Evaluación del tiempo de ejecución de un algoritmo complejo

¿No se le podría haber puesto un nombre?

Por ejemplo, "un algoritmo para conectar una matriz".

21 de diciembre de 2016

Índice

Este trabajo debería haber presentado la N y la variable Y (tiempo de m resoluciones), aunque se podrían haber usado otras respuestas, como el número de iteraciones necesarias para conectar una matriz. En este caso no hubiera hecho falta repetir m veces para tener precisión. Mostrar un gráfico con la N y la Y, para poder apreciar la relación entre ambas (seguramente no lineal, según el coste teórico). Cambiar la X tomando N^2 log(N), según lo que dice la teoría, con un factor de escala preferiblemente, y volver a mostrar el plot de esta X con la Y para ver si al menos es lineal. Etc; se puede seguir probando con otras transformaciones porque además posiblemente hay heterocedasticidad. Estimar con lm() el modelo lineal, interpretar los parámetros hallados ....

1. Resumen

1.1. Objetivo

Comprobar que el tiempo de ejecución de un algoritmo complejo que trabaja con matrices cuadradas de grandes dimensiones se corresponde con su coste computacional teórico.

1.2. Métodos

Ejecutamos el programa para matrices de dimensiones $N \times N$, con $N = 50, 100, \ldots, 1000$ y medimos el tiempo que tarda cada vez.

1.3. Resultados

Los cálculos nos corroboran lo esperado. El tiempo de ejecución experimental coincide con el teórico, aunque hay cierto alejamiento de la predicción teórica cuanto mayor sea el tamaño de las matrices a probar (*outliers*).

2. Introducción

Supongamos que tenemos una matriz cuadrada M de dimensiones $N \times N$ de *booleanos*. Esto es, que cada entrada de la matriz tiene dos únicos valores: *marcado* o *no marcado*; *cierto* o *falso*, etcétera. Inicialmente, todas las posiciones de M están sin marcar.

Diremos que M está *conectada* si y solo si existe un camino de posiciones marcadas adyacentes desde la fila superior de M hasta la fila inferior de M (dos casillas son adyacentes si están a distancia 1 y sin contar diagonales). En la Figura 1 se puede ver un ejemplo de matriz conectada.

El algoritmo se puede resumir en los siguientes pasos:

1. Vaciar una matriz M de dimensiones $M \times M$.
2. Hasta que M esté *conectada*: marcar una casilla aleatoria de M que no esté ya previamente marcada.

```
00010
01110
01001
01100
00101
```

Figura 1: Matriz 5×5 conectada

El objetivo es comprobar que el tiempo de ejecución de este algoritmo (experimental) se corresponde realmente con su coste computacional (análisis teórico).

3. Métodos

Para poder estudiar el tiempo de ejecución de este algoritmo, el primer paso fue diseñar el algoritmo. Después, se creó un pequeño *script* que repite el algoritmo 1000 veces para matrices de tamaño 50, 100, ..., 1000, y medimos el tiempo de ejecución con la comanda `time` de Unix. Decidimos repetir 1000 veces para poder medir los tiempos con mayor facilidad. La elección de los tamaños (50 ... 1000) también se hizo por facilidad.

Los experimentos y la recogida de datos se realizaron utilizando un ordenador con un microprocesador Intel Core i3 a 3,2 GHz y 4 GB de memoria RAM. Las representaciones gráficas de las premisas y el modelo lineal se realizaron con el *software* R versión 3.3.2.

Los valores recogidos fueron el tiempo que tardaba en resolverse el problema para los distintos tamaños de la matriz.

3.1. Variables

El estudio presenta la variable explicativa X (también llamada N) que es el tamaño de las matrices. Y obtendremos una variable respuesta, Y, que nos indicará el tiempo de ejecutar el algoritmo repetidas veces en función de la entrada.

Para el análisis de datos se realizó la siguiente transformación sobre la variable predictiva X: $x \to \beta x^2 \log(x)$; pues el algoritmo es $\Theta(\cdot)$ de la expresión anterior. La constante multiplicativa es $\beta = 0{,}0001$, y se determinó por observación.

Comprobación de las premisas: ¿Qué premisas?

