Practica Pagerank

Dia 3

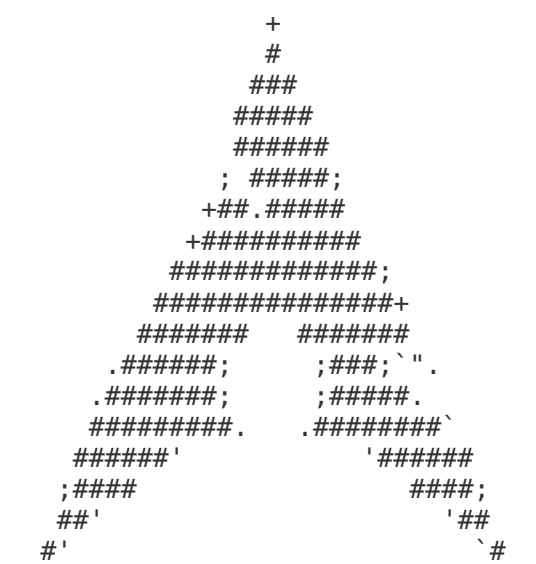
Miguel Garcia Lafuente

Índice de contenido

Para esta practica se ha usado un ordenador portatil con las siguentes caracteristicas:

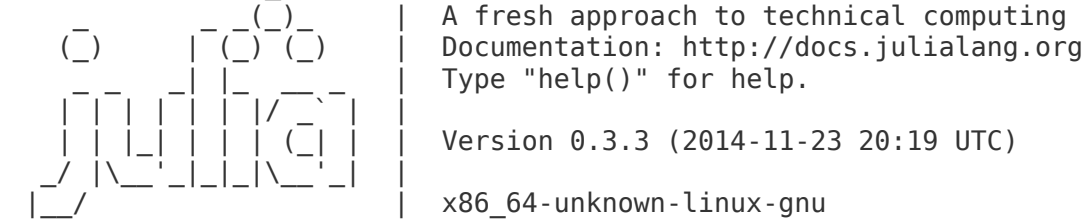
```
S.O: ArchLinux 64 bits.
Memoria RAM: 8076456 kB. (unos 7.7GB)
CPU: Intel(R) Core(TM) i5-4200M CPU @ 2.50GHz – 64 bits

~ » archey3
```



Version de Julia 0.3.3:

```
~ » julia
rock@neurotiko
```




```
julia>
```

Como apunte, a pesar de tener 4 cores, Julia esta en estado de crecimiento y por ahora solo funciona en un core, por lo que a efectos practico el procesador es de un solo nucleo a 2.5 GHz.

Toda la practica esta vez lo he pasado tambien con un programa que se llama “Ijulia notebook”, que es un modulo para el “Ipython notebook”, para poder subir codigo, y ejecutar codigo, y que quede en un formato perfecto para presentar.

Como se puede observar, esta todo el codigo, pero tambien las ejecuciones y los resultados.

Link al notebook:
http://nbviewer.ipython.org/github/rockneurotiko/rockneurotiko.github.io/blob/master/Universidad/AplicacionesNumericas/PageRank/Dia3/AplNum-Prac3.ipynb

1. Matrices dispersas (mdis_A)

Despues de optimizaciones, el codigo que mejor genera las matrices dispersas es el siguiente:

```
 1 function mdis_A(N::Int64, R::Int64=5)
 2     p = randi(N,1,R*N)
 3     q = randi(N,1,R*N)
 4     A = sparse(p,q,1.0,N,N)
 5     Nj = sum(A,1)
 6     i,j = findn(A)
 7     I,J,V = findnz(A)
 8     for idx in 1:length(V)
 9         V[idx] /= Nj[J[idx]]
10     end
11     return sparse(i,j,V,N,N)
12 end
```

Unas pruebas para ver la rapidez:

```
julia> @time mdis_A(10^1);
elapsed time: 4.9913e-5 seconds (12008 bytes allocated)

julia> @time mdis_A(10^2);
elapsed time: 0.000676117 seconds (117848 bytes allocated)

julia> @time mdis_A(10^3);
elapsed time: 0.001351825 seconds (1279416 bytes allocated)

julia> @time mdis_A(10^4);
elapsed time: 0.015278099 seconds (13303720 bytes allocated)

julia> @time mdis_A(10^5);
elapsed time: 0.20780851 seconds (133544392 bytes allocated, 16.08% gc time)

julia> @time mdis_A(10^6);
elapsed time: 2.801159681 seconds (1335940344 bytes allocated, 5.30% gc time)
```

Se puede comprobar que la memoria alocada va creciendo (mas o menos) multiplicado por 10.

La matriz 10^6 nos ha “alocado” 1274.05 MB (1GB de RAM), asi que 10^7 necesitaria unos 12.74 GB de RAM, que mi ordenador no tiene (al intentar generarla murio).

A parte de las limitaciones fisicas, el tiempo esta muy bien, para la matriz mas grande que hemos podido, solo ha tardado 2.8 segundos en generarla.

