Tópicos de Programación para Científicos Computacionales

Informe del Trabajo Práctico Final

Eduador Agustín Rosselot[1−3] y Maximiliano Jose Perez Frasette[2−3]

[1]earosselot@gmail.com [924/11]

[2]maxi.perezfrasette@gmail.com [862/12]

[3]Grupo 1 - Alumnos de doctorado del Departamento de Geología

Diciembre 2019

1. Ejercicio 1: Dividir y Conquistar

El archivo donde se encuentran las implementaciones del ejercicio 1 está incluído en el repositorio con el nombre *Ejercicio1.py*, dentro de la carpeta **src_ej1**. En esa misma carpeta, se encuentra el gráfico generado (bajo el nombre de *GraficoEj1.png*)y un archivo llamado *datosGrafico1.txt* , con los tiempos de cómputo obtenidos. También está incluído el archivo *datitos.txt*, que son los datos suministrados junto con el tp.

El código donde están las funciones implementadas se encuentra divido en cuatro secciones. En la primer sección se encuentran las funciones principales, en la segunda sección están todas las funciones auxiliares necesarias para implementar las funciones principales. En la tercer sección se encuentran las funciones implementadas para analizar los diferentes tiempos de ejecución. Por último, en la cuarta sección están los comandos para obtener el gráfico. Cada sección está dividida por un comentario que las identifica claramente.

En la sección número uno están las tres funciones principales pedidas por el enunciado del tp, `listadepuntos(fn)`, `distanciaMinima(l)` y `distancia MinimaDyC(l,algoritmo)`. La implementación de las primeras dos funciones es relativamente sencilla. `Listadepuntos(fn)` utiliza la función auxiliar `entero(a)`, que convierte *strings* en *floats*. `DistanciaMinima(l)` recibe una lista de tuplas e itera la cantidad de veces necesaria para calcular la distancia euclídea entre todos los puntos de la lista.Al finalizar, devuelve el par de puntos más cercanos. Está función necesita como parámetro una lista de al menos dos elementos, en ese caso devuelve ese par de puntos.Además, utiliza la función auxiliar `distancia(p1,p2)`, la cual calcula la distancia euclídea entre dos pares de coordenadas. Por último, `distanciaMinimaDyC(l, algoritmo)` también busca el par de puntos más cercano dentro de una lista de puntos l, pero utiliza la técnica de Dividir y Conquistar. Esta función está dividida en dos bloques. En el primer bloque, ordena a la lista de entrada "l"según la coordenada en x. Puede utilizar diferentes técnicas de ordenamiento, las cuales se pasan como argumento. Las técnicas disponibles son: *up-sort*,*merge-sort* o el algoritmo por default de python. Para las dos primeras técnicas de ordenamiento, se implementaron cuatro funciones auxiliares (las funciones `maxPos(a,b,c)` y `upSort(l)` para ordenar la lista según la técnica de *up-sort* y `merge(l1,l2)` y `mergeSort(l)` para ordenar la lista según la técnica de *merge-sort*). En el segundo bloque, se hace el llamado de la función auxiliar `minimaDistRec(l)`, que calcula el par de puntos más cercanos de manera recursiva y según la técnica de divivir y conquistar. El parámetro de entrada l, debe ser una lista ordenada según la coordenada en x. El ordenamiento se lleva a cabo dentro de `minimaDistanciaDyC(l,algoritmo)` para evitar ordenar una lista que ya

está ordenada durante los pasos recursivos. Además, `minimaDistRec(l)` utiliza la función auxiliar `paresCruzados(l1,l2,x,dist)`. Esta función auxiliar recibe dos pares de listas ordenadas en la coordenada x, que corresponden con los conjuntos de puntos de izquierda y derecha partidos en el paso de dividir.El punto x, mediante el cual se hizo la división y la minima distancia entre los pares de puntos de ambas listas. Está función implemente el paso de combinar, y devuelve posibles pares de coordenadas que podrían encontrarse a una distancia menor a la hallada hasta ese momento (mezcla pares de puntos de los grupos izquierda y derecha)

En la tercera sección del código, se encuentran implementadas las funciones `puntosAleatorios(a,b)` y `experimentarAlgoritmos()`. La primer función genera un set de puntos de cantidad a. Los posibles valores de las coordenadas varían entre -b y b. La segunda función testea a los algoritmo para diferentes sets de datos y devuelve un archivo .txt con los tiempos de ejecución de cada algoritmo.

A continuación se presentan los gráficos de tiempos obtenidos para diferentes cantidades de datos y para diferentes algoritmos empleados. Se tomó como criterio analizar a partir de 1000 puntos, porque para cantidades menores la medición del tiempo está dentro del rango de error del módulo time de python. Las mediciones se emplearon varíando los sets de datos en orden creciente (se probó para 1000, 5000, 10000, 15000, 20000 y 50000 puntos, ver Figura 1).