¡No es correcto incluir un gráfico de los resultados en la parte de Métodos!

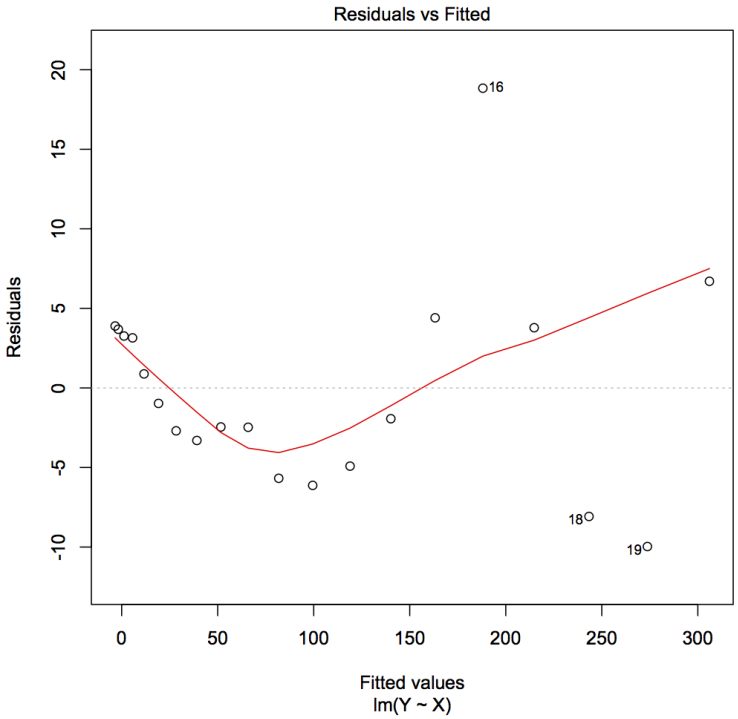

Figura 2: Residuals vs Fitted

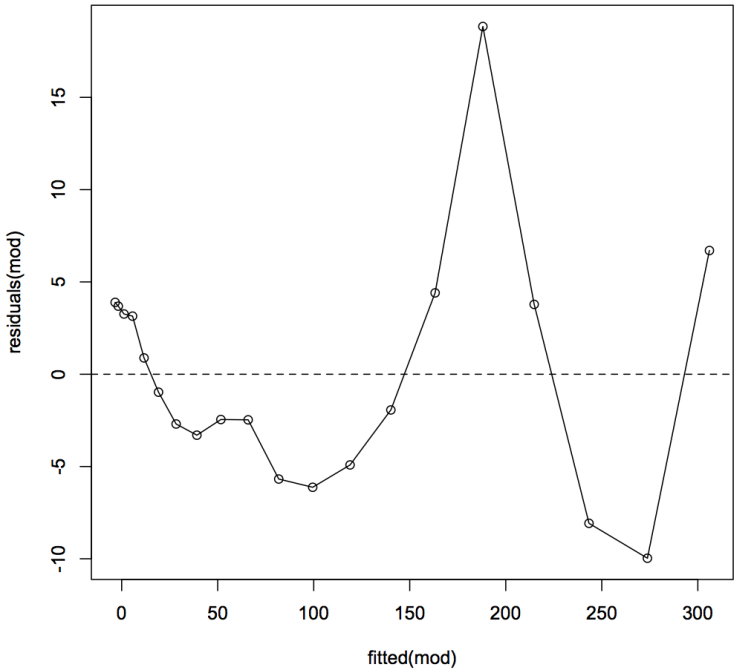

Figura 3: Independence

residuals vs fitted no es el gráfico para valorar la premisa de independencia de los residuos

