estaba yo, una hora más tarde, saludando y sentándome junto a un Descollante con cara de malas pulgas y dos cervezas de ventaja.

Lo siento hombre, ni te imaginas la caravana que me ha pillado -le dije. Ya sabes... son imprevisibles. Nunca sabes cuando te va a pillar una. ¿imprevisibles? Que va hombre... El tráfico se comporta como un simple modelo estocástico -dijo Descollante con esa mirada que pone cuando está

a punto de descubrir la Teoría de Todo.

Lo que quiero decir -continuó Descollante- es que podemos encontrar un modelo probabilístico para describirlo. ¿Recuerdas cuando el otro día hablábamos de el juego de la vida?

En ese mismo momento llegó la camarera con dos cervezas (la tercera para Descollante) y las dejó sobre la mesa junto con unos cacahuetes.
Sí, claro que lo recuerdo, pero ¿qué tiene que ver el juego de la vida con el tráfico?
Descollante sonrió de nuevo y me lanzó su típica mirada de superioridad.
El juego de la vida es un autómata celular, y en ellos el espacio, el tiempo y sus estados toman valores discretos, por lo que son muy eficientes para para implementar simulaciones en un ordenador.

Ya, pero en el juego de la vida no hay ningún proceso estocástico. Todo depende del estado inicial -dije, intentando demostrar que tenía el tema controlado.
Efectivamente -Dijo Descollante con cara de aprobación- Hay que añadir alguna variable probabilística, algo de caos al sistema, y por supuesto, una serie de reglas que rijan el cambio de estados.

Tengo que reconocer que la cosa empezaba a entusiasmarme, pero no sé si por la conversación en sí o porque yo ya estaba terminando la segunda cerveza. Curiosamente a Descollante no parecía afectarle su recién terminada cuarta jarra (en esta ocasión de cerveza negra).

Veamos, -continuó- en este caso nuestro autómata celular será de una sóla dimensión (es decir, lo podemos representar como un array de una

dimensión). Vamos a asumir que sólo simulamos el tráfico de un carril para simplificar el modelo. Cada celda va a representar a un vehículo más cierta distancia de separación entre coches. Digamos que cada celda va representar un espacio de unos 7 u 8 metros.

Comenzó a garabatear en una servilleta y continuó explicándome que cada vehículo tenía un estado compuesto por su posición en la retícula del autómata y su velocidad. En este caso, la velocidad estaría limitada para todos los coches por igual, por lo que la velocidad de un coche estaría comprendida entre 0 (parado) y la velocidad máxima permitida. Seguidamente me habló del modelo Nagel-Schreckenberg, que según mi amigo, era de los modelos más simples ideados para simular el tráfico y que estaba compuesta por 4 sencillos pasos:

El primero de los pasos -dijo Descollante aún con un cacahuete en la boca- representa la tendencia de los conductores a ir lo más deprisa posible. Si un coche no ha alcanzado la velocidad máxima en el instante t incrementamos su velocidad en una unidad.

El segundo paso modela la interacción entre vehículos. Si hay un coche delante a una distancia d y la velocidad v del coche que estamos evaluando es mayor que d, entonces la velocidad del coche pasa a ser d en vez de v. Esto es, se procura mantener la distancia de seguridad para evitar el choque entre coches.

En el tercer paso añadimos un poco de caos. Tratamos de simular aquellos coches que de pronto, sin motivo aparente, frenan o bajan su velocidad. En

este caso, con probabilidad p, el coche reducirá su velocidad en el instante t+1 en una unidad.

Por último, la velocidad v para el instante t+1 ha quedado determinada para cada coche y sólo nos resta aumentar la posición del coche en la retícula del autómata en v posiciones.

Todo me pareció bastante sencillo, a pesar de las cuatro cervezas que ya me había metido en el cuerpo. Así que mientras volvía a casa (en taxi, por supuesto) iba dándole vueltas a la cabeza intentando acordarme de por qué habíamos acabado hablando del tráfico en vez de hablar de nuestro proyecto.

Cuando llegué a casa no podía quitarme de la cabeza todo lo que me había contado Descollante, así que me senté ante el ordenador y a duras penas pude teclear el siguiente programa en Java antes de quedarme dormido sobre el teclado.

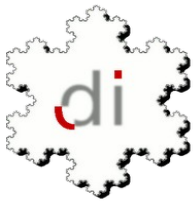
```
1.  import java.awt.*;
2.  import javax.swing.*;
3.
4.  class Trafico extends JFrame {
5.
6.      public Trafico() {
7.          super("Trafico");
8.          setBounds(0,0,600,200);
9.          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10.         Container con=this.getContentPane();
11.         con.setBackground(Color.white);
12.         GCanvas canvas=new GCanvas();
13.         con.add(canvas);
14.         setVisible(true);
```

```
15.     }
16.     public static void main(String[] args) {
17.         new Trafico();
18.     }
19. }
20.
21. class Coche {
22.     int pos;
23.     int velocidad;
24. }
25.
26. class GCanvas extends Canvas implements Runnable {
27.     int TAM_CARRETERA=600;
28.     int VMAX=4; // velocidad máxima
29.     double prob_frenada = 0.5; // prob. de frenada.
30.     int tiempo_pausa=100;
31.     Coche[] carretera = new Coche[TAM_CARRETERA];
32.     Thread t;
33.     int velLlegada=4; // cadencia de llegada de coches
34.
35.     public GCanvas() {
36.         t=new Thread(this);
37.         t.start();
38.     }
39.
40.
41.     public void paint(Graphics g) {
42.         // dibujar carretera
43.         g.drawLine(0, 95, 600, 95);
44.         g.drawLine(0, 115, 600, 115);
45.         for (int i=0; i<TAM_CARRETERA; i++) {
46.             if (carretera[i] != null) {
47.                 g.fillRect(i, 100, 1, 10);
48.             }
49.         }
50.     }
51.
52.     @Override
53.     public void run() {
54.         int cont = velLlegada;
55.         do {
56.             // ¿Crear un coche?
57.             if (cont-- == 0) {
58.                 Coche nuevoCoche=new Coche();
59.                 nuevoCoche.velocidad=0;
60.                 nuevoCoche.pos=0;
61.                 carretera[0]=nuevoCoche;
62.                 cont = velLlegada;
63.             }
64.
65.             // calcular desplazamientos
66.             for (int i=0; i<TAM_CARRETERA; i++) {
67.                 if (carretera[i] != null) {
68.                     // aceleración
```

```
69.                     if (carretera[i].velocidad<VMAX) {
70.                         carretera[i].velocidad++;
71.                     }
72.
73.                     // distancia seguridad
74.                     for (int j=1; j<carretera[i].velocidad; j++) {
75.                         if (i+j < TAM_CARRETERA) {
76.                             if (carretera[i+j] != null) {
77.                                 carretera[i].velocidad=j;
78.                                 break;
79.                             }
80.                         }
81.                     }
82.
83.                     // probabilidad frenada
84.                     double p = Math.random();
85.                     if (p>prob_frenada) {
86.                         carretera[i].velocidad--;
87.                     }
88.                 }
89.             }
90.
91.             // mover coches
92.             for (int j=TAM_CARRETERA-1; j>=0; j--) {
93.                 if (carretera[j] != null) {
94.                     Coche cocheTmp=carretera[j];
95.                     carretera[j]=null;
96.                     cocheTmp.pos += cocheTmp.velocidad;
97.                     if (cocheTmp.pos < TAM_CARRETERA-1) {
98.                         carretera[cocheTmp.pos]=cocheTmp;
99.                     }
100.                }
101.            }
102.
103.            repaint();
104.            try {
105.                Thread.sleep(tiempo_pausa);
106.            } catch (InterruptedException e) {
107.                e.printStackTrace();
108.            }
109.        } while (true);
110.    }
111.      }
```

5. Secretos inconfesables

El trabajo de programador es un trabajo poco agradecido y quizá por ello tan denostado por muchos informáticos. Hay que estar hecho de una pasta especial. Horas y horas ante un ordenador, implementando pedazos de código. Horas de concentración para no cometer errores (o los menos posibles). Horas y horas tratando de hacer realidad la fantasía perpetrada a partes iguales por el cliente y mi jefe de proyecto que, por supuesto, no ha escrito una línea de código en su vida. Eso sí, no esperes recibir la gloria el día que la aplicación comience a funcionar. Tu sólo habrás sido el albañil que ha colocado los ladrillos. Ni por asomo será mérito tuyo, sino de tu jefe por haber sabido apretarte las tuercas... En mi caso, ese jefe se llama Gaznápiro y aquél día se había levantado con ganas fastidiarnos un poco. A cada rato se posaba sobre mi hombro a mirar el código que estaba escribiendo y soltar la primera chorrada que se le pasaba por la cabeza: "Esos nombres de variable no me gustan", "Coloca las llaves



en la siguiente línea para que los bloques queden más claros", "No uses la doble barra para el comentario, que no es ANSI C", ... Y así toda la mañana. Hasta que por fin salió a ver a un cliente. Como tenía ganas de desahogarme le envié a Sofía el siguiente mensaje por email:

AOPA CWVJWMEÑL AO QJ EJQPEH

Era un mensaje cifrado. Lo enviaba así, por una parte, porque nunca me he fiado de la privacidad del correo electrónico del trabajo y, por otra, porque sabía que a Sofía le encantaban los enigmas.

No habían pasado ni cinco minutos cuando Sofía me miró con cara de perdonarme la vida: vas a tener que esmerarte más porque este cifrado lo descifraría mi abuela hasta borracha. Aunque te doy toda la razón a lo que dices.

Lo cierto es que era un cifrado muy básico llamado Cifrado del César, porque ya lo usó Julio César. Es lo que se llama un cifrado por sustitución, ya que lo que se hace es sustituir una letra por otra. En este caso yo había hecho la siguiente correspondencia:

ABCDEFGHIJKLMNÑOPQRSTUVWXYZ
WXYZABCDEFGHIJKLMNÑOPQRSTUV

Es decir, sustituir la A por W y así sucesivamente; o lo que es lo mismo,

desplazar el abecedario 4 posiciones a la derecha. Sustituyendo las letras de esta forma la frase en cuestión era:

ESTE GAZNAPIRO ES UN INUTIL

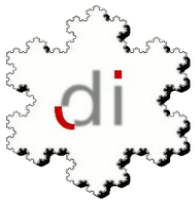
AOPA CWVJWMEÑL AO QJ EJQPEH

Sofía había deducido que "AO QJ" podría ser "ES UN" ya que no hay demasiadas palabras de dos letras que puedan ir juntas en una frase, y a partir de ahí había deducido el resto de sustituciones (supongo que antes probó con "ES LA" o "EN EL").

Sofía se impulsó sobre su silla de ruedas y se acercó a mi mesa: si quieres jugar hazlo bien -dijo con una sonrisa de oreja a oreja. ¿No se te ocurre nada mejor?

Sí -contesté- supongo que la clave está en buscar un algoritmo de codificación que sea sólo conocido por el emisor y el receptor. Supones mal -contestó Sofía. Seguramente un buen criptoanalista acabaría descubriendo tu algoritmo y ¿entonces qué? ¿inventas otro algoritmo? Y si no que se lo cuenten a los Alemanes de la Segunda Guerra Mundial y su máquina Enigma.

Lo mejor -continuó- sería poder contar con un sistema de encriptación de información que fuera seguro incluso aunque el algoritmo fuera conocido. Se trata de codificar la información con una clave que sólo el emisor y el receptor conozcan, y que aún sabiendo el procedimiento para desencriptar la información, sin la clave sea imposible o muy difícil de descifrar. Algo como lo que tu has hecho con el cifrado del César: Has usado la clave 4, ya que, por ejemplo, la letra R menos 4 posiciones se corresponde con la letra



Ñ.

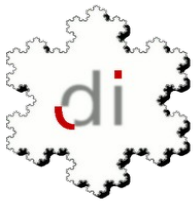
Pero tú has roto la clave en muy poco tiempo -contesté- Si uso otra clave la volverías a averiguar.

Sí, porque tu clave es pequeña -prosiguió Sofía- De hecho, sin tener ninguna pista, sólo tengo que probar 26 claves diferentes para dar con tu frase encriptada. La idea es tener una clave tan fuerte que sea computacionalmente muy costoso encontrarla probando todas las posibilidades. En eso se basan los sistemas criptográficos actuales, como RSA.

Había leído un artículo sobre RSA no hacía mucho. Es el método más usado para firmar y encriptar datos actualmente. Es un sistema criptográfico de clave pública o asimétrico, es decir, que al contrario de un sistema de clave simétrica, donde emisor y receptor deben conocer la clave (¿quién nos asegura que nadie la intercepta cuando emisor y receptor intercambian la clave?), en el caso de RSA disponemos de una clave privada, que sólo nosotros conocemos, y otra pública que conoce todo el mundo. De esta forma, si quiero enviar un mensaje cifrado a alguien sólo tengo que encriptarlo con su clave pública, y sólo él podrá desencriptarlo con su clave privada.

Lo que no tenía muy claro es cómo funcionaba internamente y por qué se suponía que era tan robusto. Afortunadamente Sofía estaba dispuestísima a explicármelo:

La seguridad de un buen sistema de cifrado se basa en encontrar un proceso que sea fácil de resolver computacionalmente en una dirección, pero muy difícil de resolver en la dirección contraria. En eso se basa RSA. La idea básica es encontrar dos números primos los suficientemente grandes. Una



vez los tengamos multiplicarlos es fácil, obtendremos un número muy grande. Sin embargo, el proceso contrario es muy difícil. Factorizar ese gran número para encontrar los primos iniciales es una tarea computacionalmente muy costosa. Sofía me explico los pasos para generar las claves pública y privada:

- Elegimos dos número primos a los que llamaremos P y Q. Mientras más grandes, más difícil será romper el cifrado, pero también será más costosa la encriptación/desencriptación. Hay que buscar un equilibrio. Primos de 1024bits en adelante son usados habitualmente con una seguridad bastante decente.

- Efectuamos la operación N=PxQ. A N lo llamamos Módulo de la clave. Por ejemplo, supongamos que tomamos P=157 y Q=251. En este caso N=39407. Evidentemente estos números son muy pequeños, pero así podremos seguir mejor el ejemplo.

- Calculamos phi(N)=(P-1)x(Q-1). Phi(N) es el número de enteros más pequeños que N que son primos relativos de N. En este caso phi(N)=(157-1)x(251-1)=39000.

- Buscamos un entero al que llamaremos E que es menor que phi(N) y además primo relativo con phi(N). Denominamos E como exponente de la clave pública. En este caso vamos a elegir E=5549.

- Buscamos un entero D tal que DxE=1 mod phi(N) (la inversa del modulo Phi(N)). A D lo denominamos exponente de la clave privada. Tras operar obtenemos D=7949.

En este punto tenemos ya la clave pública, que es la tupla (N, E) y la clave privada formada por (N, D).

Cifrar un mensaje es de lo más sencillo. Dado que RSA trabaja con números enteros habrá que codificar el texto que quiero cifrar con números enteros (por ejemplo, su valor ASCII). Supongamos que quiero cifrar un entero M para enviártelo a ti. Con tu clave pública lo encriptaría usando la fórmula:

valor_cifrado=M^E mod N.

Y para descifrarlo tú sólo tendrías que usar tu clave privada: M=valor_cifrado^D mod N.

Desgraciadamente -continuó Sofía- cuando trabajamos con valores grandes los números que se manejan son astronómicos y no son manejables por los tipos primitivos de ningún lenguaje de programación, que son habitualmente de hasta 64bits. Es por ello, que si quieres hacer una inplementación del algoritmo tendrás que recurrir a alguna librería para el cálculo con números de precisión arbitraria, como libgmp u otra similar.

Sofía se volvió a su ordenador y al rato me envió un correo con el siguiente

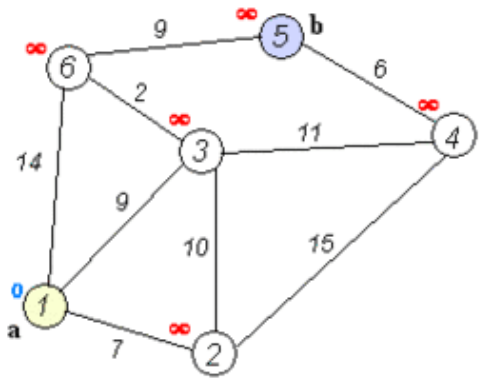
programa y un texto para descifrar que no reproduciré aquí por si mi jefe de proyecto, Gaznápiro, anda leyendo este blog.

```
1. // Este programa usa la librería libgmp (http://gmplib.org)
2. // para cálculo de números de precisión arbitraria.
3.
4. #include <stdio.h>
5. #include <gmp.h>
6.
7.
8. // Calculo de la calve privada a partir de la clave pública.
9. int genkey(int n1, int n2, mpz_t e , mpz_t palabra) {
10.     // n1 y n2 son números primos.
11.     // e es un número aleatorio primo relativo con phi(n)
12.
13.     int phi,i,tmp,ee,dd;
14.     mpz_t n,d,temp;
15.
16.     mpz_init(n);
17.     mpz_init(d);
18.     mpz_init(temp);
19.     tmp=n1*n2;
20.     mpz_set_si(n,tmp);
21.     ee=mpz_get_si(e);
22.     phi=(n1-1)*(n2-1);
23.
24.     // Buscamos la inversa de d módulo phi.
25.     i=1;
26.     dd=1;
27.     while (i>0) {
28.         tmp=ee*i++;
29.         if (tmp%phi==1) {
30.             // Tenemos d
31.             dd=i-1;
32.             i=0;
33.         }
34.     }
35.
36.     return dd;
37.
38. }
39.
40.
41. // Algoritmo de encriptación.
42. int crypt(int n1, int n2, mpz_t e , mpz_t palabra) {
43.     // n1 y n2 son números primos.
44.     // e es un número aleatorio primo relativo con phi(n)
45.
46.     int phi,i,tmp,ee;
47.     mpz_t n,temp;
48.
```

```
49.     mpz_init(n);
50.     mpz_init(temp);
51.     tmp=n1*n2;
52.     mpz_set_si(n,tmp);
53.     ee=mpz_get_si(e);
54.     phi=(n1-1)*(n2-1);
55.
56.
57.     mpz_powm(temp,palabra,e,n);
58.     tmp=mpz_get_si(temp);
59.     return tmp;
60.
61. }
62.
63.
64.
65. // Algoritmo de desencriptación RSA.
66. int decrypt(int n1, int n2, int d , mpz_t palabra) {
67.     // n1 y n2 son números primos.
68.     // e es un número aleatorio primo relativo con phi(n)
69.
70.     int tmp;
71.     mpz_t n,temp,dd;
72.
73.     mpz_init(n);
74.     mpz_init(dd);
75.     mpz_init(temp);
76.     tmp=n1*n2;
77.     mpz_set_si(n,tmp);
78.     mpz_set_si(dd,d);
79.
80.     mpz_powm(temp,palabra,dd,n);
81.     tmp=mpz_get_si(temp);
82.     return tmp;
83.
84. }
85.
86.
87. int main () {
88.     char i,texto[255];
89.     int j,d;
90.     mpz_t palabra,e;
91.
92.     // Definimos los parametros de la encriptación (Clave pública).
93.     int n1=157;
94.     int n2=251;
95.
96.     mpz_init(palabra);
97.     mpz_init(e);
98.     mpz_set_si(e,5549); // e = 5549
99.
100.    d=genkey(n1,n2,e,palabra);
101.    printf ("\n\nImplementación del algoritmo RSA de encriptación.\n");
102.    printf ("Clave Pública: N=39407, e=5549\n");
103.    printf ("1 - Encriptar.\n");
```

```
104.     printf ("2 - Desencriptar.\n");
105.     printf ("--\n");
106.     printf ("Entre opción:");
107.     scanf ("%c",&i);
108.
109.     switch (i) {
110.         case '1':
111.             printf ("Entre texto a codificar: ");
112.             scanf("%s",texto);
113.             j=0;
114.             do {
115.                 mpz_set_si(palabra,texto[j]);
116.                 printf("%d\n",crypt(n1,n2,e,palabra));
117.             } while ((texto[j++])!='\0');
118.             break;
119.         case '2':
120.             do {
121.                 printf ("\nEntre entero a decodificar: ");
122.                 scanf("%d",&j);
123.                 mpz_set_si(palabra,j);
124.                 printf("%d = %c\n",j,decrypt(n1,n2,d,palabra));
125.             } while (j!=0);
126.             break;
127.     }
128.
129.   return 0;
130.      }
```

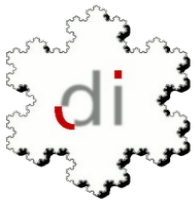
6. Atrapado en las redes de Dijkstra



Siempre que se acerca la fecha de entrega de cualquier proyecto, el estrés aumenta considerablemente. A la cercanía de la fecha se sumaba que llevábamos toda la mañana sin conexión de red y, por lo tanto, no podíamos acceder a los repositorios que están en la delegación central, para poder seguir con el trabajo. Mi superior inmediato, Gaznápiro, no paraba de pasear de arriba a abajo de la sala, y cada vez que pasaba ante nosotros refunfuñando, mi compañero, el señor Lego se mostraba visiblemente más nervioso. Como uno ya está de vuelta de casi todo y he acabado asumiendo que los problemas de mis jefes son de mis jefes y no míos (al igual que los sueldos de mis jefes son de mis jefes y no míos) cogí al señor Lego del brazo y casi arrastrándolo lo saqué fuera de la oficina para invitarle a un café.

Siempre se cae la red cuando más falta hace -se quejaba el sr. Lego- A ver si inventan una red a prueba de fallos.

De hecho, sr. Lego, las redes TCP/IP actuales son tolerantes a fallos -empecé a decir-. En la teoría, claro. Están diseñadas para que si un nodo



de la red cae, se pueda reencaminar el tráfico por otro.

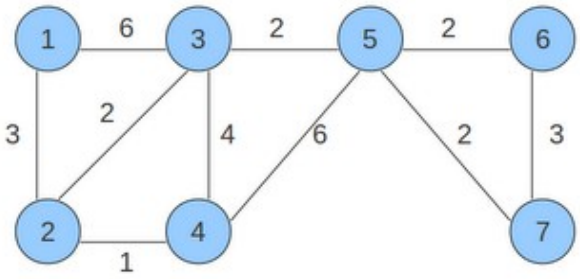
El Sr. Lego me confesó que no tenía muy claro eso de las redes, los nodos, los caminos y los paquetes. ¿cómo decide un router a qué nodo tiene que enviar un paquete?

Bueno Sr. Lego, en realidad los conceptos detrás del diseño de redes vienen de lejos y no son tan complicados. Si quiere le cuento algunas cosas que aprendí mientras trabajaba con mi amigo Descollante como administrador de redes.

El Sr. Lego no contestó, así que interpreté que me permitía, sin mucho entusiasmo, seguir adelante.

Para simplificar usaremos los grafos para modelar las redes. Un grafo está compuesto por un conjunto de nodos que pueden o no estar conectados entre sí. En nuestro modelo, la velocidad de transmisión puede ser distinta entre dos nodos, ya sea por limitaciones físicas, técnicas o simplemente por que haya más tráfico en un enlace concreto. Para representar esto le damos un peso a cada enlace o arista. Esto es lo que se llama un grafo ponderado. Para simplificar, vamos a suponer que la comunicación es siempre bidireccional entre nodos, aunque en la vida real podría darse el caso de que dos nodos sólo pudieran comunicarse en un sentido (por un cortafuegos, por ejemplo) o incluso que la velocidad fuera distinta en un sentido y el otro (porque haya más paquetes esperando ser enviados en un sentido, por ejemplo). En ese caso usaríamos un digrafo o grafo dirigido para representar el modelo.