2. Funcion calculo_PR

Para llevarlo a cabo, usaremos una funcion auxiliar para generar el vector “v”.

```
function calculate_v(dj::Array{Int64,2},
                     e::Array{Float64,2}=ones(1,size(dj,2)),
                     alpha::Float64=0.85)
    return (alpha * dj) + ((1-alpha) * e)
end
```

Y ya, la funcion importante es la siguiente:

```
function calculo_PR(A::SparseMatrixCSC{Float64,Int64},
                    tolerancia::Float64=1e-12,
                    n_iter::Int64=500,
                    r::Array{Float64,2}=ones(size(A,1),1),
                    alpha::Float64=0.85,
                    Nj::Array{Float64,2}=sum(A,1))
    n = size(A,1)
    last = zeros(n)
    dj::Array{Int64,2} = [x == 0 ? 1:0 for x=Nj]'
    v = calculate_v(dj)
    e1 = ones(size(A,1),1)
    i = 0
    while i < n_iter && maximum(abs(r-last)) > tolerancia
        i += 1
        last = r
        last = last ./ norm(last)
        esc = v*r
        r = alpha*A*last + (1/n * e1 * esc)
    end
    return (pagerank(r), norm(r-last,1))
end
```

La funcion es muy sencilla, y casi se lee sola, los tipos estan puestos para mejorar el compilador JIT que usa Julia, ya que sabe exactamente que tipos estan y puede reservar la memoria necesaria.

Para hacer las pruebas, vamos a usar la siguiente funcion “test” que hace uso de la macro “@time”

```
function test(N::Int64)
    @time a = mdis_A(N);
    @time calculo_PR(a);
end
```

Veamos los resultados:

```
julia> x, p = test(10^1);
elapsed time: 3.5674e-5 seconds (12408 bytes allocated)
elapsed time: 6.7705e-5 seconds (71112 bytes allocated)

julia> p
2.3168411633633923e-12

julia> sizeof(x)
80

julia> x, p = test(10^2);
elapsed time: 0.000241763 seconds (117528 bytes allocated)
elapsed time: 0.000381824 seconds (524712 bytes allocated)

julia> p
6.7608037224164974e-12

julia> sizeof(x)
800

julia> x, p = test(10^3);
elapsed time: 0.001347145 seconds (1279816 bytes allocated)
elapsed time: 0.00244701 seconds (4942488 bytes allocated)

julia> p
1.1357452131544044e-10

julia> sizeof(x)
8000

julia> x, p = test(10^3);
elapsed time: 0.001331278 seconds (1279736 bytes allocated)
elapsed time: 0.002491017 seconds (4943576 bytes allocated)

julia> p
9.302403205374565e-11

julia> sizeof(x)
8000

julia> x, p = test(10^4);
elapsed time: 0.033614157 seconds (13306024 bytes allocated, 59.65% gc time)
elapsed time: 0.034519361 seconds (53613064 bytes allocated, 45.75% gc time)

julia> p
7.280327512723672e-11

julia> sizeof(x)
80000

julia> x, p = test(10^5);
elapsed time: 0.208644534 seconds (133543784 bytes allocated, 23.95% gc time)
elapsed time: 0.362151298 seconds (492812344 bytes allocated, 37.64% gc time)

julia> p
3.024801713898302e-9
```

```
julia> sizeof(x)
800000

julia> x, p = test(10^6);
elapsed time: 2.352491702 seconds (1335943208 bytes allocated, 10.67% gc time)
elapsed time: 4.582350449 seconds (4928012344 bytes allocated, 18.27% gc time)

julia> p
1.5131614666287745e-8

julia> sizeof(x)
8000000

julia> x, p = test(2*10^6);
elapsed time: 4.934265793 seconds (2671942792 bytes allocated, 10.55% gc time)
elapsed time: 8.513741088 seconds (9568006472 bytes allocated, 13.26% gc time)

julia> p
6.03260282657685e-8

julia> sizeof(x)
16000000
```

	Tiempo (seg)	Memoria (MB)	Precision $\|\|Gx-x\|\|_1$
10^1	6,9859e-5	0.067 MB	2.31684116336339e-12
10^2	0,000374197	0.5 MB	6.76080372241649e-12
10^3	0,001367687	4.7 MB	1.13574521315440e-10
10^4	0,026235145	51.12 MB	9.30240320537456e-11
10^5	0,369947148	469.98 MB	3.024801713898302e-9
10^6	4,582350449	4699,71 MB	1.513161466628774e-8
2*10^6	8,513741088	9124.64 MB	6.03260282657685e-8

* La memoria no es la memoria total usada, sino la memoria que ha tenido que alocar en total en la ejecucion, no sabia a que se referia la practica con la Memoria usada (memoria de la matriz?, memoria del vector x?), asi que he decidido poner esta, que es lo maximo que ha usado el metodo.

Como se puede observar, la velocidad del algoritmo es muy buena, y eso que solo se ejecuta en un core, si se paralelizase el algoritmo seria muy eficiente.

Se puede comprobar tambien como el tiempo de ejecucion del algoritmo es mas o menos el doble de la creacion de la matriz A con mdis_A.

Notese tambien que la precision, según se va aumentando el tamaño de la matriz A, la precision disminuye notablemente, aunque sigue siendo aceptable.