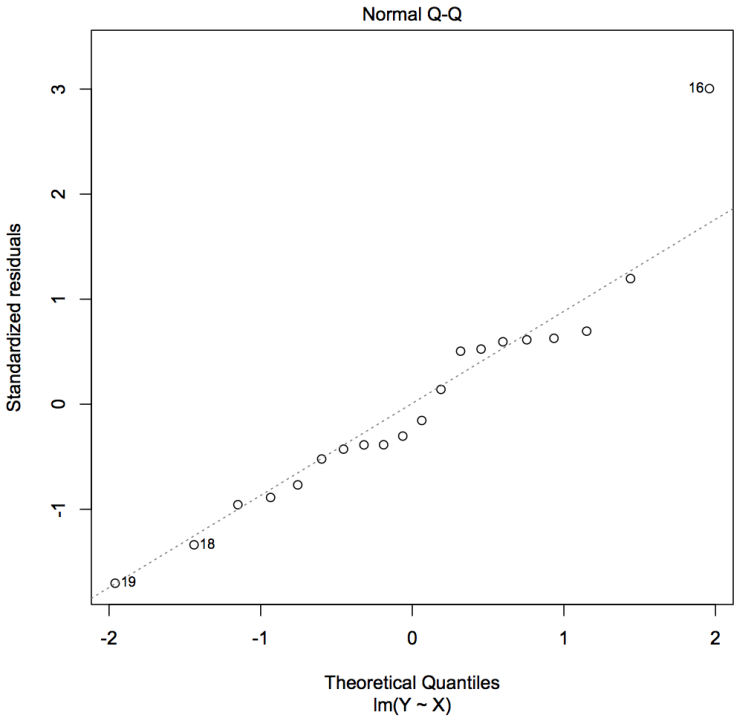

Figura 4: NormalQ-Q

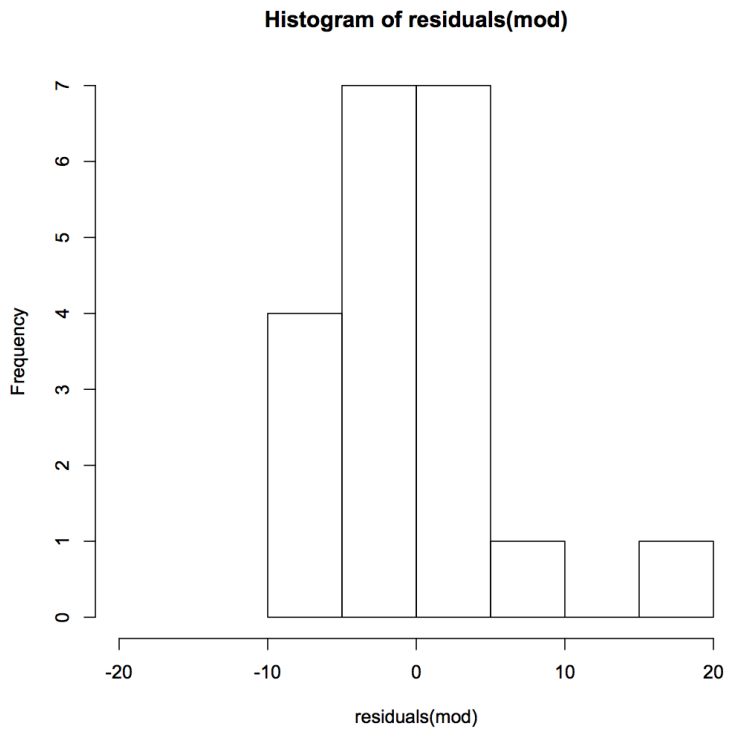

Figura 5: Histograma de residuos

Todo este análisis debería ir en la parte de resultados

Linealidad y homoscedasticidad Como podemos ver en la Figura 2, los residuos frente a los valores predecidos muestran que con la transformación de la varaible X conseguimos cumplir la premisa de linealidad. Sin embargo, observamos que la homoscedasticidad no se valida con este gráfico, ya que podemos ver claramente que para los valores pequeños la desviación es mucho menor que para los valores más extremos. Creemos que esto se debe a que, para tamaños más grandes de las matrices, las limitaciones que ofrece la máquina dejan una marca más significativa que para tamaños pequeños.

Independencia Observando la Figura 3, podemos interpretar que los datos dependen de nuestra variable con una correlación positiva, ya que el tiempo que observamos al ejecutar el algoritmo depende directamente del tamaño de la matriz.

Interpretación errónea de lo que es la premisa

Normalidad El histograma y el QQ-Norm de los residuos muestran que estos siguen una distribución normal, pese a que podemos observar la aparición de un *outlier*. Tanto el histograma como el QQ-Plot nos indican que los residuos pueden ser modelados según una distribución normal. Véanse las figuras 4 y 5.