Cogí una servilleta e hice el siguiente esquema.



Cada nodo representa un router en la red, y cada línea es un enlace con su peso (velocidad de transmisión). En una red real, los pesos podrían cambiar dinámicamente según las condiciones de la red (tráfico, caída de un enlace, interferencias o ruidos en la comunicación, etc...). Para nuestro ejemplo lo supondremos estático por simplificar.

Ya veo, entonces para ir del nodo 1 al 4 podemos coger el camino 1-3-4 y 1-2-4 ¿no?

Efectivamente sr. Lego, de hecho también podríamos coger el 1-3-2-4 o el 1-2-3-4. O incluso los routers podrían decidir por su propia cuenta seguir la ruta 1-2-3-1 indefinidamente. Es lo que se llama un bucle o ciclo, y es muy importante que un paquete que se transmita por la red no quede dando vueltas en un bucle y no llegue a su destino. ¿Alguna idea de como solucionarlo sr. Lego?

Bueno, podríamos hacer que los paquetes lleven la cuenta de por cuántos routers han pasado y si supera cierta cantidad, pues se descartan.

Bien sr. Lego, de hecho en TCP/IP, los routers van aumentando un contador de saltos que tiene el paquete de forma que cuando superan un valor

llamado TTL (Time To Live) estos son descartados. Esto soluciona el problema de que no queden indefinidamente dando vueltas por la red, pero no soluciona el problema de que entren en un bucle. Para este caso, nos interesaría obtener un grafo en el que hayamos eliminado los bucles, o lo que es lo mismo, un árbol que mantenga un único camino posible entre nodos, y a ser posible, que los caminos que queden sean los mejores. Es lo que se conoce como árbol de expansión, árbol recubridor o spanning tree, como dicen los hijos de Shakespeare.

Kruskal, que por cierto falleció no hace muchos meses, fue un matemático que propuso un algoritmo para encontrar el árbol de expansión mínimo de un grafo dado. Es mínimo porque el árbol que nos interesa es aquel en el cual la suma de los pesos de las aristas es mínimo.

Los pasos del algoritmo de Kruskal son los siguientes:

1 - Se selecciona la arista con menor valor (si hay más de una, cualquiera de ellas).
2 - Repetir el paso 1 siempre que la arista elegida no forme ciclo con cualquiera de las seleccionadas anteriormente.

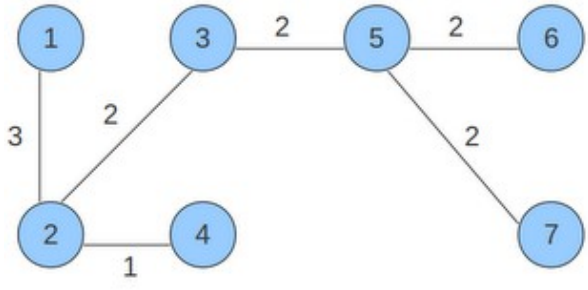
3 - El algoritmo termina cuando hemos seleccionado n-1 aristas del grafo (n=número de nodos), o lo que es lo mismo, todos los nodos tienen alguna arista seleccionada.

En nuestro ejemplo el algoritmo sería:

- Seleccionamos la arista (2,4) con peso 1

- Seleccionamos la arista (2,3) con peso 2
- Seleccionamos la arista (5,7) con peso 2
- Seleccionamos la arista (5,6) con peso 2
- Seleccionamos la arista (3,5) con peso 2
- Descartamos la arista (6,7) con peso 3, ya que forma ciclo con (5,6) y (5,7).
- Seleccionamos la arista (1,2) con peso 3

En este punto hemos acabado, ya que tenemos n-1 aristas seleccionadas, quedando el siguiente árbol de expansión.



Ahora podríamos enviar un paquete desde un punto a otro de la red con la seguridad de que nunca entrará en un bucle. Protocolos como STP (Spanning Tree Protocol) se encargan de construir este tipo de árboles dentro de una gran red, aunque utilizan mecanismos adicionales algo más complejos que los de Kruskal. Estos protocolos actúan en el nivel 2 de la capa OSI.

Además el camino será siempre mínimo -dijo el sr. Lego mientras apuraba su taza de café.

Fijesé sr. Lego que el algoritmo de Kruskal nos asegura que la suma de los

pesos de las aristas será mínimo, pero nada más. Por ejemplo, si quisiera enviar un paquete del nodo 6 al 7, si lo hacemos a través del árbol de expansión tendríamos que seguir el camino 6-5-7 con un coste de 4. Mientras que en el grafo original hubiéramos llegado directamente con el camino 6-7 con un coste de 3.

Los actuales algoritmos de enrutamiento como RIP (Routing Information Protocol), BGP (Border Gateway Protocol) u OSPF (Open Shortest Path First), que actúan en el nivel 3 de la capa OSI, son capaces de encontrar el camino más óptimo a partir de las condiciones actuales de la red (tráfico, congestión, estado de los enlaces, etc...). La mayoría de estos protocolos se basan en un algoritmo creado por Edsger Dijkstra, que es un algoritmo (del tipo voraz) capaz de encontrar el camino más corto en un grafo o digrafo ponderado como es el caso de nuestra red de ejemplo. Concretamente se encarga de calcular la distancia mínima de un nodo inicial al resto de nodos del grafo.

En el algoritmo de Dijkstra se etiqueta cada nodo con la tupla *[dist, padre]* donde la dist es la distancia mínima desde el nodo inicial (distancia acumulada) y padre es el nodo predecesor en el camino mínimo que une el nodo inicial con el que se está calculando.

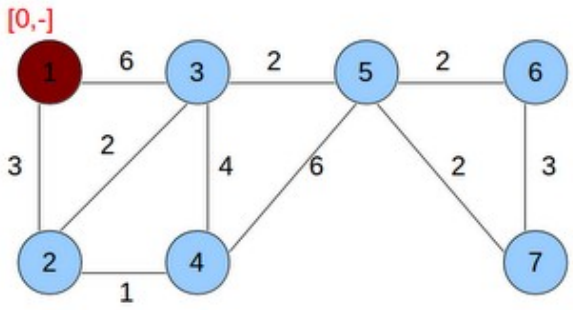
Básicamente los pasos del algoritmo son:

1 - Seleccionamos el nodo no visitado con menor distancia acumulada.
2 - Sumamos la distancia acumulada con la distancia a los nodos adyacentes y los etiquetamos con [dist, padre]. En caso de que alguno de los nodos

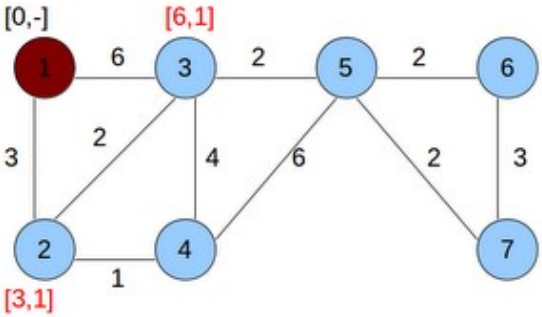
adyacentes esté ya etiquetado nos quedamos con el de menor distancia acumulada.

3 - Marcamos el nodo actual como visitado y regresamos al paso 1.

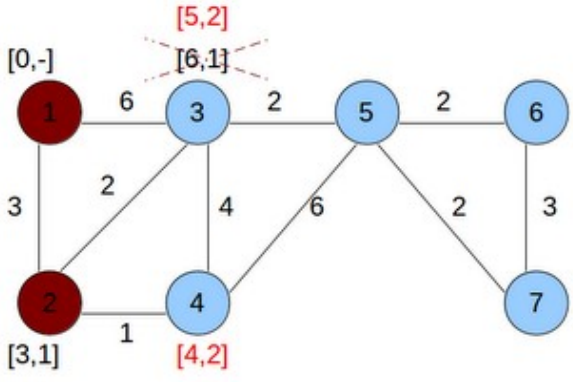
Pero mejor lo vemos con un ejemplo. Vamos a calcular el camino mínimo desde el nodo 1 al resto de nodos en nuestro grafo -dije mientras acumulaba una buena provisión de servilletas en las que dibujar.



Elegimos el primer nodo (el nodo 1) y lo marcamos con distancia 0 y sin padre.

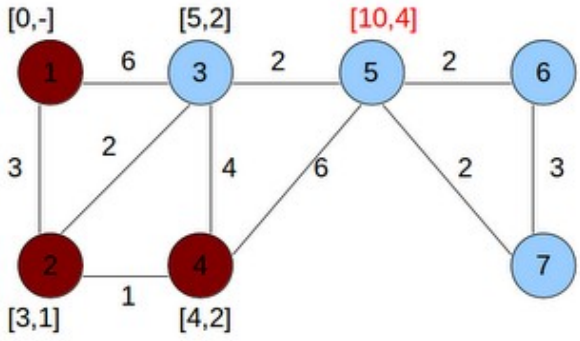


Etiquetamos los dos nodos adyacentes con la distancia y el nodo predecesor. También marcamos el nodo 1 como visitado (en rojo).

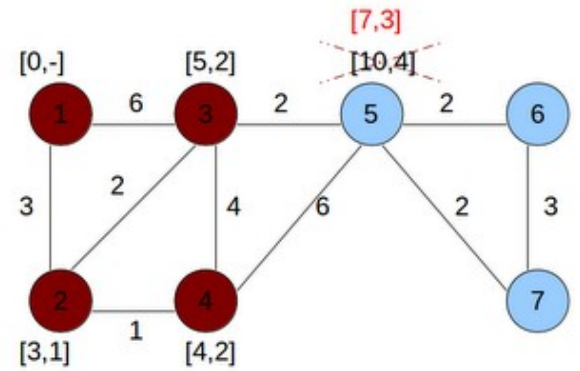


Seleccionamos el nodo con menor distancia acumulada. En este caso el 2 que tiene peso acumulado 3. Seleccionamos los nodos adyacentes y

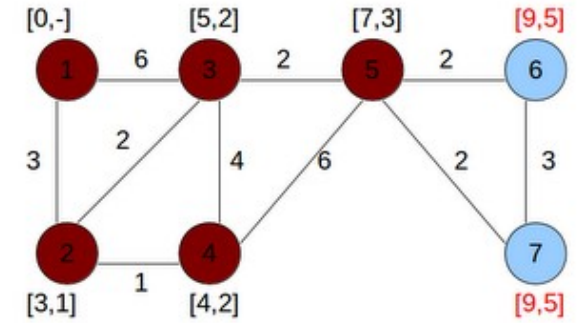
los etiquetamos con el peso acumulado (peso del nodo actual más el peso de la arista del nodo adyacente). Como padre ahora pondremos el nodo 2. Como ves el nodo 3 tiene ahora dos etiquetas, así que nos quedamos con la de menor peso. Marcamos el nodo 2 como visitado.



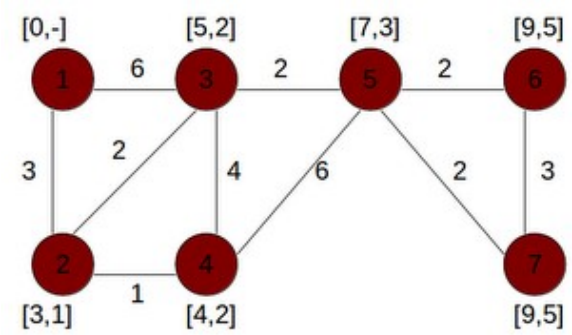
Seleccionamos el nodo 4 que es el de menor peso acumulado. En este caso ya no tenemos en cuenta el nodo 2 porque ya está marcado como visitado ni el nodo 3 porque el peso acumulado que tiene marcado es menor que el nuevo, así que etiquetamos el nodo 5 y marcamos el 4 como visitado.



El siguiente nodo de menor peso acumulado es el 3. Como el 4 ya está visitado sólo nos queda por etiquetar el 5. En este caso, como el nuevo peso acumulado es menor reetiquetamos con el nuevo peso y el nuevo padre. Marcamos el nodo 3 como visitado.



Seleccionamos el nodo 5 y etiquetamos los dos nodos adyacentes no vitados. Seguidamente lo marcamos como visitado.



Finalmente seleccionamos el 6 y como no mejora el peso acumulado hasta el nodo 7 no cambiamos la etiqueta. Lo mismo ocurre con el 7, así que marcamos ambos como visitados y obtenemos finalmente los siguientes pesos y caminos.

Ahora podemos responder cualquier pregunta que nos hagan sobre el grafo. Por ejemplo:

¿Cuál es la distancia (peso) mínima desde el nodo 1 al nodo 7? Directamente podemos contestar que es 9, ya que es su peso acumulado.

¿Cuál es el camino mínimo del nodo 1 al 7? Sólo hay que mirar quién es el padre del nodo 7 e ir recorriendo el camino hacia atrás viendo quién es el padre de cada nodo hasta llegar al 1. En este caso, el camino mínimo es 1-2-3-5-7.

El Sr. Lego parecía pensativo mientras terminaba de dibujar los esquemas.

¿Y esto se podría aplicar, por ejemplo, a el cálculo de la ruta más corta para un viaje por carretera? -dijo el Sr. Lego

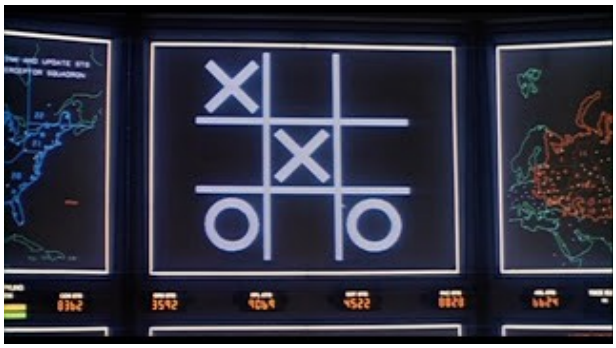
Claro que sí, y también para la obtención de la mejor ruta para un viaje en metro. Las aplicaciones son muchas y muy variadas.

Ya de vuelta en la oficina el Sr. Lego se enfrascó en la tarea de hacer una implementación del algoritmo de Dijkstra mientras se recuperaba la conectividad en la red y este fue el resultado.

```
1.  import sys
2.
3.  def dijkstra(grafo, nodo_inicial):
4.      etiquetas = {}
5.      visitados = []
6.      pendientes = [nodo_inicial]
7.      nodo_actual = nodo_inicial
8.
9.      # nodo inicial
10.     etiquetas[nodo_actual] = [0,'']
11.
12.     # seleccionar el siguiente nodo de menor peso acumulado
13.     while len(pendientes) > 0:
14.         nodo_actual = nodo_menor_peso(etiquetas, visitados)
15.         visitados.append(nodo_actual)
16.
17.         # obtener nodos adyacentes
18.         for adyacente, peso in grafo[nodo_actual].iteritems():
19.             if adyacente not in pendientes and adyacente not in visitados:
20.                 pendientes.append(adyacente)
21.             nuevo_peso = etiquetas[nodo_actual][0] + grafo[nodo_actual]
    [adyacente]
22.             # etiquetar
23.             if adyacente not in visitados:
24.                 if adyacente not in etiquetas:
25.                     etiquetas[adyacente] = [nuevo_peso, nodo_actual]
26.                 else:
27.                     if etiquetas[adyacente][0] > nuevo_peso:
28.                         etiquetas[adyacente] = [nuevo_peso, nodo_actual]
29.
30.         del pendientes[pendientes.index(nodo_actual)]
31.
32.     return etiquetas
33.
34.
35. def nodo_menor_peso(etiquetas, visitados):
36.     menor = sys.maxint
37.     for nodo, etiqueta in etiquetas.iteritems():
38.         if etiqueta[0] < menor and nodo not in visitados:
39.             menor = etiqueta[0]
40.             nodo_menor = nodo
41.
42.     return nodo_menor
43.
44.
```

```
45. if __name__ == "__main__":
46.     grafo = {
47.         '1': {'3':6, '2':3},
48.         '2': {'4':1, '1':3, '3':2},
49.         '3': {'1':6, '2':2, '4':4, '5':2},
50.         '4': {'2':1, '3':4, '5':6},
51.         '5': {'3':2, '4':6, '6':2, '7':2},
52.         '6': {'5':2, '7':3},
53.         '7': {'5':2, '6':3}}
54.
55.     etiquetas = dijkstra(grafo, '1')
56.     print etiquetas
```

7. La máquina invencible



Es raro que algún día me encuentre en la oficina con poco trabajo que hacer. También es raro que no ande por allí mi jefe Gaznápiro fustigándonos con su verborrea y sus dotes para convertir el proyecto más simple en un infierno. Sin embargo, que se den las dos cosas a la vez es un hecho tan estadísticamente improbable como que no te pille un gran atasco a la salida del trabajo.

Aquella mañana el azar había querido que el Sr. Lego y yo andáramos enfrascados tranquilamente en una conversación cinematográfica. Hablábamos de los gazapos informáticos que minaban la mayoría de las películas, lo cuales no son pocos, y quizás tan habituales como los gazapos científicos, pero esa es otra historia. Hablábamos de clásicos como Tron, Hackers o Conspiración en la red. Todo iba bien hasta que el Sr. Lego comenzó a criticar Juegos de Guerra, lo cual era demasiado para alguien como yo, que conocía al dedillo cada frase del diálogo y, siendo aún un adolescente, había descubierto en esta película las maravillas de la Inteligencia Artificial y el Hacking reunidos en una hora y media de metraje.

.- Menudo rollo ese del ordenador que habla como un humano -se quejaba

el Sr. Lego- ni que los ordenadores pudieran pensar.

.- De hecho, Sr. Lego, ya en 1950 Alan Turing, en un artículo llamado Computing machinery and inteligence, propuso su famoso Test de Turing para evaluar la inteligencia de un ordenador. Básicamente sostiene que si una máquina se comporta en todos los aspectos como inteligente, entonces debe ser inteligente (no entraré en disquisiciones filosófica sobre si esto tiene o no sentido). Le recomiendo a usted que repase mentalmete la charla que tuvimos hace unos meses sobre Inteligencia Artificial en la que hablamos de cómo solucionar puzzles sencillos en un ordenador.

.- Cierto, pero una cosa es un puzzle de niños y otra que el ordenador hable y hasta sea capaz de ganarle a un humano jugando al tres en raya, las damas o el ajedrez.

.- Sr, Lego, ¿no ha jugado usted nunca a un juego de ajedrez? O es usted realmente bueno o se verá en complicaciones para ganarle al ordenador de su casa. Imaginese enfrentarse a ordenadores especificamente diseñados para jugar como el famoso Deep Blue, que ya tiene sus añitos.

.- Bueno, supongo que usaran técnicas similares a las que me enseñaste cuando hablamos de los puzzles ¿no?

.- Si y no -dije tras pensármelo un poco. Realmente hay varias estrategias que dependen mucho del juego al que se quieran aplicar, aunque

básicamente se basan en lo mismo. En la búsqueda en un espacio de estados, como en el caso del puzzle (aunque se pueden utilizar otras técnicas como sistemas expertos, redes de neuronas, etc... pero de eso hablamos otro día). Una de las más conocidas y usadas es el llamado algoritmo minimax, que probablemente es el que hubiera usado la WOPR para jugar al tres en raya (o tic-tac-toe como lo llaman los angloparlantes).

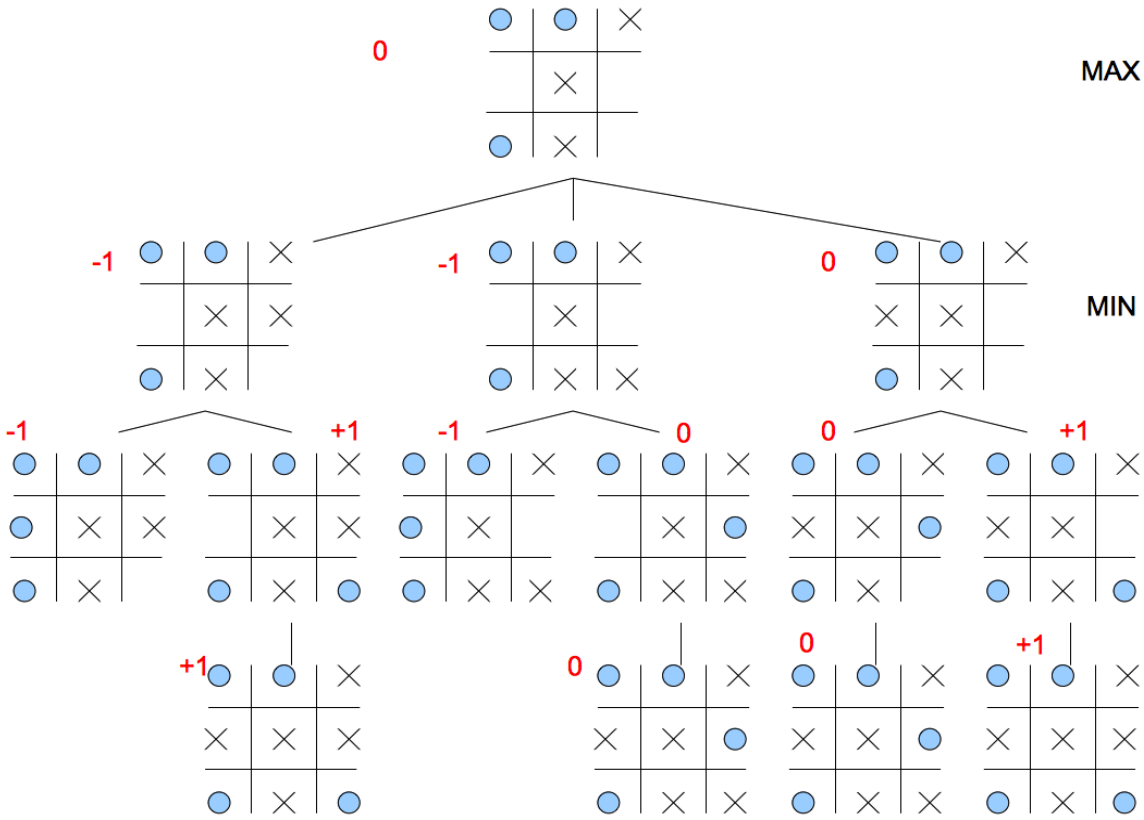
.- Es decir, que el minimax ese no es más que una búsqueda en un árbol de estados.

.- Bueno, si quiere le cuento como funciona el algoritmo. Es más sencillo de lo que usted cree.

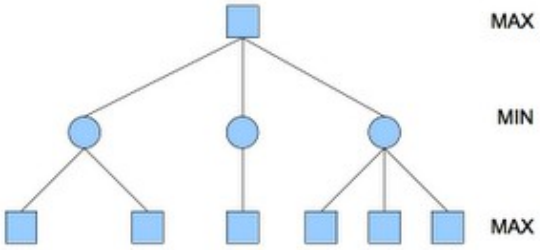
Cuando atacamos el problema de resolver el puzzle, estábamos enfrentándonos a lo que se denomina un juego sin contrincante. En el caso del 3 en raya o el ajedrez, sí hay un adversario, y por lo tanto, utilizaremos la estrategia minimax, que se adapta mejor a lo que se denomina juego con contrincante. Hay que tener en cuenta que usaremos esta estrategia en juegos en los que no influya el azar (como en juegos de cartas), sino en lo que denominaremos juegos perfectamente informados (es decir, que tenemos toda la información tanto de nuestra jugada como la del adversario). Como ejemplo tomaremos como modelo el juego del 3 en raya o tic-tac-toe por ser más sencillo.