3.2. Análisis

El análisis se basó en comparar con un modelo lineal la variable predictiva con los valores obtenidos al ejecutar el algoritmo.

Para la ejecución de este experimento tuvimos que descartar el tiempo que el proceso que ejecutaba el algoritmo pasaba en sistema trabajando con la paginación debido a que, para los tamaños más grandes, producía diferencias demasiado significativas; todas ellas, ajenas al propio algoritmo.

Me extraña que este algoritmo consuma tanta memoria.

4. Resultados

Podemos comprovar que, excluyendo algun caso de outlier, el tiempo de ejecución del algoritmo en el ordenador se ajusta al valor teorico: $\Theta(mn^2 \log(n))$.

Véase el último anexo.

El último anexo es una tabla de datos.

No hay análisis en ningún sitio. Debe haber un análisis estadístico, y una estimación de los parámetros, y una interpretación basada en inferencia estadística y los riesgos manejados.

5. Discusión

5.1. Conclusiones

Si no hay resultados, no hay confirmación.

Los resultados confirman la teoria conocida de que el tiempo de ejecución del algortimo es del orden $\Theta(mn^2 \log(n))$ donde n es el tamaño de la entrada de la matriz.

Aunque como podemos observar en el apartado anterior, cuanto más grande es el tamaño de la matriz más tiempo necesita el proceso en modo sistema, para las matrices con mayores dimensiones vemos que este tiempo asciende a 30 segundos o más.

Este tiempo de sistema puede variar dependiendo en los componentes de la máquina donde se esta ejecutando el algortimo.

5.2. Limitaciones

A la hora de hacer este trabajo, nos hemos encontrado con algunas limitaciones. La más importante ha sido el tiempo: no hemos tenido todo el tiempo que habríamos deseado para redactar este informe.

Además, inicialmente hemos tenido problemas con la RAM. Esta era insuficiente en la máquina virtual en que ejecutábamos el programa a pesar de que el algoritmo es razonablemente eficiente; por ello, hemos tenido que ejecutar el programa en un ordenador personal.

Aún habiendo ejecutado el algoritmo en un ordenador personal este también presenta diversas limitaciones, como los procesos que se estan ejecutando concurrentemente durante la ejecución del algoritmo, los componentes del ordenador, las aplicaciones que hay abiertas durante el proceso de ejecución etc...

5.3. Trabajo Futuro

Por otra parte, como trabajo futuro nos hubiese gustado comparar diferentes modalidades de conexión (por ejemplo: conexión *simple* (solo norte, sur, este y oeste) *versus* conexión *compleja* (añadiendo diagonales)), comparar las variaciones entre experimentos con matrices de las mismas dimensiones, etcétera.

A. Anexo: código del proyecto

El programa que se encarga de generar los datos que hemos presentado anteriormente lo hemos escrito en C++. Aquí incluimos una versión del código de este programa. En particular, obviamos el tratamiento de errores y la separación estructural de clases funcionales (cosa que hacemos correctamente en nuestros archivos).

Para compilar el proyecto se debe compilar con el comando `c++` y la opción `-std=c++11`.

A.1. Programa para medir tiempos

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
using namespace std;

int main()
{
    for (int i = 50; i <= 4000; i += 50) {
        char buff[80];
        sprintf(buff, "Conexion %d\n", i);
        write(1, &buff, strlen(buff));
        if (fork() == 0) {
            sprintf(buff, "time ./main.o %d > d2.dat", i);
            system(buff);
            exit(0);
        }
        while(waitpid(-1,NULL,0)>0);
    }
}
```

A.2. Programa principal

```
#include <iostream>
#include <algorithm>
using namespace std;

#include "experimento.cc"

int iteraciones = 1000;      // Número de repeticiones
bool conexion_simple = true; // Tipo de conexión

// El tamaño de la matriz, N
// se lee como argumento
// con ./main n
int main(int argc, char* argv[])
{
  // Inicializar la semilla aleatoria
  srand(time(NULL));

  // Cabecera para R y generación de resultados
  cout << "conexiones" << endl;
  for (int i = 0; i < iteraciones; ++i) {
    cout << experimento(atoi(argv[1]), conexion_simple) << endl;
  }
}
```

A.3. Experimento

```
// experimento.cc
#include "UF.cc"

// Pre:  cierto
// Post: devuelve "la posición (i, j) es correcta en una matriz n x n"
bool valido(int i, int j, int n)
{
  return 0 <= i and i < n and 0 <= j and j < n;
}
```

```
// Pre:  cierto
// Post: devuelve la posición de un vector de tamaño n x n equivalente
//       a la posición (i, j) de una matriz n x n.
int real(int i, int j, int n)
{
  return i*n + j;
}

// Pre:  M es n x n; uf es un union-find correcto de M.
// Post: devuelve "hay un camino que conecta M"
bool matriz_conectada(vector<vector<bool>>& M, UF& uf, int n)
{
  return uf.connected(n*n, n*n + 1);
}

// Pre:  M es n x n; uf es un union-find correcto de M.
// Post: devuelve el número de conexiones necesarias para conectar M.
int rellenar(int n, bool conexion_simple, UF& uf, vector<vector<bool>>& M)
{
  int pasos = 0;
  while (not matriz_conectada(M, uf, n)) {
    int i, j;
    do {
      i = rand() % n; // 0..n-1
      j = rand() % n; // 0..n-1
    } while (M[i][j]);

    // (i, j) es una posicion aleatoria sin ocupar

    M[i][j] = true;
    ++pasos;

    int pos = real(i, j, n);

    // Conectar con los adyacentes
    if (valido(i + 1, j, n) and M[i + 1][j]) {
      uf.unir(real(i + 1, j, n), pos);
    }
    if (valido(i - 1, j, n) and M[i - 1][j]) {
```

```
      uf.unir(real(i - 1, j, n), pos);
    }
    if (valido(i, j + 1, n) and M[i][j + 1]) {
      uf.unir(real(i, j + 1, n), pos);
    }
    if (valido(i, j - 1, n) and M[i][j - 1]) {
      uf.unir(real(i, j - 1, n), pos);
    }

    // Si hago una conexión compleja,
    // conectar, además, en diagonal
    if (not conexion_simple) {
      if (valido(i + 1, j + 1, n) and M[i + 1][j + 1]) {
        uf.unir(real(i + 1, j + 1, n), pos);
      }
      if (valido(i - 1, j - 1, n) and M[i - 1][j - 1]) {
        uf.unir(real(i - 1, j - 1, n), pos);
      }
      if (valido(i + 1, j - 1, n) and M[i + 1][j - 1]) {
        uf.unir(real(i + 1, j - 1, n), pos);
      }
      if (valido(i - 1, j + 1, n) and M[i - 1][j + 1]) {
        uf.unir(real(i - 1, j + 1, n), pos);
      }
    }

    // En el caso de estar en la primera o última filas,
    // conectar con los elementos n*n o n*n + 1 del union-find.
    // Estos dos vértices son especiales; se usan solo
    // de manera virtual aquí.
    if (i == 0) {
      uf.unir(n*n, pos);
    } else if (i == n - 1) {
      uf.unir(n*n + 1, pos);
    }
  }
  return pasos;
}
```

```
// Pre:  n >= 2
// Post: devuelve el número de conexiones necesarias para conectar
//       una matriz de dimensiones n x n.
int experimento(int n, bool conexion_simple)
{
  UF uf(n*n + 2);
  vector<vector<bool>> M(n, vector<bool>(n, false));
  return rellenar(n, conexion_simple, uf, M);
}
```

A.4. Clase *union-find*

```
// UF.cc
#include <vector>
using namespace std;

class UF {
public:
  UF() { }

  // Pre: n > 0
  // Post: construye un union-find de n elementos
  UF(int n)
  {
    v = vector<int>(n);
    sizes = vector<int>(n);
    for (int i = 0; i < n; ++i) {
      v[i] = i;
      sizes[i] = 1;
    }
  }

  // Pre: 0 <= x < tamaño del union-find
  // Post: devuelve el representante de x
  int find(int x) const
  {
    while (x != v[x])
      x = v[x];
```