El algoritmo minimax construye un árbol partiendo del estado actual del juego. Supongamos que en el siguiente turno le toca mover al ordenador. A partir del estado actual del juego (es decir, la disposición de las piezas en el tablero) construiremos un nodo hijo por cada posible movimiento legal que

puede realizar el ordenador (llamaremos al ordenador jugador MAX). En el



siguiente nivel, por cada posible movimiento del ordenador generaremos los posibles movimientos legales que podría realizar el jugador humano (llamaremos al usuario jugador MIN), y así sucesivamente hasta llegar a un nodo (que llamaremos nodo terminal) en el que haya un desenlace válido, es decir, que gane el ordenador, que gane el jugador o que haya empate. Es decir, que en cada nivel del árbol se van alternando el jugador MAX y el jugador MIN (de ahí el nombre de estrategia minimax). Si representamos con un cuadrado los nodos MAX y con un círculo los nodos MIN, un posible despliegue de un árbol minimax sería el siguiente.



Cada vez que llegamos a un nodo terminal usamos lo que llamaremos función de evaluación, que no es más que una función que sopesa qué tan buena es la solución a la que conduce ese nodo terminal, asignándole un valor numérico. En el caso del tic-tac-toe la función es muy sencilla. Por ejemplo, podríamos asignar un +1 a los nodos terminales en los que gana el ordenador, un 0 en los que se produce empate y un -1 en los que gana el jugador humano.

En el siguiente esquema vemos un posible despliegue minimax para el juego del tic-tac-toe.

El algoritmo evalúa cada nivel del árbol de abajo hacia arriba, y en los niveles MAX elegirá los caminos que conduzcan a puntuaciones más altas, que son las que favorecen al ordenador, y en los niveles MIN tratará de escoger aquellos caminos con menor puntuación, ya que penalizan al jugador humano.

Veámoslo con un ejemplo. En la figura de arriba se despliega un árbol a partir del estado actual de juego (nodo MAX). La función de evaluación recorre el árbol hasta llegar a los nodos terminales, y los puntúa con +1 para nodos donde gana el ordenador, 0 para empate y -1 para nodos donde

gana el jugador humano. En la gráfica se marca con +1 el nodo inferior derecho y el inferior izquierdo (gana MAX) y con 0 los dos centrales (tablas). Los nodos inmediatamente superiores son nodos MAX, por lo que puntuaremos los estados con el mayor valor de los hijos (en esta caso, los que tienen descendencia sólo tienen un hijo). Vemos que en este nivel hay dos nodos terminales en los que gana MIN, así que los marcamos con -1.
El siguiente nivel inmediatamente superior es el turno de MIN, por lo que seleccionaremos las menores puntuaciones de los nodos hijos. En el estado de la izquierda nos quedamos con -1 que es el menor, en el central con -1 y en el de la derecha con 0.

Finalmente, el nodo raíz es una nodo MAX por lo que seleccionará el mayor de los valores de los nodos hijos, en este caso el 0. Es decir, el siguiente movimiento del ordenador será el que está etiquetado con 0 por lo que pondrá la X en la casilla lateral izquierda de la línea central.

Una vez mueva el jugador humano realizará la misma operación para volver a elegir el siguiente movimiento.

El algoritmo es más o menos como sigue.

```
1. minimax(jugador, tablero)
2.     Si es nodo terminal devolver ganador.
3.
4.     nodos_hijos=todos los movimientos legales desde el estado actual
5.     Si es el turno de MAX
6.         devolver valor máximo de la llamada a minimax() para cada nodo
   hijo.
7.     Sino
8.         devolver valor mínimo de la llamada a minimax() para cada nodo
   hijo.
```

Se trata, como se puede observar de una implementación recursiva de un recorrido en profundidad.

.- No parece un algoritmo muy complicado de implementar -dijo el Sr. Lego, sorprendiéndome de cómo habían evolucionado últimamente sus capacidades. Seguro que podría implementarlo -continuó.

.- Lo animo a usted a intentarlo Sr. Lego.

No habían pasado ni un par de horas cuando el Sr. Lego me presentó el siguiente código en Python.

```
1.  #! /usr/bin/python
2.  import sys
3.
4.  MAX = 1
5.  MIN = -1
6.  global jugada_maquina
7.
8.  def minimax(tablero, jugador):
9.      global jugada_maquina
10.
11.     # Hay ganador o tablas? (nodo terminal)
12.     if game_over(tablero):
13.         return [ganador(tablero), 0]
14.
15.     # generar las posibles jugadas
16.     movimientos=[]
17.     for jugada in range(0,len(tablero)):
18.         if tablero[jugada] == 0:
19.             tableroaux=tablero[:]
20.             tableroaux[jugada] = jugador
21.
22.             puntuacion = minimax(tableroaux, jugador*(-1))
23.             movimientos.append([puntuacion, jugada])
24.
25.
26.     if jugador == MAX:
27.         movimiento = max(movimientos)
28.         jugada_maquina = movimiento[1]
29.         return movimiento
30.     else:
```

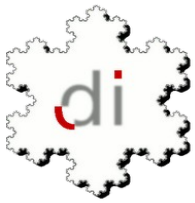
```
31.         movimiento = min(movimientos)
32.         return movimiento[0]
33.
34.
35. def game_over(tablero):
36.     # Hay tablas?
37.     no_tablas = False
38.     for i in range(0,len(tablero)):
39.         if tablero[i] == 0:
40.             no_tablas = True
41.
42.     # Hay ganador?
43.     if ganador(tablero) == 0 and no_tablas:
44.         return False
45.     else:
46.         return True
47.
48.
49. def ganador(tablero):
50.     lineas = [[0,1,2], [3,4,5], [6,7,8], [0,3,6], [1,4,7], [2,5,8],
    [0,4,8], [2,4,6]]
51.     ganador = 0
52.     for linea in lineas:
53.         if tablero[linea[0]] == tablero[linea[1]] and tablero[linea[0]] ==
    tablero[linea[2]] and tablero[linea[0]] != 0:
54.             ganador = tablero[linea[0]]
55.
56.     return ganador
57.
58. def ver_tablero(tablero):
59.     for i in range(0,3):
60.         for j in range(0,3):
61.             if tablero[i*3+j] == MAX:
62.                 print 'X',
63.             elif tablero[i*3+j] == MIN:
64.                 print 'O',
65.             else:
66.                 print '.',
67.
68.         print ''
69.
70.
71. def juega_humano(tablero):
72.     ok=False
73.     while not ok:
74.         casilla = input ("Casilla?")
75.         if str(casilla) in '0123456789' and len(str(casilla)) == 1 and
    tablero[casilla-1] == 0:
76.             if casilla == 0:
77.                 sys.exit(0)
78.             tablero[casilla-1]=MIN
79.             ok=True
80.     return tablero
81.
```

```
82.
83. def juega_ordenador(tablero):
84.     global jugada_maquina
85.     punt = minimax(tablero[:], MAX)
86.     tablero[jugada_maquina] = MAX
87.     return tablero
88.
89.
90. if __name__ == "__main__":
91.     print 'Introduce casilla o 0 para terminar'
92.     tablero = [0,0,0,0,0,0,0,0,0]
93.
94.     while (True):
95.         ver_tablero(tablero)
96.         tablero = juega_humano(tablero)
97.         if game_over(tablero):
98.             break
99.
100.        tablero = juega_ordenador(tablero)
101.        if game_over(tablero):
102.            break
103.
104.    ver_tablero(tablero)
105.    g = ganador(tablero)
106.    if g == 0:
107.        gana = 'Tablas'
108.    elif g == MIN:
109.        gana = 'Jugador'
110.    else:
111.        gana = 'Ordenador'
112.
113.    print 'Ganador:' + gana
```

El Sr. Lego hizo varios intentos infructuosos de ganar al ordenador. Siempre quedaban en tablas.
Sr. Lego, por mucho que lo intente, no conseguirá ganarle nunca -dije.

.- ¿Cómo puede ser eso? ¿quieres decir que si implemento un ajedrez o unas damas usando la estrategia minimax también será invencible?

.- Desgraciadamente no -contesté. En el caso del tic-tac-toe, el tablero de juego es tan pequeño que podemos desplegar el árbol completo de juego, es decir, que el ordenador conoce de antemano todas las posibles jugadas. Con lo cuál es imposible sorprenderlo.
Sin embargo, en juegos como el ajedrez o las damas las posibilidades se disparan exponencialmente. Por ejemplo, si quisiéramos desplegar un juego completo de las damas tendríamos nada más y nada menos que 10^40 nodos



no terminales. Si nuestro ordenador procesara 1000 nodos por segundo tardaría 10000000000000000000000000000000000000 segundos, es decir, 3,171*10^23 millones de años. Teniendo en cuenta que la edad de universo se estima en 15.000 millones de años, si hubiéramos ejecutado nuestro juego el día del Big Bang, ahora mismo no llevaría ni un 1% del tablero completamente escrutado.
.- El sr. Lego me miraba con cara de incredulidad, así que me apresuré a explicarle cómo se utilizaba el algoritmo minimax en estos juegos.

.- Realmente Sr. Lego, hay formas de acelerar el proceso. Uno de los métodos que se utilizan es la poda alpha-beta, pero de esto hablaremos otro día si le apetece.
En la vida real, normalmente no se expanden más de 4 ó 5 niveles del árbol de juego. Los nodos que antes llamábamos terminales ahora los llamaremos nodos frontera, ya que no terminan el juego, pero no podemos permitirnos descender más (Si el juego está muy avanzado sí que podremos encontrar nodos terminales). El peso de la estrategia recae ahora de forma aplastante sobre la función de evaluación. La función de evaluación lo tiene fácil en los nodos terminales, pero en los nodos frontera tiene que ser capaz de evaluar lo bueno que es dicho nodo frontera y asignarle un valor numérico. Para ello se utilizan diferentes heurísticas y dependiendo de lo buena que sean, mejor jugará el programa. Ejemplos de heurísticas para un juego de ajedrez podrían ser: Las piezas que quedan en el tablero, la estructura de los peones, el control de la parte central del tablero, el avance a posiciones cercanas al rey contrario, la posibilidad de enroque, la seguridad del propio rey, etc...

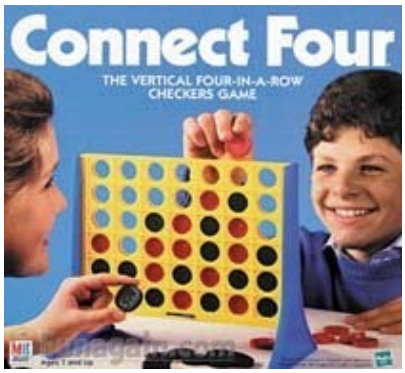
.- Es decir, que en el fondo, los programas de ajedrez actuales no son capaces de ir más que a 4 ó 5 jugadas por delante ¿no?

.- Efectivamente, y por eso, un jugador muy adiestrado puede ganar a un ordenador. Además, en los juegos de ajedrez tradicionales no se suele ir más allá de un par de niveles en el modo fácil.
De todas formas, los programas de ajedrez utilizan una serie de recursos adicionales a las estrategias de búsqueda en árboles, como librerías de aperturas y otros mecanismos basados en sistemas expertos.

El Sr. Lego hizo un par de intentos adicionales de ganar a su propio programa, pero quedó en tablas.

.- Estúpido juego. La única forma de ganar es no jugar -sentenció el Sr. Lego.

8. La máquina invencible II



No es que no valore ciertas virtudes como la tenacidad, pero en el caso del Sr. Lego la cosa va un paso más allá. No es el programador más brillante de la empresa, pero sin duda lo suple sobremanera. Y es que cuando se propone algo, nada ni nadie puede apartarlo de su objetivo. Todo empezó unos días antes cuando tuvimos una charla sobre inteligencia artificial en juegos y el algoritmo minimax. Desde alquel día andaba obsesionado con hacer un juego al que él jugaba de pequeño: El Conecta 4. Un juego de dos jugadores en el que hay que insertar fichas y apilarlas verticalmente para conseguir hacer una línea de 4 fichas del mismo color, ya sea horizontal, vertical o en diagonal.

Recuerdo haber pasado horas en mi infancia jugando a aquél juego, pero creo que el nivel de obsesión del Sr. Lego, en relación al juego, era muy superior al mío.

Usando el algoritmo minimax, el Sr. Lego, había conseguido hacer jugar al ordenador de forma bastante decente, aunque como yo había podido ganarle un par de veces, el quería más. Quería que el programa fuera capaz de ganar siempre.

Sr. Lego, su programa juega bastante bien, le dije. pero si quiere que sea más inteligente, no le queda más remedio que llegar a un nivel más profundo en la exploración del árbol de estado del juego.
El Sr. Lego me miró con cara de frustración y me dijo que lo había intentado, pero que si subía en un nivel la profundidad de la exploración del árbol, el tiempo que tardaba en "pensar" una jugada subía exponenialmente.
Efectivamente Sr. Lego, como ya comentamos cuando hablamos del algoritmo minimax, expandir el árbol más allá de unos pocos niveles no es factible. Aun así, podemos recurrir a algunas técnicas para mejorar el algoritmo, de forma que podamos plantearnos desplegar algún que otro nivel más sin penalización de tiempo.

Al Sr. Lego se le iluminó la cara: ¿Cómo? ¿Cómo puede hacerse eso? La respuesta a sus plegarias se llama poda alfa-beta, Sr. Lego. Además, todavía podemos hacer un poco más óptimo el algoritmo minimax. Fíjese -continué- que cuando usamos el algoritmo minimax, cada vez que el ordenador ensaya una jugada (despliega un nodo hijo en el árbol) tiene que mirar si está en un nivel MIN o en un nivel MAX, y dependiendo de esto elegir el nodo sucesor con mayor o con menor valor. En un juego como el Conecta 4, que tiene un factor de ramificación de 7 implica que si desplegamos el árbol hasta una profundidad de 5 niveles tendremos que hacer esa misma comprobación 7+7^2+7^3+7^4+7^5=19607 veces. Si pudiéramos evitar esta comprobación nos ahorraríamos algunos ciclos de CPU.
- ¿Así podríamos usar ese tiempo para poder descender más niveles? ¿no?

preguntó el Sr. Lego.

Bueno, -contesté- ahorramos algo de tiempo y para despliegues de pocos niveles quizás notemos alguna mejora, pero me temo que esa mejora de tiempo es despreciable en comparación a como crece el tiempo necesario de cálculo para descender un nivel más en el árbol. Aun así, como le explicaré más adelante, nos va a venir bien eliminar esas comprobaciones, ya que nos va a facilitar también la implementación de la poda alfa-beta.
- No se me ocurre como quitar esas comprobaciones, hay que saber siempre si estamos en un nivel MIN o MAX ya que si no, no podremos quedarnos con el valor adecuado.

Le daré una pista Sr. Lego, si se fija un poco, la siguiente igualdad matemática es fácilmente demostrable:

max(x, y) = -min(-x, -y)

Por ejemplo, estará de acuerdo conmigo en que

max(3,5) = -min(-3, -5) = -(-5) = 5

El Sr. Lego no respondió, estaba intentando desentrañar cómo esta igualdad matemática podía mejorar en algo el algoritmo minimax. Le dejé un poco de tiempo para pensar y en pocos segundos chasqueó los dedos y gritó: ¡lo tengo!
Si cada vez que descendemos un nivel en el árbol cambiamos el signo de la mayor de las puntuaciones obtendremos un resultado equivalente al del

algoritmo minimax, sólo que como ahora siempre nos quedamos con el mayor valor de todos los hijos, no hace falta saber en qué nivel estamos, ya que siempre que estemos en un nivel MAX el valor de la función de evaluación será positivo y cuando estemos en un nivel MIN el valor será negativo.

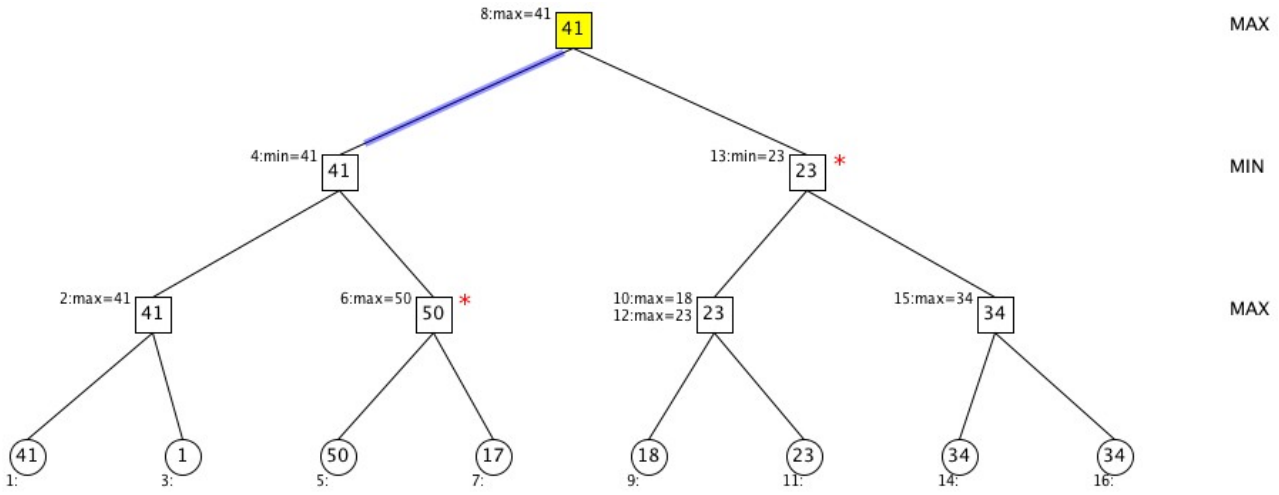
Touché -respondí- acaba de dar en la diana Sr. Lego. De hecho este algoritmo es conocido como negamax, y su pseudocódigo sería el siguiente:

```
1. negamax(jugador, tablero)
2.     Si es nodo terminal o frontera devolver ganador.
3.
4.     nodos_hijos=todos los movimientos legales desde el estado actual
5.         devolver valor máximo de la llamada a -negamax() para cada nodo
   hijo.
```

Como puede observar Sr. Lego, nos ahorramos una comparación y también la selección del máximo o el mínimo (más comparaciones).

- ¿Y en qué consiste esa poda alfa-beta? -preguntó el Sr. Lego visiblemente impaciente.

Vamos a verlo con un sencillo ejemplo. Imagine el siguiente despliegue de un árbol minimax (que como hemos visto es equivalente a un despliegue negamax).



Como puede observar, Sr. Lego, es un despliegue del árbol de juego en el que en los niveles MAX elegimos la mejor puntuación de los nodos hijos y en los nodos MIN, la menor.

Vamos a fijarnos en los dos nodos marcados con un asterisco rojo. Vamos a suponer que aún no los hemos evaluado, es decir, no tienen ningún valor asignado. Empecemos por el primero, el de más a la izquierda: Se trata de un nodo MAX, por lo tanto, su padre es un nodo MIN. Es decir, el padre cogerá el valor menor entre 41 y el que salga de evaluar los dos nodos hijos (50 y 17). Cuando el algoritmo evalúe el primer hijo (50) ya sabemos dos cosas.

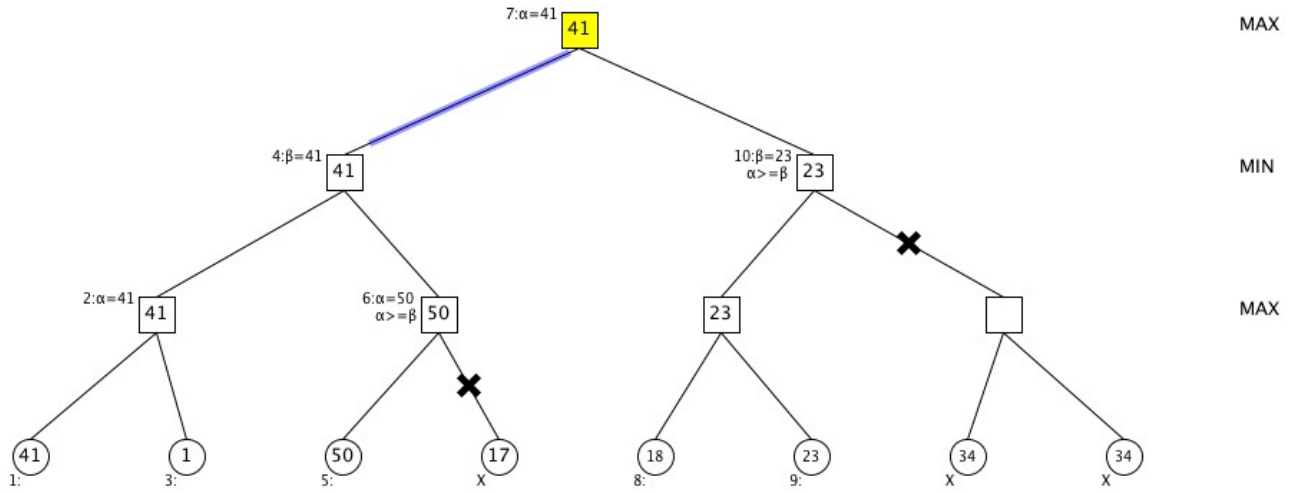
- Como el nodo marcado con el asterisco es un nodo MAX, su valor va a ser como mínimo 50 (si los otros hijos son menores serán descartados por tratarse de un nodo MAX).

- Como el padre del nodo marcado con el asterisco es un nodo MIN se quedará con el menor de los valores de sus hijos, es decir, elegirá entre 41 y (como mínimo) 50. Por lo tanto, no importa el valor del resto de hijos del

nodo marcado con el asterisco, ya que va a elegir el nodo con valor 41 siempre.

Lo mismo sucede con el otro nodo de la derecha. Como es un nodo MIN y el primero de sus hijos tiene un valor de 23, este nodo tendrá como mínimo valor 23 o menor, pero nunca mayor. Como el nodo padre es MAX siempre elegirá al hijo con valor 41 antes que al otro que tendrá 23 o menor. Esto nos permite desechar partes completas del árbol ahorrando el despliegue de un montón de ramas. Es como si las cortáramos, por eso llamamos a este algoritmo poda alfa-beta.

En la siguiente gráfica vemos el resultado de aplicar el algoritmo, señalando con una X las ramas que son podadas.



El algoritmo se llama poda alfa-beta porque vamos a utilizar dos variable (alfa y beta) para recordar y propagar hacia arriba el valor de los nodos. Si en un nodo concreto observamos que el valor de alfa es mayor o igual que el de beta, podamos la rama.

Se estima que este algoritmo mejora en un 30% el rendimiento de minimax o negamax.

El algoritmo es como sigue:

```
1. minimax_alfa_beta(jugador, tablero, alfa, beta)
2. Si es nodo terminal devolver ganador.
3.
4. nodos_hijos=todos los movimientos legales desde el estado actual
5. Si es el turno de MAX
6.     por cada nodo hijo:
7.         puntuacion=minimax_alfa_beta(MIN, tablero_hijo, alfa, beta)
8.         si puntuacion>alfa entonces alfa=puntucaion
9.         si alfa>=beta entonces devolver alfa (poda)
10.
11.    devolver alfa
12. Sino
13.    por cada nodo hijo:
14.        puntuacion=minimax_alfa_beta(MAX, tablero_hijo, alfa, beta)
15.        si puntuacion<beta entonces beta=puntucaion
16.        si alfa>=beta entonces devolver beta (poda)
17.
18.    devolver beta
```

Hay que tener en cuenta que la primera llamada a la función minimax_alfa_beta() hay que hacerla con los valores:
alfa = -infinito

beta = +infinito

Para el algoritmo negamax la cosa es incluso más sencilla, ya que no hay que ir comprobando si es el turno de MIN o de MAX:

```
1. negamax_alfa_beta(jugador, tablero_hijo, alfa, beta)
2.     Si es nodo terminal o frontera devolver ganador.
3.
4.     nodos_hijos=todos los movimientos legales desde el estado actual
5.     por cada nodo hijo:
6.         puntucion = -negamax_alfa_beta(-jugador, tablero, -beta, -alfa)
7.         si puntucion>alfa entonces alfa=puntuacion
8.         si alfa>=beta devolver alfa
```

```
9.
10.     devolver alfa
```

El Sr. Lego siguió todo el día entusiasmado pensando cómo mejorar su juego Conecta 4. Al día siguiente llegó con el siguiente listado en Python y una cara de sueño que indicaba que se había pasado la noche en vela. No sé si implementando el juego o jugando con él.

```
1.  import sys
2.  import copy
3.
4.  MAX = 1
5.  MIN = -1
6.  MAX_PROFUNDIDAD=4
7.
8.  def negamax(tablero, jugador, profundidad, alfa, beta):
9.      max_puntuacion = -sys.maxint-1
10.     alfa_local = alfa
11.     for jugada in range(7):
12.         # columna totalmente llena?
13.         if tablero[0][jugada] == 0:
14.             tableroaux = copy.deepcopy(tablero)
15.             inserta_ficha(tableroaux, jugada, jugador)
16.             if game_over(tableroaux) or profundidad==0:
17.                 return [evalua_jugada(tableroaux, jugador), jugada]
18.             else:
19.                 puntuacion = -negamax(tableroaux, jugador*(-1),
    profundidad-1, -beta, -alfa_local)[0]
20.             if puntuacion>max_puntuacion:
21.                 max_puntuacion = puntuacion
22.                 jugada_max=jugada
23.
24.             # poda alfa beta
25.             if max_puntuacion >= beta:
26.                 break
27.             if max_puntuacion > alfa_local:
28.                 alfa_local = max_puntuacion
29.
30.     return [max_puntuacion, jugada_max]
31.
32.
33. def evalua_jugada(tablero, jugador):
34.     n2=comprueba_linea(tablero, 2, jugador)[1]
35.     n3=comprueba_linea(tablero, 3, jugador)[1]
36.     n4=comprueba_linea(tablero, 4, jugador)[1]
37.     valor_jugada = 4*n2+9*n3+1000*n4
```

```
38.     return valor_jugada
39.
40.
41.
42. def game_over(tablero):
43.     # Hay tablas?
44.     no_tablas = False
45.     for i in range(7):
46.         for j in range(7):
47.             if tablero[i][j] == 0:
48.                 no_tablas = True
49.
50.     # Hay ganador?
51.     if ganador(tablero)[0] == 0 and no_tablas:
52.         return False
53.     else:
54.         return True
55.
56.
57.
58. def comprueba_linea(tablero, n, jugador):
59.     # Comprueba si hay una linea de n fichas
60.     ganador = 0
61.     num_lineas = 0
62.     lineas_posibles=8-n
63.
64.     # Buscar linea horizontal
65.     for i in range(7):
66.         for j in range(lineas_posibles):
67.             cuaterna = tablero[i][j:j+n]
68.             if cuaterna == [tablero[i][j]]*n and tablero[i][j]!=0:
69.                 ganador = tablero[i][j]
70.                 if ganador==jugador:
71.                     num_lineas=num_lineas+1;
72.
73.     # Buscar linea vertical
74.     for i in range(7):
75.         for j in range(lineas_posibles):
76.             cuaterna=[]
77.             for k in range(n):
78.                 cuaterna.append(tablero[j+k][i])
79.             if cuaterna == [tablero[j][i]]*n and tablero[j][i]!=0:
80.                 ganador = tablero[j][i]
81.                 if ganador==jugador:
82.                     num_lineas=num_lineas+1;
83.
84.     # Buscar linea diagonal
85.     for i in range(4):
86.         for j in range(lineas_posibles-i):
87.             cuaterna1=[]
88.             cuaterna2=[]
89.             cuaterna3=[]
90.             cuaterna4=[]
91.
```

```
92.                 for k in range(n):
93.                     cuaterna1.append(tablero[i+j+k][j+k])
94.                     cuaterna2.append(tablero[i+j][i+j+k])
95.                     cuaterna3.append(tablero[i+j+k][6-(j+k)])
96.                     cuaterna4.append(tablero[j+k][6-(i+j+k)])
97.
98.                 if cuaterna1==[cuaterna1[0]]*n and tablero[i+j][j]!=0:
99.                     ganador = tablero[i+j][j]
100.                    if ganador==jugador:
101.                        num_lineas=num_lineas+1;
102.
103.                elif cuaterna2==[cuaterna2[0]]*n and tablero[i+j][i+j]!=0:
104.                    ganador = tablero[i+j][i+j]
105.                    if ganador==jugador:
106.                        num_lineas=num_lineas+1;
107.
108.                elif cuaterna3==[cuaterna3[0]]*n and tablero[i+j][6-j]!=0:
109.                    ganador = tablero[i+j][6-j]
110.                    if ganador==jugador:
111.                        num_lineas=num_lineas+1;
112.
113.                elif cuaterna4==[cuaterna4[0]]*n and tablero[j][6-(i+j)]!=0:

114.                    ganador = tablero[j][6-(i+j)]
115.                    if ganador==jugador:
116.                        num_lineas=num_lineas+1;
117.
118.    return (ganador, num_lineas)
119.
120.
121.
122.def ganador(tablero):
123.    return comprueba_linea(tablero, 4, 0)
124.
125.
126.
127.def ver_tablero(tablero):
128.    for i in range(7):
129.        for j in range(7):
130.            if tablero[i][j] == MAX:
131.                print 'X',
132.            elif tablero[i][j] == MIN:
133.                print 'O',
134.            else:
135.                print '.',
136.        print ''
137.    print '-------------'
138.    print '1 2 3 4 5 6 7'
139.
140.
141.
142.def inserta_ficha(tablero, columna, jugador):
143.    # encontrar la primera casilla libre en la columna
144.    # y colocar la ficha
```

```
145.     ok = False
146.     for i in range(6,-1, -1):
147.         if (tablero[i][columna] == 0):
148.             tablero[i][columna]=jugador
149.             ok=True
150.             break
151.
152.     return ok
153.
154.
155.
156. def juega_humano(tablero):
157.     ok=False
158.     while not ok:
159.         col = input ("Columna (0=salir)?")
160.         if str(col) in '01234567' and len(str(col)) == 1:
161.             if col==0:
162.                 sys.exit(0)
163.             ok=inserta_ficha(tablero, col-1, MIN)
164.         if ok == False:
165.             print "Movimiento ilegal"
166.
167.     return tablero
168.
169.
170.
171. def juega_ordenador(tablero):
172.     tablerotmp=copy.deepcopy(tablero)
173.     punt, jugada = negamax(tablerotmp, MAX, MAX_PROFUNDIDAD, -sys.maxint-
    1, sys.maxint)
174.     inserta_ficha(tablero, jugada, MAX)
175.
176.     return tablero
177.
178.
179.
180. if __name__ == "__main__":
181.     # tablero de 7x7
182.     tablero = [[0 for j in range(7)] for i in range(7)]
183.
184.     ok=False
185.     profundidades=[3, 4, 6]
186.     while not ok:
187.         dificultad = input ("Dificulatad (1=facil; 2=medio; 3=dificil): ")
188.         if str(dificultad) in '123' and len(str(dificultad)) == 1:
189.             MAX_PROFUNDIDAD=profundidades[dificultad-1]
190.             ok=True
191.
192.
193.     while (True):
194.         ver_tablero(tablero)
195.         tablero = juega_humano(tablero)
196.         if game_over(tablero):
197.             break
```

```
198.
199.         tablero = juega_ordenador(tablero)
200.         if game_over(tablero):
201.             break
202.
203.     ver_tablero(tablero)
204.
205.     g = ganador(tablero)[0]
206.     if g == 0:
207.         gana = 'Tablas'
208.     elif g == MIN:
209.         gana = 'Jugador'
210.     else:
211.         gana = 'Ordenador'
212.
213.
214.     print 'Ganador:' + gana
```

9. ¿Sueñan los ordenadores con imitar a Cervantes?

Cualquier día es bueno, y más si la climatología acompaña, para dar un paseo por el parque. Aquel día coincidía con una feria del libro, y mi amigo Descollante y yo andábamos de caseta en caseta en busca de algún buen libro extranjero de informática (porque de los autóctonos mejor ni hablamos). En una de las casetas estaba Fulano sentado, firmando libros a troche y moche, el último fabricante de bestsellers en serie. A mí no me parece mal siempre que ayude a que algunos lean algo, aunque sea por poder decir que se han leído el último libro de moda, pero mi amigo Descollante suele ser algo más radical e intolerante con la literatura barata y de consumo, y más si mediocres escritores que aprovechan filón literario del último tema de moda, junto con un marketing estudiadísimo, acaban forrándose de la noche a la mañana. Quizás influya que Descollante lleva dentro un escritor frustrado y un montón de negativas de editoriales a sus espaldas. Y no es que sea Borges o Delibes, pero tampoco se le da mal eso de contar historias dándole a la tecla.

Para mí que Fulano tiene un grupo de monos escribiéndole las cosas esas que publica -comenzó a decir Descollante- Seguro que tiene un programa de

ordenador que le escribe los libros.

Si fueras capaz de hacer un programa para escribir bestsellers seguro que te forrabas más que Fulano con sus libros -bromeé.

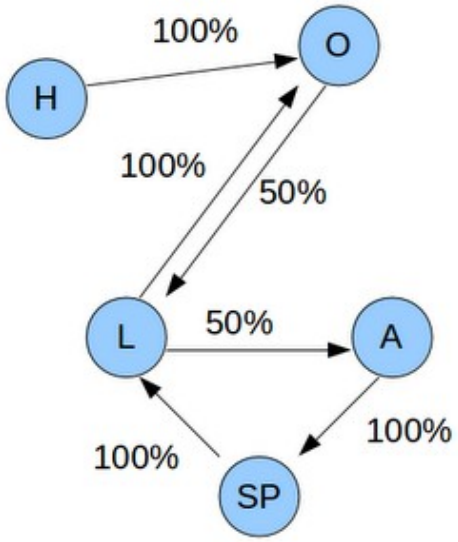
Descollante me miró con media sonrisa y me dijo que quizás no estábamos tan lejos de conseguir que los ordenadores fueran capaces de hacer literatura. No pude esconder mi sonrisa al pensar que ya estaba fanfarroneando de nuevo. Tuvo que captar mi pensamiento porque se paró en seco frente a una terraza-bar y se sentó en una silla sin ni siquiera avisar. Antes de que pudiera reaccionar había pedido dos Cocacolas. No hace mucho estuve "jugando" con cadenas de Markov para tratar de generar texto con cierto sentido. Y no fue mal el experimento -comenzó a decirme Descollante.

¿Cadenas de Markov? me suena de la facultad -le contesté- pero no recuerdo muy bien en qué consistían.

Descollante sonrió y empezó a explicarme que las cadenas de Markov son una estructura matemática cuya definición formal es algo compleja, aburrida y además requiere sólidos conocimientos de Probabilidad. Así que, saltándose toda la parte aburrida, me explicó que una cadena de Markov es básicamente un grafo dirigido (un digrafo realmente) cuyos nodos están conectados por aristas, y a cada arista se le asocia un valor que es la probabilidad que hay de pasar de un nodo a otro.

La verdad es que me he quedado un poco a cuadros -le dije. Es muy sencillo, Alberto. Imagina que le mostramos a un ordenador el siguiente texto: "Hola Lola". Y queremos construir una cadena de Markov que nos indique qué probabilidad hay, en esta frase, de pasar de una letra a

otra. O mejor dicho de que a una letra le siga otra concreta. Por ejemplo, ¿qué probabilidad hay de que a la letra H le siga una O? Teniendo en cuenta que las letras posibles son sólo H, O, L, A y espacio (SP), ya que no distinguimos aquí entre mayúsculas y minúsculas. Una cadena de Markov para esta frase tendría el siguiente aspecto.



Según el grafo, la probabilidad de que tras una L aparezca una A es del 50%. Aunque para trabajar con estos porcentajes mejor diremos que la probabilidad es de 0.5, es decir, la probabilidad será un valor real entre 0 y 1. La suma de las probabilidades de las aristas salientes de un nodo siempre tiene que valer 1.

Como ya habrás deducido -me dijo Descollante por sorpresa- el cambio de un estado a otro es una función de probabilidad que depende exclusivamente del estado anterior. Por lo tanto, aquello que haya ocurrido antes del estado anterior no tiene relevancia.

Vale, le dije. Y... ¿cómo se supone que esto va a escribir un libro?

Descollante empezó a reír mientras sacaba su portátil. Mira -me dijo- esto es una prueba que hice el otro día.

Me enseño el siguiente listado.

```
1.  import random
2.
3.  def create_char_map(filepath):
4.    chars={}
5.    f = open(filepath, 'r')
6.    done=False
7.    lastchar=''
8.    while not done:
9.      t=f.readline()
10.     if len(t)==0:
11.       done=True
12.     else:
13.       for c in t:
14.         if not c in chars:
15.           chars[c]={}
16.         if len(lastchar)>0:
17.           if c in chars[lastchar]:
18.             chars[lastchar][c]=chars[lastchar][c]+1.0
19.           else:
20.             chars[lastchar][c]=1.0
21.         lastchar=c
22.   f.close()
23.
24.   return chars
25.
26.
27.
28. def markov_net(chars):
29.   for c in chars:
30.     tot=0
31.     for n in chars[c]:
32.       tot=tot+chars[c][n]
33.
34.     for n in chars[c]:
35.       chars[c][n]=chars[c][n]/tot
36.
37.   return chars
38.
39.
40. def generate_text(chars, initial_char, num_chars):
41.   text=''
42.   ch=initial_char
43.   for c in range(num_chars-1):
44.     text += ch
45.     ch=prob_choice(chars[ch])
46.   return text
47.
```

```
48.
49. def prob_choice(prob_chars):
50.   probability = []
51.   char = []
52.   cur_prob = 0.0
53.
54.   for c, p in prob_chars.iteritems():
55.     cur_prob = cur_prob + p
56.     probability.append(cur_prob)
57.     char.append(c)
58.
59.   rnd = random.random() * cur_prob
60.   for i, total in enumerate(probability):
61.     if rnd < total:
62.       return char[i]
63.
64.
65. if __name__ == "__main__":
66.   # cambiar alicia.txt por el nombre del
67.   # archivo que se quiera utilizar.
68.   chars = create_char_map('alicia.txt')
69.   chars = markov_net(chars)
70.   text = generate_text(chars, 'd', 1000)
71.   print text
```

Este programita lee un texto (preferiblemente grande) y construye una malla de probabilidades como esta, pero evidentemente mucho más grande para albergar todas las letras del alfabeto y los signos de puntuación.

	H	*O*	*L*	*A*	*sp*
H	0	0	0	0	0
O	1	0	0.5	0	0
L	0	1	0	0	1
A	0	0	0.5	0	0
sp	0	0	0	1	0

La idea es que a partir de esta malla de probabilidades podemos hacer que el ordenador genere un texto sin sentido, pero con silabas habituales del idioma. Algo así como un lenguaje inventado.

Tras ejecutarlo usando el texto del libro Alicia en el país de las maravillas obtuvimos el siguiente texto:

```
de domal ajaterébobasta sesude co taditegabocudes a ni Ala y desuiñose os e
itadonco ladezaden mo do paribl dace senanoca ca lie e mel M, diciontando
renastuefa seloren coso llomara br Fun sero pre ire do-. atis Resties tun
magunande la e sapu sa gils dedonco do acaren dontiga ventrtijo ta e padele
aitoja vor dicancora-Nos y qur a puema dicienchadante deca la m!
```

Pues no creo que esto se venda más que los libros de Fulanito -dije riendo. Aunque la verdad es que parecen palabras inventadas de un idioma de Tolkien. Descollante ejecutó el programa varias veces más obteniendo diferentes resultados.

Bueno, todavía podemos mejorarlo un poco -dijo. En vez de construir una malla de probabilidades con las letras del alfabeto podemos hacerlo directamente con palabras completas.

¿Quieres decir que habría que construir una malla en la que, dada un palabra, contenga la probabilidad con la que aparecerá la siguiente palabra?

Efectivamente -respondió. Por cada palabra del texto analizado incluiremos todas las palabras que en algún momento del texto la sigan y con qué frecuencia, por ejemplo, en la frase "Soy grande porque no soy pequeño" vemos que tras la palabra "soy" aparecen las palabras "grande" y "pequeño", por lo tanto tras la palabra "soy" hay una probabilidad de 0.5 de que aparezca "grande" y 0.5 de que aparezca la palabra "pequeño". Descollante se puso a teclear y cambió el programa que acabó teniendo el siguiente aspecto:

```
1. import random
2.
3. def create_word_map(filepath):
4.   words={}
5.   f = open(filepath, 'r')
```

```
6.    done=False
7.    lastword=''
8.    while not done:
9.      t=f.readline()
10.     if len(t)==0:
11.       done=True
12.     else:
13.       for c in t.split(' '):
14.         if not c in words:
15.           words[c]={}
16.         if len(lastword)>0:
17.           if c in words[lastword]:
18.             words[lastword][c]=words[lastword][c]+1.0
19.           else:
20.             words[lastword][c]=1.0
21.         lastword=c
22.   f.close()
23.   return words
24.
25.
26.
27. def markov_net(chars):
28.   for c in chars:
29.     tot=0
30.     for n in chars[c]:
31.       tot=tot+chars[c][n]
32.
33.     for n in chars[c]:
34.       chars[c][n]=chars[c][n]/tot
35.
36.   return chars
37.
38.
39. def generate_text(chars, initial_char, num_chars):
40.   text=''
41.   ch=initial_char
42.   for c in range(num_chars-1):
43.     text += ' '+ch
44.     ch=prob_choice(chars[ch])
45.   return text
46.
47.
48. def prob_choice(prob_chars):
49.   probability = []
50.   char = []
51.   cur_prob = 0.0
52.
53.   for c, p in prob_chars.iteritems():
54.     cur_prob = cur_prob + p
55.     probability.append(cur_prob)
56.     char.append(c)
57.
58.   rnd = random.random() * cur_prob
59.   for i, total in enumerate(probability):
```

```
60.      if rnd < total:
61.        return char[i]
62.
63.
64. if __name__ == "__main__":
65.    chars = create_word_map('alicia.txt')
66.    chars = markov_net(chars)
67.    text = generate_text(chars, 'La', 1000)
68.    print text
```

Descollante ejecutó el programa de nuevo usando el texto de Alicia en el país de las maravillas y este fue el texto obtenido.

```
El Lirón se llevó a la boca, mamá! -protestó Alicia quedó muy soliviantadas.

 -¡Ratoncito querido! ¡vuelve atrás, que los pájaros se me suene de estas
      palabras, su libro de las palabras:

 Cuando volvió hacia arriba, pero no es muy cómodo, y después una de debajo de
      extrañar que vio al cuerpo, y a suceder en que comer y seguía achicándose
      rápidamente. Se está del jardín, y Morcaro, duques de haber muchas ganas
      de manzanas, con el suelo su alrededor hacia el aire. Incluso la dirección
      en esto, pero no lo que no se interrumpió, y tan terrible entre los
      pececillos
 Para esto no hacer esta mañana -dijo Alicia.

 -¡Yo no quiere decir! -refunfuñó el Grifo.

 Alicia empezaba a oír esto no había largado al principio -dijo Alicia-, qué
      tipo de que ir a
```

Curioso -dije- parece un texto escrito por un paranoico, pero algunas frases tienen sentido y todo. Parece interesante esto de las cadenas de Markov. Pues sí -dijo Descollante. Las cadenas de Markov se aplican en gran cantidad de campos: en problemas de termodinámica y física estadística, para crear modelos meteorológicos, en simulación, juegos de azar, economía y predicción de valores de bolsa, incluso el algoritmo page rank de google hace uso de él.

10. La leyenda de los “predictores” perfectos

En una mañana cualquiera, aún no habría podido ponerme a trabajar en serio, pero aquél día sólo llevaba media hora sentado en mi mesa y había podido leer todos los emails pendientes, contestarlos, planificarme la jornada e incluso había empezado a trabajar en el código del programa que tenía entre manos. Un día normal mi compañera Sofía me habría entretenido contándome alguna batallita nocturna tras haber derribado alguna web "enemiga" (defacement lo llama ella), pero esa mañana estaba especialmente silenciosa y, pese a ir de alternativa por la vida, ahí estaba con su ipod enchufado a sus orejas. Gaznápiro, mi jefe, había pasado por delante de nuestras mesas como una exhalación, con la cabeza gacha y con un montón de papeles bajo el brazo. Mal presagio sin duda. Es la calma que precede a la tempestad. Cuando Gaznápiro llega tan temprano al trabajo acompañado de tantos papeles (ya podría comprarse una carpeta el hombre) es señal de que tiene que preparar una reunión con la plana mayor, de la que suele salir bastante cabreado. Evidentemente, tiene que desfogar con alguien y parece que, además de informáticos, somos buenos esparrings, porque nada más salir de esas reuniones la emprende, metafóricamente hablando, con nosotros.

Pero lo más extraño de todo era que el Sr. Lego aún no había aparecido por allí, y eso que suele ser el primero en llegar cada mañana. Puntual como un reloj suizo. Fue pensar en ello y aparecer por la puerta de la oficina. Traía el hombre mala cara esa mañana y unas ojeras bastante pronunciadas.
Buenos días Sr. Lego ¿mala noche? -dije.

Pues sí -respondió- he dormido regular. El crío, que anda con dolores de estómago y se pasa la noche llora que te llora, el pobre. Y encima escuchando la radio en el coche me entero de que han bajado las acciones de Microsol, justo ahora que acababa de comprar algunas por probar esto de la bolsa. Afortunadamente no invertí mucho, pero mira que tengo mala pata. Ojalá hubiera algún programa que te dijera en qué empresa hay que invertir para ganar dinero.

¡Ojalá! -dije. Pero me temo que todavía no lo han inventado. Aun así hay bastante gente investigando en ese campo, y por lo que sé, algunos programas no lo hacen mucho peor que brokers profesionales.
Evidentemente, acababa de captar la atención del Sr. Lego porque había dejado de mirar los valóres de bolsa para mirarme a mí.
Bueno -continué- el otro día leí algo sobre que habían desarrollado un programa que usando algoritmos genéticos era capaz de hacer mejor papel que la mayoría de los inversores humanos en el mercado de futuros.
¿Algoritmos genéticos? -preguntó el Sr. Lego.

Sí, es una de las ramas de la Inteligencia Artificial que se está explorando con más interés en la actualidad, y además tiene un campo de aplicación bastante amplio.

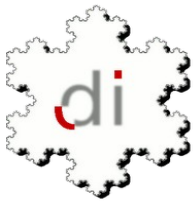
Pues no tenía ni idea -reconoció el Sr. Lego. ¿Cómo funcionan los algoritmos

genéticos?

Verá usted Sr. Lego, no es fácil explicar cómo funcionan en pocas palabras, pero le voy a contar la leyenda de los "predictores" perfectos y verá como lo entiende rápidamente.

Cuenta la leyenda que en un mundo paralelo al nuestro vivían unos seres llamados predictores. En ese mundo todo era digital y estaba formado por ceros y unos. Todo se comportaba de forma binaria. Encendido/apagado, activo/inactivo, abierto/cerrado, etc… Los predictores no eran una excepción, y se alimentaban de cifras binarias. La comida de los predictores llegaba en forma de impulsos desde no se sabe dónde, el caso es que cada segundo llegaba un impulso que los predictores debían asimilar y digerir. Esos impulsos podían corresponder a un cero o un uno y siempre llegaban formando el mismo patrón que se repetía una y otra vez: 1101001011. El Sr. Lego me estaba mirando con cara de incredulidad como pensando "No puede estar contándome semejante tontería", así que me apresuré a continuar antes de que me cortara en seco.