```
    return x;
  }

  // Pre: 0 <= x, y < tamaño del union-find
  // Post: devuelve si x e y están conectados
  bool connected(int x, int y) const
  {
    return find(x) == find(y);
  }

  // Pre: 0 <= x, y < tamaño del union-find
  // Post: une x e y
  void unir(int x, int y)
  {
    x = find(x);
    y = find(y);

    if (x != y) {
      if (sizes[x] < sizes[y]) {
        v[x] = y;
        sizes[y] += sizes[x];
      }
      else {
        v[y] = x;
        sizes[x] += sizes[y];
      }
    }
  }

private:
  // Vector de conexiones
  vector<int> v;

  // Tamaño de los árboles
  vector<int> sizes;
};
```

A.5. Eficiencia del programa

Si hemos podido generar tantos datos en relativamente poco tiempo, esto es porque el algoritmo que utilizamos es muy eficiente.

Sean n el tamaño de la matriz y m el número de repeticiones del experimento. El coste del algoritmo es m veces el coste de conectar una matriz. Para conectar una matriz, necesitamos $\Theta(n^2)$ pasos. Como la selección de siguiente casilla la realizamos de manera totalmente azarosa y despreocupada según el siguiente código,

```
int i, j;
do {
    i = rand() % n;
    j = rand() % n;
} while (M[i][j]);
```

el coste de encontrar todas las casillas libres a ocupar es $\Theta(n^2 \log n)$.[1] El resto de operaciones que se ejecutan para cada casilla son $\Theta(1)$, pues el mayor coste, que es el de inserción en el *union-find*, es $\Theta(\log^*(n^2))$, que podemos asumir constante. Solo nos queda la condición del `while`, que es aproximadamente constante, por la misma razón.

En definitiva, el coste es $\Theta(mn^2 \log n)$.

Vale la pena hacer esta valoración porque, si la comprobación de conexión de las matrices las hiciésemos de manera natural, con n DFS —o DFS—, el coste por cada iteración sería cuadrático en vez de (casi) constante y, por tanto, el coste total aumentaría hasta $\Theta(mn^4 \log n)$. En definitiva, el programa resultante sería demasiado lento para generar todos los datos requeridos en un tiempo, siquiera, mucho más grande.

B. Anexo: código R

```
X <- c(0.49, 1.914, 4.481, 8.814, 12.523,
18.216, 25.655, 35.862, 49.203, 63.385,
76.107, 93.342, 114.005, 138.225, 167.615,
```

[1] Este resultado se deriva de la esperanza de la suma de varias distribuciones geométricas. Véase el *Coupon collector's problem* para más información.

```
206.915, 218.565, 235.257, 263.781, 312.742);

Y <- c(0.424742501, 2, 4.896205333, 9.204119983, 14.98712505,
22.29409129, 31.16483354, 41.63295986, 53.7275534, 67.47425011,
82.89597136, 100.013445, 118.8455893, 139.409804, 161.7221961,
185.7977592, 211.6505174, 239.2936433, 268.7395554, 300)

mod = lm(formula = Y~X);
summary(mod);
plot(mod);

# Para el plot de indepencencia
plot(fitted(mod),residuals(mod))
abline(h=0,lty=2)
lines(smooth.spline(fitted(mod),residuals(mod)))

# Histograma
hist(residuals(mod),xlim=c(-20,20))
```

C. Anexo: datos

Y	X transformada	Tiempo + sistema
0.49	0.424742501	0.507
1.914	2	1.951
4.481	4.896205333	4.683
8.814	9.204119983	10.751
12.523	14.98712505	12.791
18.216	22.29409129	18.516
25.655	31.16483354	26.472
35.862	41.63295986	38.452
49.203	53.7275534	55.919
63.385	67.47425011	71.878
76.107	82.89597136	84.958
93.342	100.013445	104.462
114.005	118.8455893	127.616
138.225	139.409804	158.371
167.615	161.7221961	199.447
206.915	185.7977592	232.679
218.565	211.6505174	253.995
235.257	239.2936433	254.526
263.781	268.7395554	283.511
312.742	300	240,524