El problema es que los predictores necesitaban emitir una señal idéntica a la que recibían para poder digerirla. Los predictores emitían cada segundo, de forma sincronizada con la llegada de comida, una señal. La señal que emitían los predictores estaba codificada en su ADN en forma de secuencia fija de 10 bits. Cada predictor tenía una secuencia diferente, por lo que algunos, dependiendo del grado de similitud entre la secuencia de su ADN y la del patrón de llegada de comida se alimentaban mejor que otros. Es decir, si un predictor tenía la secuencia 1001011001 en su ADN (recordemos que el patrón de llegada de alimentos es 1101001011), este



comerá 7 de cada 10 veces que recibe comida, ya que hay 3 cifras que son diferentes entre los dos patrones.

O sea, que habrá predictores que estén mejor alimentados que otros ¿no? -dijo el Sr. Lego.

Efectivamente, son predictores mejor adaptados al mundo y, por lo tanto, con mayor probabilidad de sobrevivir y procrear.

El Sr. Lego sonrió: Así que los predictores también tienen hijos. Realmente, Sr. Lego, son seres hermafroditas y muy promiscuos. Cada pocos segundos sienten la necesidad de unirse a otro predictor. Cada vez que dos predictores tienen relaciones tienen exactamente dos hijos (uno cada uno). Afortunadamente, cada vez que nace un predictor otro se desintegra y de esta forma la población se mantiene estable.

Qué bonita historia de Amor… -se mofó el Sr. Lego.

Verá Sr. Lego, lo más impresionante de esta historia es que tras algunas generaciones de predictores, el ADN de la mayoría de la población era capaz de predecir al 100% el patrón de llegada de alimento. Un hecho increíble. La población entera había evolucionado en su conjunto para adaptarse al medio. Es por ello que llamamos a esta historia la leyenda de los predictores perfectos.

Vaya, esto se parece bastante a la selección natural de la que hablaba Darwin -dijo el Sr. Lego. Sobreviven los más adaptados y con el tiempo los peor adaptados desaparecen.

Efectivamente, pero ¿sabe cuál es el mecanismo que consiguió que toda la

población convergiera hacia un ADN perfecto? El secreto estaba en que los mejores adaptados también tenían más posibilidades de reproducirse y, por ende, de transmitir su ADN de mejor calidad.

Como la vida misma -bromeó el Sr. Lego. Lo que no entiendo es qué tiene que ver todo esto con un algoritmo capaz de predecir el comportamiento de la bolsa.

Bueno Sr. Lego, una vez establecida la analogía con el mundo de los predictores vamos a entrar en materia y a ver como funciona un algoritmo genético o evolutivo, como prefiero llamarlo.

En una población completa, llamaremos individuo a cada uno de sus componentes. Al ADN de cada predictor lo llamaremos gen y a cada bit que lo compone lo llamaremos cromosoma.

La población inicial tiene que ser lo suficientemente grande como para tener cierta riqueza genética, así que lo primero que hay que hacer es generar una población grande y con ADNs aleatorios.

Todo algoritmo genético consta de cuatro fases: Selección, cruce, mutación y extinción.

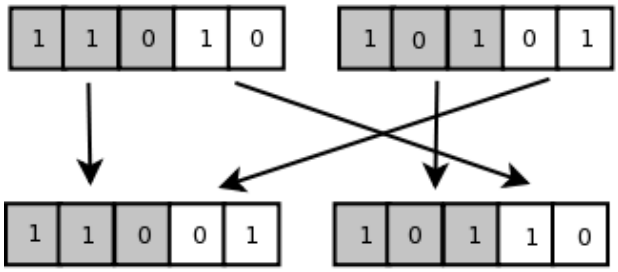
La selección consiste en escoger los mejores individuos para que procreen. Sin embargo, en pro de la diversidad genética no conviene quedarse estrictamente con los mejores. Es preferible asignar una probabilidad de ser seleccionado a cada individuo según su nivel de adaptación. Por ejemplo, un predictor que acierta 8 veces con su secuencia de ADN tiene que tener un 50% más de probabilidad de ser seleccionado para procrear que un

predictor con 4 aciertos.

En cualquier caso, un predictor con mal ADN tiene que tener alguna probabilidad de procrear, aunque sea baja, ya que uno de sus hijos podría aportar una gran mejora a la población.

El nivel de adaptación al medio de cada predictor nos lo dará la llamada función de adaptación, que es una función que nos indica cómo de bien adaptado está el individuo. En el caso de los predictores, la función de adaptación nos devolvería el número de bits del ADN del predictor que coinciden con el patrón de llegada de comida.

En la fase de cruce es donde procrean los individuos seleccionados dando lugar a dos hijos. Se llama fase de cruce porque se seleccionan dos individuos y se entrecruzan sus ADNs. El proceso consiste en seleccionar de forma aleatoria un punto de corte del ADN y recombinarlos según el siguiente esquema. Supongamos que el punto de corte ha sido el tercer cromosoma.



Es decir, que del cruce de los padres (arriba) obtenemos dos nuevos hijos (abajo).

La tercera fase, llamada mutación, imita las mutaciones genéticas

biológicas. Estas mutaciones no deben ser muy frecuentes (sobre un 1% de probabilidad de que se produzca, aunque se puede jugar con este valor). La mutación consiste en cambiar uno de los cromosomas de forma aleatoria en un gen, es decir, seleccionamos un cromosoma aleatoriamente y si es un cero lo cambiamos a un uno y si es un uno lo transformamos en un cero.

Finalmente nos queda la fase de extinción, donde elegiremos a los dos individuos peor adaptados para eliminarlos.

Todo el proceso se repite hasta que, o bien la población converge (aproximadamente un 95% de la población tiene un valor similar en la función de adaptación) o bien se alcanza un número máximo de iteraciones.

Este sería, grosso modo, el pseudocódigo de un algoritmo genético:

```
generar población inicial de forma aleatoria
  Mientras(no fin)
   Para tamaño(población)/2 hacer
      seleccionar dos individuos
      cruzar los dos individuos
      mutar con cierta probabilidad los nuevos individuos
      añadir los dos nuevos individuos a la población
      eliminar los dos peores individuos de la población
    fin para
  fin mientras
quedarse con la mejor solución
```

El señor lego estaba entusiasmado. Parecía haberlo entendido y ya estaba pensando en construir su propio modelo del mundo de los predictores. No tardó más de un par de horas en enviarme el siguiente código.

```
import math
import random

def initial_population(max_population, num_vars):
```

```
5.    # crear población inicial aleatoria
6.    population=[]
7.    for i in range(max_population):
8.      gene=[]
9.      for j in range(num_vars):
10.       if random.random()>0.5:
11.         gene.append(1)
12.       else:
13.         gene.append(0)
14.     population.append(gene[:])
15.   return population
16.
17. def adaptation_function(gene, solution):
18.   # valor de adaptación del gen
19.   cont=0
20.   for i in range(len(gene)):
21.     if (gene[i]==solution[i]):
22.       cont=cont+1
23.   return cont
24.
25.
26. def evaluate_population(population, solution):
27.   # evalua todos los genes de la poblacion
28.   adaptation=[]
29.   for i in range(len(population)):
30.     adaptation.append(adaptation_function(population[i], solution))
31.   return adaptation
32.
33.
34. def selection(population, solution):
35.   adaptation=evaluate_population(population, solution)
36.   # suma de todas las puntuaciones
37.   total=0
38.   for i in range(len(adaptation)):
39.     total=total+adaptation[i]
40.   # escogemos dos elementos
41.   val1=random.randint(0,total)
42.   val2=random.randint(0,total)
43.   sum_sel=0
44.   for i in range(len(adaptation)):
45.     sum_sel=sum_sel+adaptation[i]
46.     if sum_sel>=val1:
47.       gene1=population[i]
48.       break
49.   sum_sel=0
50.   for i in range(len(adaptation)):
51.     sum_sel=sum_sel+adaptation[i]
52.     if sum_sel>=val2:
53.       gene2=population[i]
54.       break
55.   return gene1, gene2
56.
```

```
57. def crossover(gene1, gene2):
58.   # cruza 2 genes y obtiene 2 descendientes
59.   new_gene1=[]
60.   new_gene2=[]
61.   cutpoint=random.randint(0, len(gene1))
62.   new_gene1[0:cutpoint]=gene1[0:cutpoint]
63.   new_gene1[cutpoint:]=gene2[cutpoint:]
64.   new_gene2[0:cutpoint]=gene2[0:cutpoint]
65.   new_gene2[cutpoint:]=gene1[cutpoint:]
66.   return new_gene1, new_gene2
67.
68. def mutation(prob, gene):
69.   # muta un gen con una probabilidad prob.
70.   if random.random<prob:
71.     chromosome=random.randint(0, len(gene))
72.     if gene[chromosome]==0:
73.       gene[chromosome]=1
74.     else:
75.       gene[chromosome]=0
76.   return gene
77.
78. def drop_worse_genes(population, solution):
79.   # elimina los dos peores genes
80.   adaptation=evaluate_population(population, solution)
81.   i=adaptation.index(min(adaptation))
82.   del population[i]
83.   del adaptation[i]
84.   i=adaptation.index(min(adaptation))
85.   del population[i]
86.   del adaptation[i]
87.
88. def best_gene(population, solution):
89.   # devuelve el mejor gen de la poblacion
90.   adaptation=evaluate_population(population, solution)
91.   return population[adaptation.index(max(adaptation))]
92.
93. def genetic_algorithm():
94.   max_iter=10
95.   max_population=50
96.   num_vars=10
97.   done=False
98.   solution = initial_population(1, num_vars)[0]
99.   population = initial_population(max_population, num_vars)
100.
101.  iterations=0
102.  while not done:
103.    iterations=iterations+1
104.    for i in range((len(population))/2):
105.      gene1, gene2 = selection(population, solution)
106.      new_gene1, new_gene2 = crossover(gene1, gene2)
107.      new_gene1 = mutation(0.1, new_gene1)
108.      new_gene2 = mutation(0.1, new_gene2)
109.      population.append(new_gene1)
```

```
110.       population.append(new_gene2)
111.       drop_worse_genes(population, solution)
112.
113.     if (max_iter<iterations):
114.       done=True
115.
116.   print "Solucion: "+str(solution)
117.   best = best_gene(population, solution)
118.   return best, adaptation_function(best, solution)
119.
120.
121.
122.if __name__ == "__main__":
123.   random.seed()
124.   best_gene = genetic_algorithm()
125.   print "Mejor gen encontrado: "+ str(best_gene[0])
126.   print "Funcion de adaptacion: "+str(best_gene[1])
```

Excelente Sr. Lego, veo que ha captado perfectamente la idea. De hecho sus predictores se convierten en predictores perfectos tras muy pocas iteraciones. ¡Enhorabuena!

Sí, y me parece que los predictores están muy felices, pero ¿valen para algo más los algoritmos genéticos?

Pues valen para muchas cosas. Son de hecho una metaheurística usada en búsquedas locales.

¿cómo? ¿metaqué? -preguntó el Sr. Lego.

Metaheurística. Es una forma de búsqueda heurística que podemos adaptar a casi cualquier problema que no tenga un algoritmo propio que lo resuelva. Una especie de comodín. Si no sabemos cómo afrontar la solución de un problema podemos usar una metaheurística como pueda ser un algoritmo genético.

Nos vale para problemas de optimización, diseño automático, aprendizaje, predicción, etc…

De hecho, si podemos representar el problema como ristras de números binarios, podemos usar el programa que acaba de hacer para resolverlo. Por ejemplo, si quisiéramos encontrar un valor entero para x que maximice la función f(x)=x^2, sólo tendríamos que crear una población de números binarios, que representarían a sus contrapartidas enteras, y usar como función de adaptación el valor de la propia función X^2 al evaluar el número representado por el ADN. En pocas iteraciones obtendríamos una población que convergería hacia ese valor de X que maximiza la función. Interesante -dijo el señor Lego. ¿Pero sólo podemos tener genes cuyos cromosomas son ceros y unos?

Para nada -contesté. En el campo de la investigación suelen representarse así ya que con números binarios podemos codificar casi cualquier valor. De hecho, los cromosomas podrían ser cualquier cosa: números, letras, etc… Eso sí, teniendo en cuenta que hay que elegir una función de adaptación acorde a la representación que usemos para la información. Bueno, dijo el Sr. Lego. Voy a darle algunas vueltas a todo esto que me has contado para ver como hago un predictor perfecto para los valores de la bolsa.

Desafortunadamente (o afortunadamente) la bolsa no sigue patrones fijos, al menos aparentemente, aunque quizás dé usted con un predictor bursatil que le saque de esta empresa.

Ojalá, contestó el Sr. Lego. Aunque echaría de menos nuestras conversaciones.

11. La máquina predictora de estados de ánimo

No es ningún secreto que un tonto "leal" tiene más posibilidades de ascender en una empresa que un listo librepensador, pero si el tonto, además, disfruta gritando a los demás, sus posibilidades de ascender rozan el 100%. En eso andaba Gaznápiro, entrenando sus mejores gritos con el Sr. Lego dentro de su despacho (su faceta de tonto no necesitaba entrenarla más porque, probablemente, había llegado ya a lo más alto que nadie pueda llegar). Los gritos podían oírse en toda la oficina, entre nasales y con un tono agudo que daban más risa que miedo, y que agudizaban, aún más si cabía, la condición de tonto "leal" que le hace el trabajo sucio al gran jefe. El Sr. Lego salió del despacho con la cabeza baja y maldiciendo hasta en arameo antiguo. Sofía, que se sentaba en la mesa contigua a la mía no paraba de anotar cosas en una libreta pequeña de color rojo, de esas de anillas en la parte de arriba.

¿Qué anotas con tanto interés? -pregunté a Sofía intentando esconder mi curiosidad.

Anoto el estado de ánimo que tiene Gaznápiro. Lo hago desde hace nueve días. -respondió.

Me quedé mirandola un rato esperando que me dijera que estaba de broma, pero no lo hizo. En vez de eso, prosiguió contándome detalles. Intento encontrar un patrón de comportamento para ver si descubro qué es lo que hace que tenga un día de perros o un día de buen humor. Así sabré cuando pedir un ascenso o un día de vacaciones. Anoto lo que ha desayunado, el color de la corbata, si toma o no café… en fin todo lo que pueda observar lo anoto.

Lo peor de todo es que, conociendo a Sofía, lo estaba diciéndolo totalmente en serio.

El Sr. Lego, que andaba pegando el oído preguntó a Sofía si podía decirle qué tal día iba a tener mañana, ya que tocaba reunión de proyecto y andaba algo retrasado con la programación.

Bueno, -respondió Sofía- desgraciadamente aún no he encontrado ningún patrón.
Ambos me miraron como esperando que me sacara un as de la manga. Venga Alberto, seguro que conoces algún truco para convertir todos estos datos en algo utilizable -dijo el Sr. Lego dándome palmaditas en la espalda y sonriendo como si me hubiera pillado en un renuncio.

Claro, si la información que ha anotado Sofía es la mitad de precisa de lo que esperaría de ella podemos utilizar un clasificador bayesiano para modelar el comportamiento de Gaznápiro e intentar predecir con qué estado de ánimo llegará mañana a la oficina.

Ahora eran Sofía y el Sr. Lego los que me miraban a mí como si fuera un

bicho raro, lo que me hizo anotar mentalmente: "No volver a mirar a mis compañeros como si se hubieran escapado de un manicomio".

Usando los datos que había tomado Sofía obtuvimos la siguiente tabla en la que anotamos el color de la corbata, lo que había comido en el desayuno y lo que había bebido. En la última columna anotamos si ese día había estado o no de buen humor.

día	Corbata	Desayuno	Bebida	Humor
1	Roja	Pan	Café	Malo
2	Roja	Pan	Zumo	Malo
3	Roja	Fruta	Café	Bueno
4	Amarilla	Pan	Café	Malo
5	Amarilla	Fruta	Café	Bueno
6	Azul	Pan	Zumo	Bueno
7	Azul	Fruta	Café	Malo
8	Azul	Fruta	Zumo	Bueno
9	Azul	Pan	Café	Malo

Un clasificador bayesiano -comencé a explicar- selecciona la mejor clasificación según los atributos de los que disponemos. En nuestro caso, las clasificaciones posibles son Bueno o Malo, que describen el estado de humor de Gaznápiro. Nuestros atributos son: Corbata, Desayuno y Bebida, que pueden tomar los siguientes valores:

Corbata={Roja | Amarilla | Azul}

Desayuno={Pan | Fruta}

Bebida={Café | Zumo}

Espero que recordéis bien el teorema de Bayes, porque lo que sigue se basa en él. No es por casualidad que se llame clasificador bayesiano.

Nuestro clasificador va a basarse en la siguiente expresión:

$$P=\operatorname{argmax} P(c)\prod_i P(a_i|c)$$

Como sabéis P(c) es la probabilidad de que ocurra la clasificación c, siendo c={Bueno | Malo}. Por otro lado, $P(a_i|c)$ es lo que llamamos probabilidad condicionada de a sabiendo que ocurre c. Por ejemplo, P(Azul|Bueno) es la probabilidad de que Gaznápiro lleve corbata azul sabiendo que hoy tiene buen humor.

Para calcular $P(a_i|c)$ vamos a usar la siguiente fórmula:

$$P(a_i|c)=\frac{n_c+mp}{n+m}$$

Donde:

n es el número total de ocurrencias de la clasificación c.

n_c es el número de ocurrencias de la clasificación c para el atributo a_i.

p es el valor a priori estimado para $P(a_i|c)$.

m es el tamaño de muestra equivalente.

Sofía no quitaba la vista del folio donde andaba garabateando las fórmulas, y el Sr. Lego no dejaba de mirarme tratando de seguir todo aquel maremagnum de formulitas y conceptos de Probabilidad.

Está bien, veámoslo con un ejemplo concreto -concedí. Imaginad que mañana Gaznápiro llega a la oficina con una corbata amarilla, y desayuna pan y zumo. Fijaos en que ese caso concreto no está en nuestra tabla, sin embargo, vamos a tratar de inferir con los datos que tenemos cuál será su estado de ánimo. Necesitamos calcular las siguientes probabilidades:

P(Amarilla|Bueno); P(Pan|Bueno); P(Zumo|Bueno)

P(Amarilla|Malo); P(Pan|Malo); P(Zumo|Malo)

Y también las probabilidades de:

P(Bueno) y P(Malo)

Para luego poder aplicar la primera fórmula y obtener un valor que nos permita clasificar los atributos. Según la primera formula, el cálculo se realiza así:

clasificar_dia_bueno = P(Bueno)*P(Amarilla|Bueno)*P(Pan|Bueno)*P(Zumo|Bueno)

Y para el caso de que tenga un mal día será:

clasificar_dia_malo = P(Malo)*P(Amarilla|Malo)*P(Pan|Malo)*P(Zumo|Malo)

Vamos a ir calculando cada una de las probabilidades condicionadas usando la segunda fórmula. Empecemos con P(Amarilla|Bueno). Tenemos que n=4, ya que hay un total de 4 casos en la tabla en el que humor=Bueno. Para n_c tenemos que n_c=1, ya que sólo hay un caso en el que el día haya sido bueno cuando llevaba corbata amarilla. Para p tenemos que p=1/3, ya que tenemos 3 posibles valores del atributo Corbata (Rojo, Amarillo y Azul). Finalmente, m es un valor arbitrario, pero ha de ser consistente para todos los casos. Vamos a tomar un valor m=3.

Es decir:

n=4

n_c=1

p=1/3=0.333

m=3

P(Amarilla|Bueno)=(1+(3*0.333))/(4+3)=0.285

No vamos a desarrollar aquí el cálculo de todas las probabilidades, ya que se calcula siempre igual. Simplemente, calcularemos otra más correspondiente al caso humor=malo y después pondremos todos los resultados ya calculados.

Como ejemplo calcularemos la probabilidad P(Pan|Malo): Ahora tenemos:

n=5 (hay 5 casos en la tabla en el que Humor=Malo)

n_c=4 (hay 4 casos en el que comió pan teniendo un día malo)

p=1/2=0.5 (hay dos posibilidades: Pan o Fruta, por lo que la probabilidad es del 50% para cada una).

m=3

P(Pan|Malo)=(4+(3*0.5))/(5+3)=0.687

Resumiendo, estos son los valores que hemos obtenido después de calcular todas las probabilidades condicionadas:

P(Amarilla|Bueno)=0.285
P(Pan|Bueno)=0.357
P(Zumo|Bueno)=0.5

P(Amarilla|Malo)=0.249
P(Pan|Malo)=0.687
P(Zumo|Malo)=0.312

Nos resta calcular la probabilidad de P(Bueno) y P(Malo) que son: P(Bueno)=4/9=0.444 (4 ocurrencias de Bueno sobre 9)

P(Malo)=5/9=0.555 (5 ocurrencias de Malo sobre 9)

Aunque aquí nos hemos limitado a dos, evidentemente podemos hacer una clasificación en más de dos categorías.

Usando la primera fórmula obtenemos el siguiente resultado para el caso Humor=Bueno:
clasificar_dia_bueno = 0.444*0.285*0.357*0.5 = 0.0225

y para el caso Humor=Malo:

clasificar_dia_malo = 0.555*0.249*0.687*0.312 = 0.0296

Por lo que, en principio, podemos prever que, dados los condicionantes observados (corbata amarilla y desayuno de pan y zumo), lo más probable es que tenga un mal día.

Sofía y el Sr. Lego levantaron la vista del papel donde estaba haciendo los cálculos y se me quedaron mirando sin saber qué decir, así que me adelanté a la pregunta que suponía que estaban a punto de hacerme: Evidentemente, esto no es más que un modelo estadístico y no el oráculo de Delfos, esto no quiere decir que siempre que Gaznápiro pida pan y zumo para desayunar y traiga una corbata amarilla venga de mal humor. Además, el número de muestras (días) que hemos usado es bajo. En todo caso, este método, pese a su sencillez, es una poderosa herramienta para clasificar casi cualquier cosa. Los clasificadores bayesianos son muy utilizados, por ejemplo, en la clasificación de documentos. Un buen ejemplo de ello son los programas anti-spam que utilizan los servidores de correo electrónico. Si

os interesa, quizá otro día os explique como se utilizan para filtrar el spam. También se utiliza en minería de datos, traducción automática, correctores ortográficos, predicción (meteorológica, bursátil, etc…). Y en general tiene multitud de aplicaciones dentro del campo de la Inteligencia Artificial.

Al día siguiente el Sr. Lego llegó con el siguiente programa en Python que había preparado para predecir el humor de Gaznápiro:

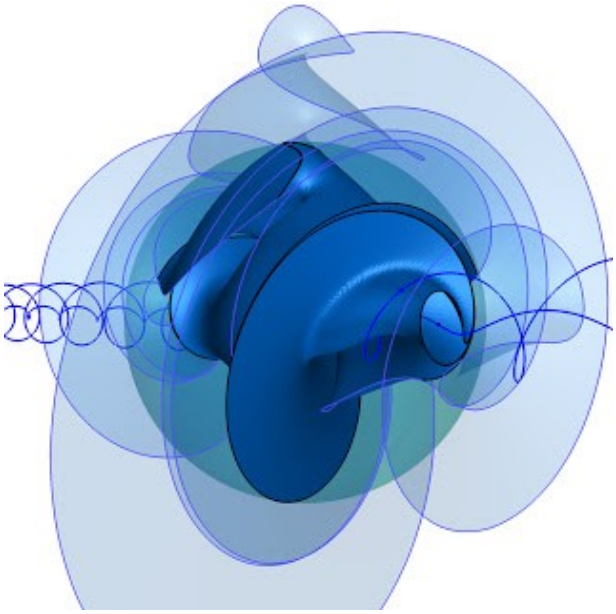
```
1.  def classify(attr, classif, data, ask):
2.    # entrenar el clasificador
3.    m=3
4.    classes={}
5.    attribs={}
6.    attribs_count={}
7.    attribs_count_per_class={}
8.
9.    # obtener atributos
10.   for i in range(len(attr)):
11.     attribs[attr[i]]=[]
12.
13.   for k,v in data.iteritems():
14.     # obtener clases
15.     if not v in classes:
16.       classes[v]=1
17.     else:
18.       classes[v]=classes[v]+1
19.
20.     # contabilizar atributos
21.     for at in k:
22.       if not at in attribs[attr[k.index(at)]]:
23.         attribs[attr[k.index(at)]].append(at)
24.       # cuenta total
25.       if not at in attribs_count:
26.         attribs_count[at]=1
27.       else:
28.         attribs_count[at]=attribs_count[at]+1
29.       # cuenta por clase
30.       if not (at,v) in attribs_count_per_class:
31.         attribs_count_per_class[(at,v)]=1
32.       else:
33.         attribs_count_per_class[(at,v)]=attribs_count_per_class[(at,v)]+1
34.
35.   # calculo valor por cada clase
36.   calc={}
37.   for i in classes:
38.     calc[i]=float(classes[i])/float(len(data))
39.     for j in ask:
```

```
40.         n=classes[i]
41.         nc=attribs_count_per_class[(j,i)]
42.         p=1.0/len(attribs[attr[ask.index(j)]])
43.         calc[i]=calc[i]*((nc+m*p)/(n+m))
44.
45.     return max(calc)
46.
47.
48. if __name__ == "__main__":
49.     attr=['corbata','desayuno','bebida']
50.     classif=['humor']
51.
52.     data={('roja','pan','cafe'):'malo',
53.       ('roja','pan','zumo'):'malo',
54.       ('roja','fruta','cafe'):'bueno',
55.       ('amarilla','pan','cafe'):'malo',
56.       ('amarilla','fruta','cafe'):'bueno',
57.       ('azul','pan','zumo'):'bueno',
58.       ('azul','fruta','cafe'):'malo',
59.       ('azul','fruta','zumo'):'bueno',
60.       ('azul','pan','cafe'):'malo'}
61.
62.     ask=['amarilla','pan','zumo']
63.
64.     theclass = classify(attr, classif, data, ask)
65.     print "Clasificado como: " + theclass
66.
```

De todas formas hoy no necesito ejecutarlo para saber cómo vendrá Gaznápiro a la oficina -dijo el Sr. Lego sonriendo- He visto su coche en el parking y alguien acaba de hacerle un buen bollo en la puerta. ¿Crees que deberíamos añadir un atributo al programa para eso?

PD. Para los que tengan curiosidad, la ejecución del programa predijo que Gaznápiro tendría un mal día… y acertó.

12. De la cola del cine a la danza estelar



Llevabamos ya casi media hora en la cola y aquello parecía no avanzar, lo que para ser un jueves por la tarde era bastante poco corriente. Mi amigo Descollante y yo solemos ir al cine este día porque suele haber poca afluencia, sin embargo, hoy la cosa estaba imposible.

Para la próxima sacamos la entrada por Internet -dije- aunque haya que pagar algo más. No entiendo como puede haber tanta gente hoy y el Jueves pasado estuvimos solos.

Pues no hay ningún misterio -dijo Descollante sin ni siquiera volver su mirada- los grupos de personas se comportan como sistemas dinámicos. No hay más.

Como era habitual, Descollante, que no distingue una pregunta retórica de

otra real, se disponía a buscarle una explicación racional a un hecho que nadie más que él se atrevería a analizar seriamente, así que tenía dos opciones: callarme y seguir disfrutando de la cola, o hacer como si me interesara su comentario y enfrentarme a una disertación que no estaba seguro de querer escuchar. Por desgracia mi maldita manía de empatizar hasta con las piedras me hizo pronunciar dos fatídicas palabras: ¿sistemas dinámicos?

Así que Descollante se animó: Sí, la dinámica de sistemas trata de explicar el comportamiento de sistemas complejos que evolucionan a través del tiempo. El comportamiento de las masas se asemeja a un sistema dinámico, ya que la forma de actuar de un grupo grande de personas a menudo no puede entenderse a partir del comportamiento de cada individuo por separado.

¿No irás a decirme que esta cola podríamos haberla predecido? Porque si es así nos forramos prediciendo el comportamiento de las masas.

Tú siempre pensando en el dinero -rió mi amigo. De hecho no creo, ya que hablamos de lo que los matemáticos llaman sistemas no lineales: sistemas cuyo comportamiento global no puede expresarse como la suma de los comportamientos individuales.

Descollante, estoy seguro de que lo que me estás contando sería muy interesante si fuera capaz de comprender algo, pero no termino de entender a dónde quieres llegar.

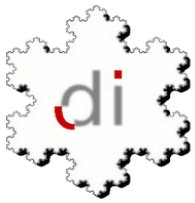
Está bien, Alberto, te pondré un ejemplo más simple. ¿Recuerdas del instituto cómo se calcula la fuerza gravitatoria ejercida entre dos cuerpos? Si -respondí- era algo así como que la fuerza es directamente proporcional al producto de sus masas e inversamente proporcional al cuadrado de su distancia.

Exacto -dijo Descollante mientras dábamos el par de pasos que había avanzado la cola- y esa formulita te permite calcular fácilmente cosas como la órbita que describirá un cuerpo alrededor del otro. Sin embargo, si añadimos un tercer cuerpo, tenemos un problema ya que lo convertimos en un sistema no lineal y describir analíticamente sus órbitas es una tarea tremendamente compleja con las matemáticas de las que disponemos. De hecho, se transforma en un sistema caótico donde empiezan a mandar las leyes de la teoría del caos.

Ahora sí que estaba desconcertado ya que aquello no tenía demasiado sentido para mí, así que no pude callarme: pero si no podemos calcular la trayectoria de sólo tres cuerpos, ¿cómo calculan los astrónomos la evolución de un cúmulo estelar o incluso de toda una galaxia?

Descollante me miró casi perdonándome la vida: he dicho que no podemos calcularlo analíticamente, pero disponemos de una herramienta maravillosa, la simulación por ordenador.

Aquello ya empezaba a gustarme más, habiendo un ordenador de por medio



siempre me siento más cómodo. Así, entre teoría del caos por aquí y problemas no lineales por allá logramos entrar en el cine y ver la peli. Ya de regreso a casa de Descollante nos sentamos ante el ordenador porque quería mostrarme cómo hacer la simulación de un sistema complejo como un cúmulo estelar.

Verás Alberto, para representar cada estrella en el ordenador almacenaremos su posición en el espacio como una coordenada tridimensional X,Y,Z. Necesitamos también almacenar su velocidad y su aceleración como vector tridimensional. La idea es que todas las estrellas ejercen una fuerza entre sí. Como hay muchas estrellas, la fuerza resultante ejercida sobre una estrella es la suma de todas las fuerzas que actúan sobre ella. Como sabes F=G(m1*m2)/d, siendo la distancia d=sqrt(x1-x2*y1-y2*z1-z2). En realidad, como bien me dijiste antes hay que utilizar el cuadrado de la distancia, pero para la simulación vamos a utilizar directamente la distancia ya que buscamos también un efecto estético en la animación del movimiento de las estrellas. Tampoco vamos a meter de por medio a la constante G, la dejaremos descansar hoy. Desde luego, si fuéramos astrofísicos tratando de hacer cálculos exactos sí que tendríamos que tener todo esto en cuenta.

Conocida la fuerza, podemos calcular la aceleración que sufre la estrella en cada eje de coordenadas:

aceleración_x=F*(posición_x2-posición_x1)/d

Es decir, la aceleración en el eje X es igual al producto de la fuerza por la diferencia entre la posición en el eje X de la otra estrella y la posición X de la estrella sobre la que estamos realizando el cálculo. Todo ello dividido por la distancia. La aceleración total que sufre la estrella es la suma de todas las aceleraciones (una por cada estrella que la rodea). El cálculo en el eje Y y Z es idéntico.

Ahora que tenemos calculado el resultado de todas las interacciones entre estrellas, sólo nos resta calcular su velocidad sumándole la aceleración calculada en el paso anterior.

velocidad_x = velocidad_x + aceleración_x

Y lo mismo para las otras dos componentes Y y Z.

Ya sólo tenemos que sumar la velocidad a la posición de la estrella y habremos calculado su movimiento.

posición_x = posición_x + velocidad_x

Y de nuevo repetimos el cálculo para los ejes Y y Z.

Repitiendo estos cálculos en un bucle y representando cada estrella gráficamente tendremos una simulación animada de un cúmulo estelar.

Mientras me lo explicaba, Descollante iba tecleando un pequeño programa en Java.

He creado un campo aleatorio de 25 estrellas -dijo Descollante- pero si quieres más sólo tienes que cambiar el valor de la variable NUM_STARS. Tampoco he simulado la colisión entre estrellas ni nada parecido, ya que simplemente queremos ver visualmente como evoluciona un sistema dinámico.

Hoy en día, los gobiernos y grandes corporaciones utilizan herramientas de análisis de sistemas dinámicos para predecir comportamientos económicos o incluso sociales.

Pues no sabes cómo me tranquiliza eso -dije.

¿Te tranquiliza? dijo sorprendido Descollante.

Reí pero no dije nada más. Aunque se lo explicara, dudo que Descollante fuera capaz de entender lo que es una frase irónica.

Aquí se reproduce el código que tecleó Descollante:

```
1.  import java.util.Random;
2.  import java.awt.*;
3.  import javax.swing.*;
4.
5.  class SpaceDance extends JFrame {
6.
7.    int RES_X = 640;
8.    int RES_Y = 480;
9.
10.   public SpaceDance() {
11.     super("SpaceDance");
12.     setBounds(0, 0, RES_X, RES_Y);
13.     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14.     Container con = this.getContentPane();
15.     con.setBackground(Color.black);
16.     GCanvas canvas = new GCanvas();
17.     con.add(canvas);
18.     setVisible(true);
19.   }
20.
21.   public static void main(String[] args) {
22.     new SpaceDance();
```

```
23.   }
24. }
25.
26. class Star {
27.
28.   int posx, posy, posz; // posición
29.   double spdx, spdy, spdz; // velocidad
30.   double accx, accy, accz; // aceleración
31.   double mass; // masa
32. }
33.
34. class GCanvas extends Canvas implements Runnable {
35.
36.   int RES_X = 640;
37.   int RES_Y = 480;
38.   int NUM_STARS = 25;
39.   int tiempo_pausa = 100;
40.   Star[] stars = new Star[NUM_STARS];
41.   Thread t;
42.
43.   public GCanvas() {
44.     t = new Thread(this);
45.     t.start();
46.   }
47.
48.   @Override
49.   public void paint(Graphics g) {
50.     // dibujar las estrellas
51.     g.setColor(Color.black);
52.     g.fillRect(0, 0, RES_X, RES_Y);
53.     for (int i = 0; i < NUM_STARS; i++) {
54.       if (stars[i] != null) {
55.         int tam = (int) stars[i].mass;
56.         if (tam < 2) {
57.           g.setColor(Color.white);
58.         } else if (tam < 4) {
59.           g.setColor(Color.yellow);
60.         } else if (tam < 6) {
61.           g.setColor(Color.blue);
62.         } else {
63.           g.setColor(Color.red);
64.         }
65.         g.fillOval(stars[i].posx, stars[i].posy, tam, tam);
66.       }
67.     }
68.   }
69.
70.   public void initStars() {
71.     // inicializar estrellas aleatoriamente
72.     Random rnd = new Random();
73.     for (int i = 0; i < NUM_STARS; i++) {
74.       stars[i] = new Star();
75.       stars[i].posx = rnd.nextInt(RES_X);
76.       stars[i].posy = rnd.nextInt(RES_Y);
```

```
77.       stars[i].posz = rnd.nextInt(RES_Y);
78.       stars[i].accx = 0;
79.       stars[i].accy = 0;
80.       stars[i].accz = 0;
81.       stars[i].mass = (rnd.nextFloat() + 0.1) * 10;
82.     }
83.   }
84.
85.   @Override
86.   public void run() {
87.     initStars();
88.     do {
89.
90.       for (int i = 0; i < NUM_STARS; i++) {
91.         // cálculo de las aceleraciones
92.         stars[i].accx = 0;
93.         stars[i].accy = 0;
94.         stars[i].accz = 0;
95.         for (int j = 0; j < NUM_STARS; j++) {
96.           if (i != j) {
97.             int dx = stars[i].posx - stars[j].posx;
98.             int dy = stars[i].posy - stars[j].posy;
99.             int dz = stars[i].posz - stars[j].posz;
100.            // distancia
101.            double d = Math.sqrt(dx * dx + dy * dy + dz * dz);
102.            if (d != 0) {
103.              // fuerza
104.              double f = (stars[i].mass*stars[j].mass) / d;
105.              // aceleraciÛn
106.              stars[i].accx += f * (stars[j].posx - stars[i].posx) / d;
107.              stars[i].accy += f * (stars[j].posy - stars[i].posy) / d;
108.              stars[i].accz += f * (stars[j].posz - stars[i].posz) / d;
109.            }
110.          }
111.        }
112.      }
113.
114.      // cálculo de las velocidades
115.      for (int i = 0; i < NUM_STARS; i++) {
116.        stars[i].spdx += stars[i].accx;
117.        stars[i].spdy += stars[i].accy;
118.        stars[i].spdz += stars[i].accz;
119.      }
120.
121.      // cálculo de las coordenadas
122.      for (int i = 0; i < NUM_STARS; i++) {
123.        stars[i].posx += (int) stars[i].spdx;
124.        stars[i].posy += (int) stars[i].spdy;
125.        stars[i].posz += (int) stars[i].spdz;
126.      }
127.
128.      repaint();
129.      try {
130.        Thread.sleep(tiempo_pausa);
```

```
131.       } catch (InterruptedException e) {
132.         e.printStackTrace();
133.       }
134.     } while (true);
135.   }
136.       }
```

13. ¿Cuántas palabras caben en una imagen?

Sofía no suele ser una persona muy expresiva, y difícilmente uno penetra su coraza para poder siquiera atisbar qué es lo que está pensando en cada momento. Ya sé que llegar al trabajo por la mañana temprano no infunde la mayor de las alegrías, pero Sofía parecía tener su propio rito de autoresignación consistente en sentarse tratando de no llamar demasiado la atención, colocarse sus cascos, abrir sus herramientas de trabajo que organizaba siempre de la misma manera en la pantalla de la derecha (era de las pocas programadoras que tenía dos monitores) y en la de la izquierda comenzaba a leer algún artículo o alguna web de noticias hasta que, cuando lo consideraba oportuno, volvía la cabeza al otro monitor y comenzaba a escribir código. El rito debe ser poderoso porque, probablemente es la programadora más brillante de la empresa.

El caso es que aquella precisa mañana, su rito cambió. En vez de colocarse los cascos y abrir una web, abrió un ejemplar del "Código de Da Vinci" de Dan Brown y se puso a leer. A mí, la verdad, es que ese libro no me entusiasmaba especialmente y puestos a elegir prefiero al Holmes de Conan Doyle o al Hércules Poirot de Agatha Christie. El caso es que no es el tipo de libro que esperaba ver leyendo a Sofía, así que me atreví a preguntar.

¿Qué tal el libro? ¿Te gusta? -pregunté.

No mucho, pero habla de criptografía y tenía curiosidad. Me ha llamado la atención la parte en la que el protagonista interpreta supuestos mensajes secretos de Da Vinci en su obra de La Última Cena. ¿No es curioso pensar que el pintor podría estar comunicándose con alguien que vive cientos de años después de su muerte?

Visto así sí que impresiona -dije- pero supongo que es lo que buscan todos los pintores al pintar un cuadro ¿no?

Si, pero imagina poder esconder información oculta en la obra y que pase desapercibida para el público general, pero no para el receptor del mensaje.
Pues sería muy interesante, pero sería más fácil enviarle una carta con algún código cifrado.

Si, pero si alguien la intercepta tiene ya una información valiosa: El emisor quiere enviar un mensaje secreto al receptor. Eso de por sí es ya una información útil para el enemigo. Se trata, pues, de enviar el mensaje sin levantar sospechas. Hoy en día lo tenemos bastante fácil con las imágenes digitales, ya no hay que pintar un cuadro. Hay una técnica detrás de este tipo de ocultación de la información llamada Esteganografía.
Te refieres a que se pueden almacenar datos en las cabeceras de los archivos de imágenes ¿no? en fin, eso no parece muy seguro -traté de parecer inteligente haciendo mis propias deducciones.

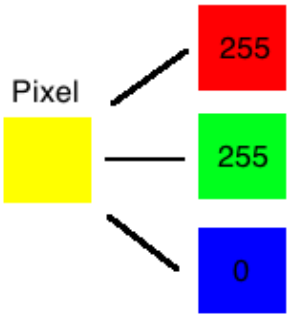
Para nada Alberto, te equivocas. Hablo de guardar información en la propia información de la imagen, en los píxeles que la componen.

Sofía me observó unos segundo como esperando a que cayera en la cuenta

de cómo podía hacerse tal cosa, como si fuera evidente, pero por enésima vez vio defraudadas sus expectativas sobre mis capacidades. Verás -prosiguió- una imagen digital de ordenador está compuesta por una cabecera, como tú bien dices, en la que se almacena información específica del formato en la que se almacena. La cabecera de una imagen JPG es diferente a la cabecera de una imagen PNG o GIF, por ejemplo. Tras la cabecera están los datos de la imagen propiamente dichos. Una imagen está compuesta por muchos píxeles, y cada pixel tiene un color. Pues bien, la clave está en cómo se almacena el color de cada pixel. El formato más común (aunque no el único) es codificar el color usando tres bytes (24 bits) en el que cada byte representa el nivel de cada componente de color (a saber: rojo, verde y azul).

Por ejemplo, un componente alto de rojo y verde, y un componente bajo de azul daría lugar al color amarillo. Podríamos codificarlo como Rojo:255, Verde:255, Azul:0. Recordemos que un byte (8 bits) pueden representar un rango de valores que va del 0 al 255 (sin signo).

En un archivo gráfico, generalmente los pixeles se almacenan como secuencias de tres bytes en formato RRVVAA (o RRGGBB de red, Green y Blue).



Vale, esto más o menos lo entiendo, pero no llego a comprender cómo puedo almacenar algo en un pixel. Para el color se usan los 24 bits y no queda ninguno sin usar donde almacenar nada sin que se vea modificado el color en sí.

Touché querido Watson, de eso precisamente se trata: de cambiar el color del pixel, pero tan ligeramente que el ojo humano sea incapaz de notar la diferencia. De hecho Alberto, ¿eres capaz de notar alguna diferencia de color entre los dos siguientes rectángulos de color rojo? (Sofía me mostró los cuadros en su pantalla).

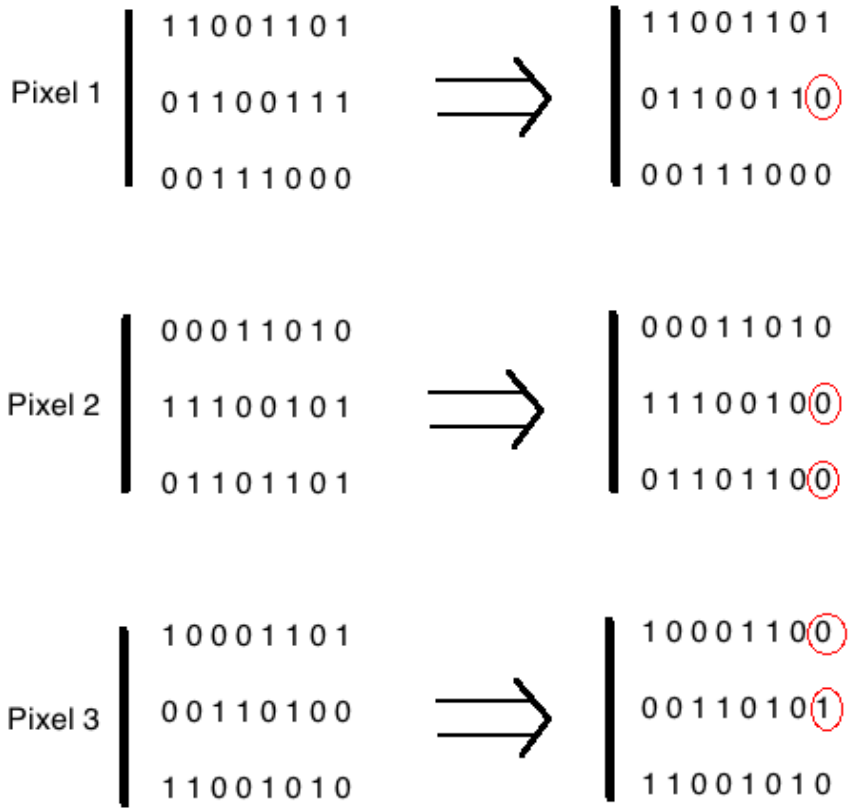
Por más que me esforcé no fui capaz de ver diferencia alguna, así que me aventuré a decir: Son iguales.

Para nada -rió sofía. El primero tienen una componente roja de 255 (en binario 11111111) y la segunda de 254 (en binario 11111110). Fíjate que sólo hay una diferencia de un bit entre uno y otro color. Entonces, si cambio un sólo bit en una componente de color, ¿no notaré la diferencia? -de nuevo traté de parecerme más a Holmes que a Watson. No, si cambio el primer bit (01111111), por ejemplo, mira lo que pasa:

Ahora, la componente roja pasa a ser 127, que dista bastante del 255 original. Por lo tanto, si queremos que el color se mantenga visualmente igual, sólo podemos tocar el bit de más a la derecha, también llamado bit menos significativo.

Ya entiendo -dije. Se trata entonces de codificar la información que queremos ocultar alterando los últimos bits de cada componente de color ¿es así?

Elemental querido Alberto. Así es. Te pongo un ejemplo. Imagina que quieres esconder un mensaje que empieza por la letra A en una imagen. La letra A tiene el código ASCII 65 (10000001 en binario). Pues bien, necesitamos modificar el último bit de la componente roja del primer bit para que acabe en 1, el de la componente verde para que acabe en 0, y así sucesivamente. Como tenemos 8 bits, cada carácter necesitará tres píxeles (3 pixeles * 3 componentes de color = 9 componentes de color) Es decir, nos sobra un bit que, por mantener las cosas simples, no vamos a modificar. El proceso sería pues el siguiente (Sofía garabateó sobre un papel):



Sofía rodeó de color rojo los bits que era necesario cambiar y me hizo notar como la componente azul del último pixel no la estábamos usando en el ejemplo, pero que podía usarse sin problema aunque complicaba un poco la programación. También me explicó que debían usarse formatos de archivos con compresión sin pérdidas, ya que formatos como el JPG pueden alterar las componentes de color originales.

Por cierto -continuó Sofía- puede almacenarse información "secreta" en otros tipos de archivos digitales, como vídeo o sonido. De hecho, en el caso del sonido el proceso es muy similar.

Lo cierto es que todo lo que me estaba contando Sofía empezaba a sonarme a buen argumento para una novela, pero como se me da mejor la programación que la literatura, decidí escribir un programa para poner todo en práctica en vez de tratar de hacerle sombra a Dan Brown. Tecleé dos programas, una llamado Encode.java que almacena un texto en una imagen y otro llamado Decode.java que realiza el proceso contrario. Reproduzco aquí los dos programas.

Encode.java

```
1.  import java.awt.Color;
2.  import java.awt.image.BufferedImage;
3.  import java.io.File;
4.  import javax.imageio.ImageIO;
5.
6.  public class Encode {
7.      public static void main(String[] args) {
8.          if (args.length<2) {
9.              System.out.println("Formato: Encode imagen \"texto\"");
10.             System.exit(0);
11.         }
12.         String img_in=args[0];
13.         String message=args[1];
14.
15.         message=message+"#";
16.         // cargar imagen
17.         BufferedImage img = null;
18.         try {
19.             img = ImageIO.read(new File(img_in));
20.         } catch (Exception e) {
21.             e.printStackTrace();
22.         }
23.
24.         if (message.length()>((img.getWidth()*img.getHeight())/3)) {
25.             System.out.println("El mensaje es demasiado grande para la
    imagen.");
26.             System.exit(0);
27.         }
28.         // almacenar mensaje
29.         // cada tres pixels un caracter
30.         int pix_index=0;
31.         byte[] chars=null;
32.         try {
33.             chars=message.getBytes("ASCII");
```

```
34.         } catch (Exception e) {
35.             e.printStackTrace();
36.         }
37.         for (int i=0; i<chars.length; i++) {
38.             for (int j=0; j<3; j++) {
39.                 int ycoord=(int)(pix_index/img.getWidth());
40.                 int xcoord=pix_index-(ycoord*img.getWidth());
41.                 int c = img.getRGB(xcoord,ycoord);
42.                 int red = (int)((c & 0x00ff0000) >> 16);
43.                 int green = (int)((c & 0x0000ff00) >> 8);
44.                 int blue = (int)(c & 0x000000ff);
45.                 if (isBitSet(chars[i], (j*3)+0)) {
46.                     red |= 1 << 0;
47.                 } else {
48.                     red &= ~(1 << 0);
49.                 }
50.                 if (isBitSet(chars[i], (j*3)+1)) {
51.                     green |= 1 << 0;
52.                 } else {
53.                     green &= ~(1 << 0);
54.                 }
55.                 if (j<2) {  // el ultimo byte no lo usamos
56.                     if (isBitSet(chars[i], (j*3)+2)) {
57.                         blue |= 1 << 0;
58.                     } else {
59.                         blue &= ~(1 << 0);
60.                     }
61.                 }
62.                 Color color=new Color(red, green, blue);
63.                 img.setRGB(xcoord, ycoord, color.getRGB());
64.                 pix_index++;
65.             }
66.         }
67.         // guardar imagen
68.         try {
69.             File outputfile = new File("encodedimage.png");
70.             ImageIO.write(img, "png", outputfile);
71.         } catch(Exception e) {
72.             e.printStackTrace();
73.         }
74.     }
75.
76.     private static Boolean isBitSet(byte b, int bit) {
77.         return (b & (1 << bit)) != 0;
78.     }
79. }
```

Decode.java

```
1. import java.awt.Color;
2. import java.awt.image.BufferedImage;
3. import java.io.File;
```

```
4.  import javax.imageio.ImageIO;
5.
6.  public class Decode {
7.      public static void main(String[] args) {
8.          if (args.length<1) {
9.              System.out.println("Formato: Decode imagen");
10.             System.exit(0);
11.         }
12.         String img_in=args[0];
13.
14.         // cargar imagen
15.         BufferedImage img = null;
16.         try {
17.             img = ImageIO.read(new File(img_in));
18.         } catch (Exception e) {
19.             e.printStackTrace();
20.         }
21.
22.         // obtener mensaje
23.         for (int i=0; i<(img.getWidth()*img.getHeight())-3; i=i+3){
24.             char car=' ';
25.             // primer pixel
26.             int ycoord=(int)(i/img.getWidth());
27.             int xcoord=i-(ycoord*img.getWidth());
28.             int c = img.getRGB(xcoord,ycoord);
29.             int red = (int)((c & 0x00ff0000) >> 16);
30.             int green = (int)((c & 0x0000ff00) >> 8);
31.             int blue = (int)(c & 0x000000ff);
32.             if (isBitSet((byte)red,0)) {
33.                 car |= 1 << 0;
34.             } else {
35.                 car &= ~(1 << 0);
36.             }
37.             if (isBitSet((byte)green,0)) {
38.                 car |= 1 << 1;
39.             } else {
40.                 car &= ~(1 << 1);
41.             }
42.             if (isBitSet((byte)blue,0)) {
43.                 car |= 1 << 2;
44.             } else {
45.                 car &= ~(1 << 2);
46.             }
47.             // segundo pixel
48.             ycoord=(int)((i+1)/img.getWidth());
49.             xcoord=i+1-(ycoord*img.getWidth());
50.             c = img.getRGB(xcoord,ycoord);
51.             red = (int)((c & 0x00ff0000) >> 16);
52.             green = (int)((c & 0x0000ff00) >> 8);
53.             blue = (int)(c & 0x000000ff);
54.             if (isBitSet((byte)red,0)) {
55.                 car |= 1 << 3;
56.             } else {
57.                 car &= ~(1 << 3);
```

```
58.                 }
59.                 if (isBitSet((byte)green,0)) {
60.                     car |= 1 << 4;
61.                 } else {
62.                     car &= ~(1 << 4);
63.                 }
64.                 if (isBitSet((byte)blue,0)) {
65.                     car |= 1 << 5;
66.                 } else {
67.                     car &= ~(1 << 5);
68.                 }
69.
70.                 // tercer pixel
71.                 ycoord=(int)((i+2)/img.getWidth());
72.                 xcoord=i+2-(ycoord*img.getWidth());
73.                 c = img.getRGB(xcoord,ycoord);
74.                 red = (int)((c & 0x00ff0000) >> 16);
75.                 green = (int)((c & 0x0000ff00) >> 8);
76.                 blue = (int)(c & 0x000000ff);
77.                 if (isBitSet((byte)red,0)) {
78.                     car |= 1 << 6;
79.                 } else {
80.                     car &= ~(1 << 6);
81.                 }
82.                 if (isBitSet((byte)green,0)) {
83.                     car |= 1 << 7;
84.                 } else {
85.                     car &= ~(1 << 7);
86.                 }
87.
88.                 if (car=='#') {
89.                     break;
90.                 }
91.                 System.out.print(car);
92.             }
93.         }
94.
95.         private static Boolean isBitSet(byte b, int bit) {
96.             return (b & (1 << bit)) != 0;
97.         }
98. }
```

Bueno, el código no es especialmente elegante, pero parece que funcionará -dijo Sofía machacando de nuevo mi autoestima.

Para probar el programa usamos una fotografía de un viaje a Dinamarca que había realizado Sofía. Tras compilar los programas ejecutamos el comando:

java Encode dinamarca.jpg "mensaje oculto"

Obtuvimos la siguiente imagen:

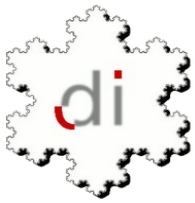
Como ves -dijo Sofía- son indistinguibles la una de la otra. Tras ejecutar: java Decode encodedimage.png obtuvimos de nuevo el texto "mensaje oculto".

Bien, se me ocurren algunas utilidades para este programa -dije. Si estas pensando en pasar mensajes ocultos dentro de imágenes a otra empresa para ganarte un sobresueldo, tendrás que refinar un poco el método, ya que, aunque codificada en la imagen, el texto no está encriptado y si alguien sospecha que hay un mensaje oculto será fácil obtenerlo. Aunque si lo que quieres es que Gaznápiro no los intercepte, creo que ni en mil años sería capaz.

14. Naturaleza matemática

He tenido que pasar por el aro soportar a mi jefe, que ganó su puesto en una tómbola (no quedaban perritos piloto). He tenido que aguantar compañeros obcecados en pensar que mientras más bajo cae el de la mesa de al lado, más altos están ellos (esto es una jungla chicos -suele decir Gaznápiro). He sufrido en silencio las... bueno, ya sabes... aunque sea metafóricamente en este caso. Y uno, que tiene la mala costumbre de comer a diario, no puede hacer más que soportar el sacrificio a cambio de algo de sustento. Pero hay cosas y cosas... Aquél Sábado, algún gerifante tuvo la fantástica idea de que debíamos renunciar a nuestro día de descanso para pasar un día de campo reunidos con -en palabras suyas- nuestra segunda familia. Para reforzar lazos, mejorar el trabajo en equipo y, bueno, todo eso que a los iluminados de RRHH les gusta decir (eso sí, que no se te ocurra ponerte malo y no justificarlo, que se te cae el pelo por mucha segunda familia que sean).

En fin, que allí estaba yo, tratando de evitar todos esos jueguecitos motivacionales, ocultándome tras una arboleda con la esperanza de que nadie me echara en falta... Pero claro, se ve que no era el único que



trataba de zafarse, porque allí estaba Sofía, sentada contra un árbol mirando al cielo como esperando a que bajara un platillo volante a abducirla. En voz baja, en parte para que no nos escucharan, y en parte para no sobresaltarla al sacarla del trance, pregunté:

- ¿Va a llover? -dijé tratando de ser gracioso.

- ¿No es fantástico? Fíjate en como se distribuyen las ramificaciones de esos árboles ¿No te recuerda a una construcción fractal?

En ese momento casi me entraron ganas de volver con el grupo y ponerme a hacer carreras de sacos o lo que fuera que estaban haciendo, aunque lo pensé mejor y decidí sentarme en el árbol de al lado con la esperanza de que Sofía volviera a caer en el trance hipnótico del que la había sacado. Pareciera que finalmente hubiera entrado en estado meditativo, sin embargo, se puso trascendente y continuó hablando.

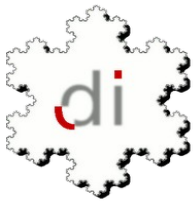
- ¿Te has fijado que casi todo en la naturaleza sigue un patrón o unas proporciones matemáticas?

- Bueno, supongo que lo dices por eso de las órbitas planetarias, Kepler y todas esas cosas ¿no?

Sofía me miró de nuevo con cierto aire de resignación antes de preguntar ¿Te gustan la flores?

- No mucho, son un poco insípidas -traté de nuevo de hacer un chiste ingenioso, sin éxito, claro.

- Algunas flores, como los lirios o el iris tienen 3 pétalos, las rosas suelen tener 5, las Espuela de Caballero tienen 8, la caléndula del maíz o algunas



margaritas tienen 13, 21 pétalos tiene la achicoria, 34 la flor del plátano y la familia de las Asteraceae puede tener 55 y 89. ¿Qué te parece? Me quedé un poco descolocado, porque no sabía exactamente qué me estaba preguntando.

- Fíjate Alberto: 3, 5, 8, 13, 21, 34, 55, 89 ¿no te dice nada?

- Pues no -tuve que reconocer.

- Se trata de la sucesión de Fibonnaci.

- ¡Ah si! Ahora caigo. Es la sucesión en la que cada número es la suma de los dos anteriores: 1, 1, 2, 3, 5, 8, 13, 21, 34, etc.

- ¿Y no te parece demasiada casualidad?

- Bueno, curioso sí que es -contesté pensativo.

- ¿Y no te parece curioso también que la relación entre la longitud real de muchos ríos y su distancia en línea recta desde el nacimiento hasta la desembocadura sea un valor algo mayor que tres?

De nuevo me quedé tratando de averiguar a donde quería llegar.
- Te lo diré de otra forma -continúo Sofía- El promedio está alrededor del número 3.14 ¿Te suena?

- ¡Claro! Es el número Pi. ¡Es la misma proporción que existe entre la circunferencia de un círculo y su diámetro!

- Estos árboles que nos rodean también parecen seguir un patrón matemático. Si te fijas, una ramita es como una versión en pequeño de la rama sobre la que ha crecido, y a su vez, esta rama es como una versión más pequeña del árbol completo.

- Es como si fueran árboles recursivos -dije tratando de parecer que seguía su argumentación.

- Sí, efectivamente...

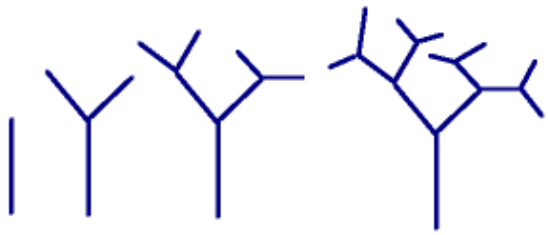
No tuvo tiempo de terminar la frase cuando escuchamos la inconfundible voz nasal y chillona de Gaznápiro detrás nuestra: ¿Qué tortolitos? ¿no os integráis en el equipo?

La idea de que Sofía pudiera tener pensamiento alguno que incluyera el concepto de relación con el sexo opuesto me causó hasta gracia, así que la mire esperando encontrar en su cara una expresión de rechazo a las palabras de Gaznápiro. Sin embargo me encontré con una sonrisa poco habitual... Lo que me dejo totalmente desconcertado.

El resto del día lo pasé tratando de no desentonar demasiado entre toda aquella jauría de "animales sociales" -que es lo que decía el jefe de RRHH que eramos, el muy… animal social.

Ya de tarde, en casa, no dejaba de pensar en lo que me había estado contando Sofía sobre los árboles, así que me senté a teclear un programa en Java tratando de emular a la naturaleza. Quizás con la idea secreta de cultivar mi propio bosque digital de bonsáis de diseño.

La idea era hacer que a partir de una rama inicial se crearan otras dos subramas, y de cada una de ellas, otras dos, es decir, crear las ramas de forma recursiva tal y como se observa en la siguiente figura.



Cada rama tendría que tener menos longitud que su rama padre (entre 5 y 10 píxeles más pequeña) y menor grosor (un píxel menos que su rama padre).

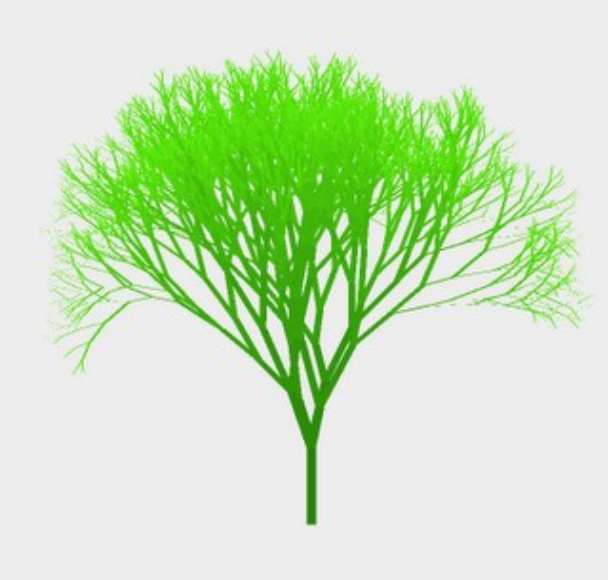
Para que pareciera un árbol de verdad introduje algo de caos en el asunto. Añadí un factor aleatorio al ángulo de crecimiento de cada rama y también a la longitud de cada rama hija.

La recursividad la limité a una profundidad de 12, y a cada recursión le asigné un color cada vez más claro para conseguir algo más de realismo. El resultado fue el siguiente código.

```
1.  import java.awt.Color;
2.  import java.awt.Graphics;
3.  import java.util.Random;
4.  import javax.swing.JFrame;
5.
6.  public class FractalTree extends JFrame {
7.
8.      public FractalTree() {
9.          super("Naturaleza fractal");
10.         setSize(800, 800);
11.         setDefaultCloseOperation(EXIT_ON_CLOSE);
12.     }
13.
14.     private void drawTree(Graphics g, int x1, int y1, double angle, int
    depth) {
15.       if (depth > 0) {
```

```
16.         Random rand = new Random();
17.         int branch_step=5+rand.nextInt(5);
18.         int x2=x1+(int)
    (Math.cos(Math.toRadians(angle))*(depth*branch_step));
19.         int y2=y1+(int)
    (Math.sin(Math.toRadians(angle))*(depth*branch_step));
20.         g.setColor(new Color(100-(depth*5),250-(depth*10),10));
21.         int[] xpoints=new int[4];
22.         int[] ypoints=new int[4];
23.         xpoints[0]=x1;
24.         ypoints[0]=y1;
25.         xpoints[1]=x1+depth;
26.         ypoints[1]=y1;
27.         xpoints[2]=x2+depth;
28.         ypoints[2]=y2;
29.         xpoints[3]=x2;
30.         ypoints[3]=y2;
31.         g.fillPolygon(xpoints, ypoints, 4);
32.         int rangle=rand.nextInt(30);
33.         int langle=rand.nextInt(30);
34.         drawTree(g, x2, y2, angle-langle, depth-1);
35.         drawTree(g, x2, y2, angle+rangle, depth-1);
36.       }
37.     }
38.
39.     @Override
40.     public void paint(Graphics g) {
41.         drawTree(g, 400, 600, -90, 12);
42.     }
43.
44.     public static void main(String[] args) {
45.         new FractalTree().setVisible(true);
46.     }
47. }
```

Aquí se pueden ver dos arbolitos creados con este programa, que por supuesto envíe a Sofía por email con la esperanza de que se sintiera, de alguna forma, halagada.



15. Secretos compartidos

No soy una persona que tenga mucho que esconder, pero hay cosas que más vale mantener en secreto. Por mi parte, lo típico: el pin del móvil y el de la tarjeta de crédito, las claves del banco, de las cuentas de correo, del ordenador, y poco más. Si alguien llegara a acceder a mi cuenta bancaria, dependiendo del día del mes, igual hasta le tocaba pagar, y en mi cuenta de correo tampoco hay ningún secreto de estado.

Sin embargo, hay secretos de los que dependen cosas mucho más importantes. A nadie se le escapa que la clave del correo de un presidente de gobierno o de un empresario importante es harina de otro costal. Con un poco de información privilegiada, ganar en la bolsa puede llegar a ser un juego de niños (está claro que no hay otra forma de acceder a esa información privilegiada que accediendo ilegalmente a sus correos, porque nadie duda de la integridad de los que manejan esa información ¿verdad?). El caso es que aquella mañana de Lunes, durante el desayuno, alguien sacó el tema -creo que el sr. Lego- y, por supuesto, a pesar de mis intentos por desviar la conversación a ámbitos más mundanos, Sofía no podía resistir la tentación de enfrascarse en cualquier conversación que oliera a seguridad informática.

Ya, de perdidos al río -pensé- así que traté de volver a entrar en la conversación:

- ¿Os imagináis tener que mantener en secreto algo como los códigos de lanzamiento de unos misiles nucleares?

¡Vaya! Ya está Alberto con sus paranoias -pude leer en la expresión de el Sr. Lego.

Sofía, sin embargo, fue mucho más directa y menos sutil:

- ¡No digas tonterías! ¿tu te fiarías de alguien tanto como para darle las claves de lanzamiento de un misil nuclear? ¿Y si se vuelve loco y le da por desatar la tercera guerra mundial?

- Quizás se podría dividir la clave en varios fragmentos y repartirlos entre varias personas, así nadie sabe la clave completa y nadie puede lanzar los misiles por su cuenta -dije tratando de impresionar a Sofía.

- Entonces, si fuera un país enemigo sólo tendría que matar a uno de los que tienen la clave para evitar que pudiera lanzar los misiles... o bastaría esperar a que a alguno le de un infarto... ¡jaque mate!

- A ver -continuó el Sr. Lego- necesitamos una forma de repartir fragmentos de la clave entre varias personas de forma que ninguna conozca la clave completa, y además, queremos ser capaces de reconstruir la clave si falla una o más de una de las personas que tienen los fragmentos. Me parece que no es posible.

- Pues siento decir que te equivocas -sonrió Sofía.

En criptografía -continuó Sofía- para resolver este tipo de asuntos se utilizan los llamados esquemas de compartición de secretos. Para este caso

concreto se utilizan los esquemas de umbral. Por ejemplo, si queremos repartir un secreto entre m personas y que el secreto pueda recuperarse con n personas, hablamos de un esquema de umbral (m,n), donde n es el umbral.

Parar a Sofía a estas alturas, estaba ya fuera del alcance de cualquier mortal, así que mi única salida fue preguntar, no sin cierta resignación: Vale, pero ¿cómo funcionan esos esquemas?

Sofía apuró el café que le quedaba en el vaso, se detuvo unos instantes como para componer en su cabeza lo que iba a decir, y continuó: Hay varios métodos, el más sencillo, quizás, es usar el esquema vectorial, introducido por el matemático George Blakley de la universidad A&M de Texas: Si queremos compartir un secreto con un esquema de umbral (m,n) hacemos corresponder el secreto con un punto P del espacio m-dimensional, y los fragmentos serán hiperplanos de dimensión m-1 que tienen a P como punto de intersección. De esta forma, con m hiperplanos, podemos reconstruir el secreto.

- Por la cara del Sr. Lego, hacía rato que ya no seguía a Sofía. Yo, por mi parte tampoco, pero creía que estaba controlando mejor mi expresión facial hasta que Sofía me miró y sonrió con esa sonrisa piedosa del que se sabe muy por encima de su interlocutor.

Está bien -dijo Sofía tratando de simplificar un poco el asunto- imaginemos que quiero compartir en secreto las tres siguientes cifras: 3,10,5. Y lo quiero hacer entre 4 personas, de forma que sólo con 3 pueda recuperar la clave. Como son tres cifras, podemos pensar en ellas como en un punto P(x,y,z) del espacio de 3 dimensiones. Sólo tendríamos que crear 4

ecuaciones como las siguientes.

$$x+2y-z=18$$
$$3x+5y+3z=74$$
$$-5x-3y+2z=-35$$
$$x-5y+2z=-37$$

Si damos una ecuación a cada una de las cuatro personas, ninguna de ellas podrá conocer el valor del punto P(x,y,z). Sin embargo, si cualquiera de ellas tres se reúnen y resuelven el sistema que forman sus tres ecuaciones, habrán conseguido descifrar el secreto P(3,10,5), aunque no esté una de las personas.

Bastante ingenioso -pensé- no se me hubiera ocurrido, y eso que una vez que conoces el sistema, es bastante obvio.

Otro esquema, algo más elaborado -continuó Sofía- es el esquema de Shamir, que inventó el matemático Adi Shamir. Este esquema es conceptualmente bastante simple, aunque su uso implica algo más que resolver un simple sistema de ecuaciones.

Sofía paró un momento como tratando de ordenar toda la información que tenía en la cabeza y seguidamente preguntó:

- Si dibujo dos puntos en un papel, ¿cuantas líneas rectas existen que pasen exactamente por esos dos puntos?

El Sr. Lego y yo nos miramos unos instantes antes de responder al unisono: una.

- Bien -continuó Sofía- del mismo modo que sólo necesito dos puntos para definir una recta, sólo necesito tres para definir una parábola ¿verdad?

Esto ya no era tan obvio, pero haciendo un acto de fe el Sr. Lego y yo asentimos con la cabeza.

En general, necesitamos n+1 puntos en el plano para definir una curva (polinomio) de grado n ¿os lo demuestro? -preguntó Sofía.

Me apresuré a decir que no era necesario antes de que se lanzara a hacer la demostración matemática.

En general -siguió Sofía- si quiero compartir un secreto repartiéndolos en m fragmentos y con un umbral de n fragmentos, tengo que construir un polinomio de grado n-1.

Imaginemos entonces que quiero compartir en secreto el pin de mi tarjeta de crédito que, digamos, es 3254. Y lo quiero hacer entre 4 personas con un umbral de 3. Necesito, por lo tanto, un polinomio de grado 2 (es decir n-1) que tendrá la forma siguiente:

$$f(x)=a_0+a_1x+a_2x^2$$

A los términos a0, a1 y a2 los llamamos coeficientes del polinomio. A a0, que es el término independiente, le asignamos el valor del secreto: en este caso a0=3254. Al resto de coeficientes le asignamos valores aleatorios. Por ejemplo: a1=12 y a2=25. Una vez definido nuestro polinomio, como quiero repartir el secreto entre 4 personas, necesito el valor de cuatro puntos cualesquiera de la función. Por ejemplo:

$$f(x)=3254+12x+25x^2$$
$$f(1) \rightarrow 3291$$
$$f(2) \rightarrow 3378$$
$$f(3) \rightarrow 3515$$
$$f(4) \rightarrow 3702$$

En este caso hemos escogido los puntos 1, 2, 3 y 4, pero podrían ser cualesquiera.
A cada una de las personas les damos un par con uno de los puntos y el valor que toma la función en dicho punto. Es decir tendríamos los siguientes cuatro fragmentos:

(1, 3291)
(2, 3378)
(3, 3515)
(4, 3702)

Pero según hemos dicho antes, un polinomio de grado n está definido de forma inequívoca por por n+1 puntos, así que como nuestro polinomio era de grado 2, sólo necesitamos 3 de los puntos (fragmentos) para poder reconstruirlo.

La idea está clara -respondí- pero ¿cómo se reconstruye un polinomio a partir de unos puntos?
No te preocupes, Alberto, eso lo resolvió Lagrange hace ya algunos añitos. Podemos usar el polinomio interpolador de Lagrange para obtener el polinomio original, pero ¿realmente necesitamos todo el polinomio original? -nos desafió Sofía.
No, sólo el término independiente del polinomio, que es el que contiene el secreto -respondió el Sr. Lego sorprendiendo a propios y a extraños. Efectivamente Sr. lego. No voy a entrar en detalles matemáticos, que son un poco aburridos, pero tras jugar un poco con el polinomio de Lagrange, podemos llegar a la siguiente formulita que nos devuelve el término independiente del polinomio.

$$f(0)=\sum_{i=1}^{t} y_i \prod_{\substack{1\leqslant i\leqslant t \\ i\neq j}} \frac{x_j}{x_j-x_i}$$

Siendo las x los puntos y las y los valores que toma la función en dichos puntos.
Como véis, la formulita de marras se puede modelar muy bien con un par de bucles así que es fácil hacer un programa para generar y develar secretos con el esquema de Shamir.
A la vuelta del desayuno, ya en la oficina, Sofía nos envió las siguientes dos funciones para generar secretos y para desvelarlos. Eso sí -nos advirtió- ningún secreto está a salvo para siempre.

```
1.  from math import ceil
2.  from random import random
3.
4.
5.  def gen_secret(secret, users, threshold):
6.      out = []
7.
8.      max_value=100    # valor maximo factores
9.      n=threshold      # grado del polinomio
10.     # generar polinomio grado n
11.     factores=[]
12.     fragmentos=[]
13.     factores.append(secret)
14.     for i in range (1,n):
15.         t=ceil(random()*max_value)
16.         factores.append(t)
17.
18.     # generar n fragmentos
19.     for i in range(users):
20.         # escogemos punto t de la funcion aleatorio
21.         t=int(ceil(random()*max_value))
22.         # calcular valor de f(t)
23.         v=factores[0]
24.         for j in range(1,n):
25.           v=v+factores[j]*(t**j)
26.
27.         fragmentos.append([t, int(v)])
28.
29.     out=fragmentos
30.     return out
31.
32.
33.
34.
35. def recover_secret(fragments, threshold):
36.     out = 0
37.
38.     if len(fragments)<threshold:
39.         print "faltan fragmentos"
40.     else:
```

```
41.
42.         for i in range(threshold):
43.             tmp_j=1
44.             for j in range(threshold):
45.                 if j!=i:
46.                     tmp_j=tmp_j*(fragments[j][0]/float((fragments[j][0]-
    fragments[i][0])))
47.             out=out+(fragments[i][1]*tmp_j)
48.
49.     return int(out)
50.
51.
52. # calculamos 4 fragmentos para secreto 1234 con umbral 3
53. s=gen_secret(1234,4,3)
54. # recuperamos el secreto con 3 fragmentos
55. print recover_secret(s, 3)
```

16. Comerciales viajeros y colinas peligrosas

El Sr. Lego no paraba de agitarse inquieto en su asiento, como esperando algo. Así llevaba ya un buen rato cuando me decidí a preguntarle si tenía algún problema o podía ayudarlo en alguna cosa. Nada, nada -me respondío inquieto- cosas mías.

Con esas traté de meterme de nuevo en el código en el que estaba trabajando, pero los movimientos casi espasmódicos del Sr. Lego empezaban a ponerme nervioso. ¿Seguro que no le pasa nada Sr. Lego? Al hacerle la pregunta volvió su cara y pude ver una mirada suplicante: Es que... esto no termina...

Me levanté y vi como estaba ejecutando un programa. Por lo menos llevaba una hora en ejecución.

Es que estoy tratando de resolver un problema para ganar un concurso que he encontrado en esta página: http://www.tsp.gatech.edu.

El concurso consistía en encontrar la solución al conocido problema TSP (Traveling Salesman Problem). El problema consiste en una serie de ciudades que un viajante de comercio tiene que recorrer. El recurrido tiene que cumplir tres requisitos, que el número de kilómetros recorridos sea el

mínimo posible, que el viajante no pase dos veces por la misma ciudad y que el viajante regrese a la ciudad de inicio.

¿Cómo está tratando de resolverlo Sr. Lego? -pregunté.

Estoy tratando de hacer una búsqueda en el espacio de estado del problema con las técnicas de búsqueda de las que hablamos hace algún tiempo. Verá Sr. Lego, no creo que aquellas técnicas de búsqueda le sean muy útiles en este problema concreto, que pertenece a una clase de problemas que en Ciencias de la Computación se denomina "intratable", o más técnicamente, NP-Dificil.

NP-¿qué? -se extrañó el Sr. Lego.

Verá, hay problemas que son tan complejos que son computacionalmente irresolubles. Por mucho tiempo que dejara su ordenador funcionando jamás encontraría la respuesta, o en el mejor de los casos, tardaría miles o millones de años. Recuerde que ya lo comentamos el día que hablamos de la búsqueda en árboles.

Para medir la complejidad de un problema usamos la notación O (O grande), también conocida como cota superior asintótica.
La cara del Sr. Lego no dejaba lugar a dudas de que estaba a punto de abandonar su idea de resolver el problema que tenía entre manos.
No se asuste -continué- una función O(x) es cota superior de otra f(x) cuando para cualquier valor que pueda tomar f(x), la función O(x) siempre será mayor. Por ejemplo:

$O(x)=3X^2$ es cota superior de $f(x)=4x$, ya que sea cual sea x, O(x) tendrá un valor superior que f(x).

Se trata pues de buscar una función O(x) que nos sirva para medir, en el caso de que x tienda a infinito, cómo se comportaría nuestro programa en cuanto a operaciones que tendría que ejecutar. En el caso del TSP, usted está probando todas las combinaciones posibles de recorridos de ciudades, es decir, si tengo 5 ciudades necesito probar 5! (leído 5 factorial) combinaciones. Es decir, 120 combinaciones, nada que cualquier ordenador de andar por casa no pueda hacer en un plis plas.
¿Entonces? -preguntó confundido el Sr.Lego- ¿Por qué tarda tanto en terminar mi programa?

Verá, con tan sólo 20 ciudades tenemos 20! = 2432902008176640000 permutaciones. Si su ordenador puede realizar una permutación cada milisegundo, tardará aproximadamente 77.146.817 años en obtener la respuesta. ¡Más de 77 millones de años!. ¿Supongo que querrá irse hoy a comer? -bromeé.

En general, para n ciudades, usted necesitará n! combinaciones, por lo tanto, la complejidad de su programa usando la notación O será O(n!), que será cota superior de la complejidad del problema TSP usando una búsqueda como la que usted ha programado. Como le decía, es un problema intratable incluso para un número de ciudades moderado.
O sea, que no hay forma de resolverlo... -concluyó el Sr. lego.
Yo no he dicho eso. En realidad, si podemos permitirnos obtener una solución que sea buena aunque no sea la óptima, hay métodos que nos permiten acercarnos bastante a la solución, y en el mejor de los casos, incluso llegar a la solución óptima. Aunque casi nunca podremos tener la certeza de que la respuesta obtenida es la mejor posible.
Si quiere, Sr. Lego, le puedo mostrar una técnica llamada Hill Climbing, o

escalada de la colina en español, para que pueda resolver su problema TSP. Está bien, pero no pienso compartir el premio -dijo medio en broma, medio en serio.

Podríamos hacer una analogía con los problemas de optimización que nos hacían resolver en el instituto ¿los recuerda?

Sí, esos en los que había que encontrar el área máxima que podíamos hallar con x metros de alambrada ¿no?

Efectivamente, se trataba de encontrar el máximo o el mínimo de una función dada. Aquí podríamos decir que buscamos minimizar la suma de distancias que hay hay que recorrer para pasar por todas las ciudades. El problema es que no tenemos una función matemática que describa el problema. ¿Cómo encontrar entonces el máximo en este caso?
El Sr. Lego se quedó esperando a que continuara sin mediar palabra.
Imagine que en una noche oscura está usted perdido en el campo. No ve más allá de unos metros por delante, pero sabe que está en una colina y que en lo más alto hay un refugio ¿cómo trataría de llegar a ella si no puede orientarse?
Trataría de tomar el camino que tiene una pendiente ascendente más acusada. Si subo siempre acabaré llegando a lo más alto de la colina -me explicó el Sr. Lego.

¡Muy bien! Esta misma estrategia podemos aplicarla a nuestro problema, pero primero planteemos el problema en términos que nos permitan tratarlo convenientemente.

Diremos que una solución válida es un recorrido cualquiera que cumpla las condición de pasar por todas las ciudades sin repetir ninguna.

La función de evaluación que nos dirá cuánto de buena es esa solución será la suma de todas las distancias que hay que recorrer entre cada ciudad. Además, para generar todas las combinaciones (estados) posibles, usaremos un operador consistente en permutar dos ciudades cualesquiera. Diremos que un estado es vecino de otro si entre ellos sólo se han permutado dos ciudades cualesquiera.

Era aparente que el Sr. Lego se esforzaba por anotar mentalmente lo que le estaba contando.

Se trata ahora de, partiendo de un estado, escoger el camino de máxima pendiente (o mínima pendiente si queremos minimizar la función). Esto es, trata de escoger aquel vecino que nos acerca más a nuestro destino. Por ejemplo, si tengo la siguiente ruta en la que cada número representa una ciudad: 1-5-3-4-2 cuya función de evaluación es 45Km (suma de las distancias), y evaluamos los siguientes vecinos (sólo evaluaremos 3 de todos los posibles para simplificar):

5-1-3-4-2 (47Km)

1-3-5-4-2 (26Km)

1-5-2-4-3 (39Km)

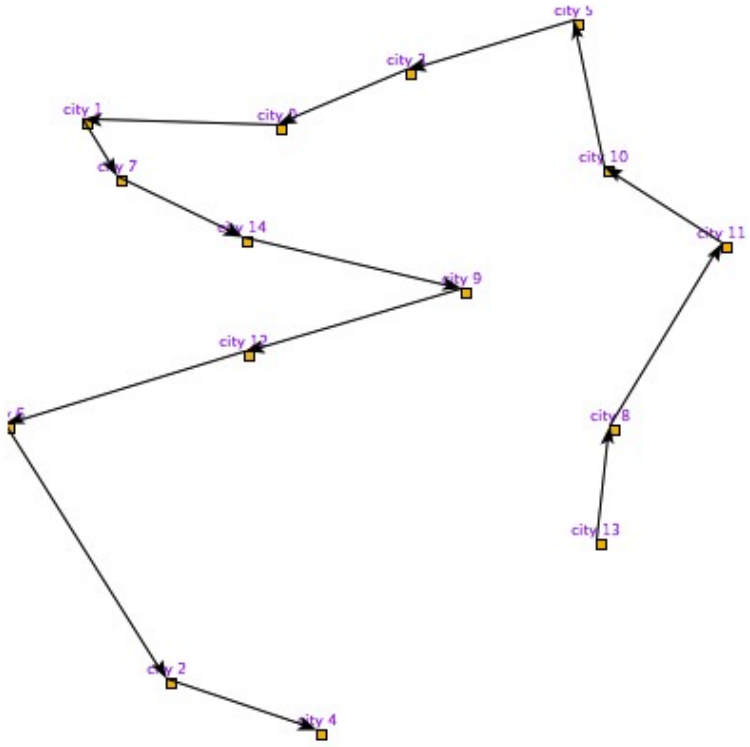
es obvio que el segundo vecino que hemos evaluado es el que más nos acerca a nuestro destino, por lo tanto, lo escogemos y volvemos a evaluar los vecinos de este nuevo estado. Repetiremos esta operación hasta que encontremos un estado cuyos vecinos no mejoran la función de evaluación. La cara del Sr. lego se iluminó de golpe. Sin decir nada corrió a su mesa y se

puso a teclear como un poseso. No pasó mucho tiempo hasta que vino a enseñarme como había implementado el algoritmo.

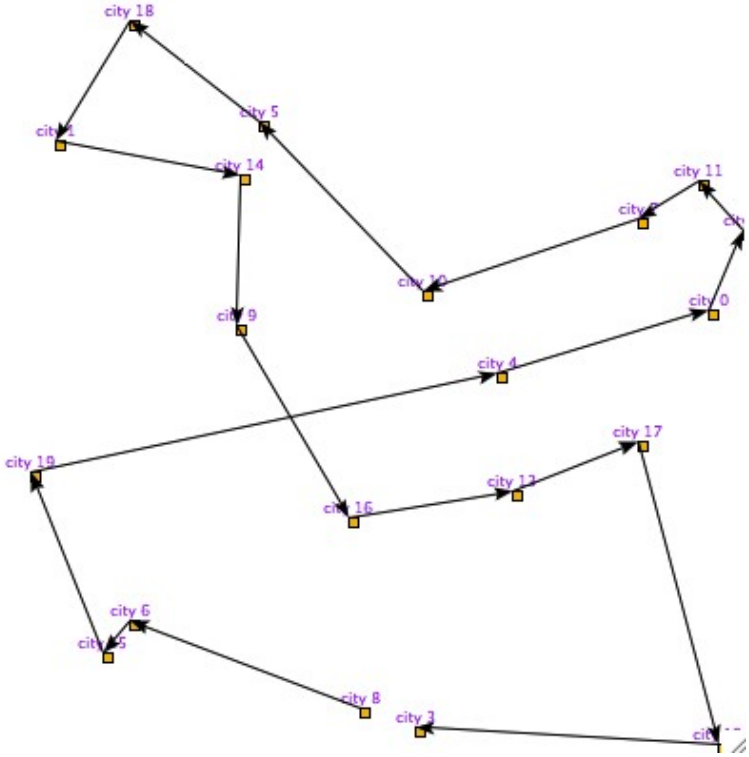
```
1.  import math
2.  import random
3.  from Tkinter import Tk, Canvas, BOTH
4.
5.
6.  def euclidean_distance(coord1, coord2):
7.    lat1=coord1[0]
8.    lon1=coord1[1]
9.    lat2=coord2[0]
10.   lon2=coord2[1]
11.   return math.sqrt((lat1-lat2)**2+(lon1-lon2)**2)
12.
13. # calcula la distancia cubierta por una ruta
14. def evaluate_route(ruta):
15.   total=0
16.   for i in range(0,len(ruta)-1):
17.     ciudad1=ruta[i]
18.     ciudad2=ruta[i+1]
19.     total=total+euclidean_distance(coord[ciudad1], coord[ciudad2])
20.   ciudad1=ruta[i+1]
21.   ciudad2=ruta[0]
22.   total=total+euclidean_distance(coord[ciudad1], coord[ciudad2])
23.   return total
24.
25.
26. def hill_climbing():
27.   # crear ruta inicial aleatoria
28.   ruta=[]
29.   for ciudad in coord:
30.     ruta.append(ciudad)
31.   random.shuffle(ruta)
32.
33.   mejora=True
34.   while mejora:
35.     mejora=False
36.     dist_actual=evaluate_route(ruta)
37.     # evaluar vecinos
38.     for i in range(0,len(ruta)):
39.       if mejora:
40.         break
41.       for j in range(0,len(ruta)):
42.         if i!=j:
43.           ruta_tmp=ruta[:]
44.           ciudad_tmp=ruta_tmp[i]
45.           ruta_tmp[i]=ruta_tmp[j]
46.           ruta_tmp[j]=ciudad_tmp
47.           dist=evaluate_route(ruta_tmp)
48.           if dist<dist_actual:
49.             # encontrado vecino que mejora el resultado
```

```
            mejora=True
            ruta=ruta_tmp[:]
            break

  return ruta


if __name__ == "__main__":
  num_stops=15
  res_x=400
  res_y=400
  coord={}
  for i in range(num_stops):
    coord['city
'+str(i)]=(random.randint(1,res_x),random.randint(1,res_y))

  ruta = hill_climbing()
  print ruta
  print "Distancia total: " + str(evaluate_route(ruta))

  # mostrar mapa
  root = Tk()
  canvas = Canvas()
  # dibujar ruta
  last_city=()
  for ciudad in ruta:
    x=coord[ciudad][0]
    y=coord[ciudad][1]
    canvas.create_rectangle(x, y, x+5, y+5, outline="#000", fill="#fb0")
    canvas.create_text(x, y-5, text=ciudad, fill="purple",
font=("Helvectica", "8"))
    if last_city:
      canvas.create_line(last_city[0],last_city[1],x,y, arrow="last")
    last_city=(x,y)
  canvas.pack(fill=BOTH, expand=1)
  root.geometry(str(res_x)+"x"+str(res_y))
  root.mainloop()
```

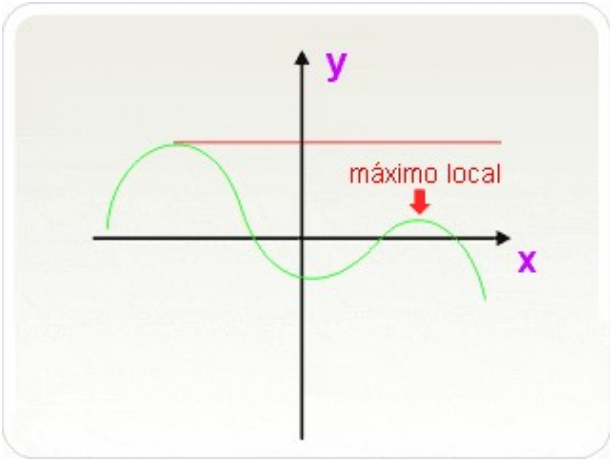
Ejecutamos el programa y obtuvimos una más que correctísima solución al problema TSP con 15 ciudades.



¿Nos atrevemos con 20 ciudades? A ver si es verdad que no tenemos que esperar 77 millones de años.
Al ejecutar el programa con 20 ciudades el resultado fue el siguiente.



Uhmmm, creo que empezamos a tener problemas -dije. Fijesé en ese cruce de flechas, no es buen presagio. Creo que hemos caído en un mínimo local.
¿Un qué? -dijo el Sr. Lego.
Imagine que sigue perdido en la misma colina de la que hablabamos antes. Usted llega a un pico, pero ¿podría asegurar que ese es el pico más alto de la colina?
No -respondió el Sr. Lego. Podría haber otro más alto, pero no podría verlo.
Exacto. Esto es lo que nos ha ocurrido al ejecutar el programa con las 20 ciudades. Hemos llegado a un mínimo (o máximo) relativo, también conocido como mínimo (o máximo) local.



Una posible solución es lanzar el algoritmo varias veces, pero cada vez empezando desde un estado distinto, y quedándonos con el mejor resultado de todos. Es lo que se llama Hill Climbing iterativo.
El Sr. Lego corrió de nuevo y modificó ligeramente el programa.

```
1.  import math
2.  import random
3.  from Tkinter import Tk, Canvas, BOTH
4.
5.
6.  def euclidean_distance(coord1, coord2):
7.    lat1=coord1[0]
8.    lon1=coord1[1]
9.    lat2=coord2[0]
10.   lon2=coord2[1]
11.   return math.sqrt((lat1-lat2)**2+(lon1-lon2)**2)
12.
13. # calcula la distancia cubierta por una ruta
14. def evaluate_route(ruta):
15.   total=0
16.   for i in range(0,len(ruta)-1):
17.     ciudad1=ruta[i]
18.     ciudad2=ruta[i+1]
19.     total=total+euclidean_distance(coord[ciudad1], coord[ciudad2])
20.   ciudad1=ruta[i+1]
21.   ciudad2=ruta[0]
22.   total=total+euclidean_distance(coord[ciudad1], coord[ciudad2])
23.   return total
24.
25.
26. def hill_climbing():
27.   # crear ruta inicial aleatoria
28.   ruta=[]
```

```
29.   for ciudad in coord:
30.     ruta.append(ciudad)
31.   random.shuffle(ruta)
32.
33.   mejora=True
34.   while mejora:
35.     mejora=False
36.     dist_actual=evaluate_route(ruta)
37.     # evaluar vecinos
38.     for i in range(0,len(ruta)):
39.       if mejora:
40.         break
41.       for j in range(0,len(ruta)):
42.         if i!=j:
43.           ruta_tmp=ruta[:]
44.           ciudad_tmp=ruta_tmp[i]
45.           ruta_tmp[i]=ruta_tmp[j]
46.           ruta_tmp[j]=ciudad_tmp
47.           dist=evaluate_route(ruta_tmp)
48.           if dist<dist_actual:
49.             # encontrado vecino que mejora el resultado
50.             mejora=True
51.             ruta=ruta_tmp[:]
52.             break
53.
54.   return ruta
55.
56.
57. if __name__ == "__main__":
58.   num_stops=20
59.   res_x=400
60.   res_y=400
61.   coord={}
62.   for i in range(num_stops):
63.     coord['city
    '+str(i)]=(random.randint(1,res_x),random.randint(1,res_y))
64.
65.   best_route=[]
66.   for i in range(20):
67.     ruta = hill_climbing()
68.     if best_route:
69.       if evaluate_route(ruta)<evaluate_route(best_route):
70.         best_route=ruta[:]
71.     else:
72.       best_route=ruta
73.
74.   ruta = best_route
75.   print ruta
76.   print "Distancia total: " + str(evaluate_route(ruta))
77.
78.   # mostrar mapa
79.   root = Tk()
80.   canvas = Canvas()
81.   # dibujar ruta
```

```
82.   last_city=()
83.   for ciudad in ruta:
84.     x=coord[ciudad][0]
85.     y=coord[ciudad][1]
86.     canvas.create_rectangle(x, y, x+5, y+5, outline="#000", fill="#fb0")
87.     canvas.create_text(x, y-5, text=ciudad, fill="purple",
    font=("Helvectica", "8"))
88.     if last_city:
89.       canvas.create_line(last_city[0],last_city[1],x,y, arrow="last")
90.     last_city=(x,y)
91.   canvas.pack(fill=BOTH, expand=1)
92.   root.geometry(str(res_x)+"x"+str(res_y))
93.   root.mainloop()
```

Al ejecutarlo vimos que el programa ya se desenvolvía bien con un número de ciudades mayores. Aun así, mientras más ciudades, más posibilidades de cruce de caminos encontrábamos, y más iteraciones necesitábamos para obtener un buen resultado.

¿Y no hay forma de mejorar el algoritmo para que sea más eficiente con un número de ciudades muy alto? -preguntó el Sr. Lego.

Lo hay -dije- pero se acerca la hora de comer, así que si le parece, otro día exploramos otras técnicas que nos ayuden a evitar caer en estos mínimos (o máximos) locales.