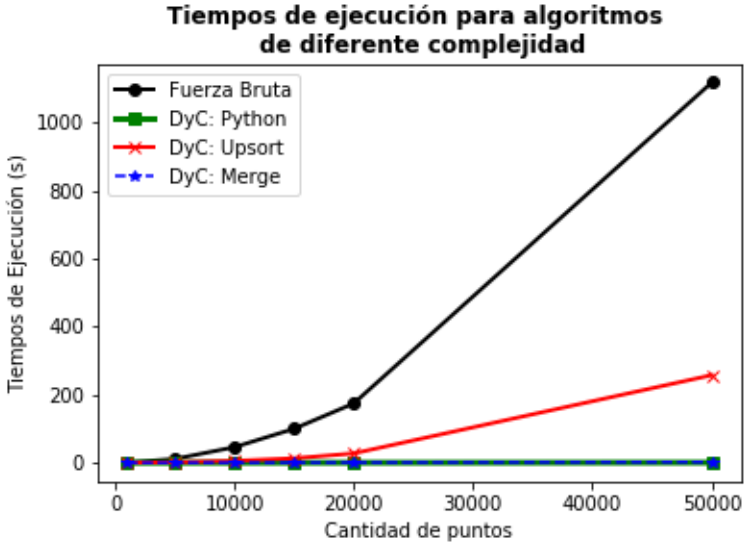

Figura 1: Tiempos de ejecución obtenidos para los diferentes algoritmos

Como se observa en la Figura 1, a medida que crece el número de datos a

calcular, aumenta el tiempo de cómputo para todas las funciones. La función más lenta es distanciaMinima(l), la cual utiliza fuerza bruta como algoritmo. Además, como estaba previsto, puede verse que los tiempos de ejecución aumentan cuadráticamente (fuerza bruta es un algoritmo de O(n^2)). Por otro lado, puede observarse que en los algoritmos que resuelven el problema mediante la técnica de dividir y conquistar, los tiempos de computo disminuyen considerablemente (esta técnica tieme complejidad de O(n.log^2(n)). En particular, si el ordenamiento se lleva a cabo de acuerdo a la técnica de *merge-sort* o mediante el algoritmo por *default* de python, los tiempos de cómputo son más rápido que al utilizar el ordenamiento por *up-sort*.Por un tema de escala, parece que las curvas de *merge-sort* y del algoritmo de python son contaste en tiempo de cómputo, pero ambas sufren un pequeño incremento en tiempo (ver Tabla 1).

Algoritmo	**Tiempos de Ejecución (s)**					
Fuerza Bruta	4.5×10^{-1}	1.1×10^{1}	4.1×10^{1}	1×10^{2}	1.8×10^{2}	2.2×10^{3}
DyC:Python	0×10^{1}	4.7×10^{-2}	1.1×10^{-1}	1.6×10^{-1}	2.0×10^{-1}	4.7×10^{-1}
DyC:Up-Sort	6.2×10^{-2}	1.4×10^{1}	5.5×10^{1}	1.3×10^{1}	2.4×10^{1}	2.0×10^{2}
DyC:Merge-Sort	1.6×10^{-2}	7.8×10^{-2}	1.7×10^{-1}	2.7×10^{-1}	3.6×10^{-1}	1.2

Cuadro 1: Tiempos de ejecución para diferentes algoritmos

2. Ejercicio 2: Python y Numba

Dentro de la carpeta **src_2** ubicada en el repositorio, se encuentran los archivos correspondientes del ejercicio 2. El archivo *ej2.py* contiene la implementación de las funciones, mientras que el archivo *ej2_graf.py* contiene la implementación de los gráficos obtenidos.

El código *ej2.py* esta implementado para correrse por consola. contiene a las funciones `gray_filer(img)` y `blur_filter(img)`. Además, se implementó la función `medirTiempos(fn,*args)`, la cual mide el tiempo de ejecución de los filtros, y la función `getfilenames(path)`, que devuelve una lista con las rutas de los archivos del tipo requerido, dentro del path. Para correr la función por consola, se necesitan 5 argumentos: la ruta de la imagen junto con su extensión, el número de threads que se quieren correr (solo sirve para escribir el .csv de salida, no cambia la variable de entorno), el número de repeticiones y el nombre del archivo de salida.

El archivo de salida es un .csv, separado por ; que sigue la siguiente estructura: tam_imagen; threads; veces que se repitió el proceso; media (tiem-

pos gray); desviacion estandar (tiempos gray); minimo (tiempos gray), maximo(tiempos gray); media (tiempos blur); desviacion estandar (tiempos blur); minimo (tiempos blur), maximo(tiempos blur).

Las figuras 2 y 3 contienen los resultados de escalabilidad para los filtros gray y blur. Cada figura contiene ocho gráficos. Cada gráfico representa el tiempo de ejecución del filtro para una determinada resolución de la imagen utilizada en función de la cantidad de núcleos utilizados (las imágenes se encuentra en el repositorio, en la ruta *src_2/im*). Las pruebas de tiempo se corrieron utilizando la maquinaria de *just in time* y se utilizó una computadora de 12 núcleos. Observando la figura 2, puede notarse que para imágenes de baja resolución, la paralelización resulta ineficiente, siendo preferible utilizar un sólo núcleo. Por otro lado, para el resto de las resoluciones, la paralelización disminuye los tiempos de ejecución, pero hay una levé tendencia a aumentar el tiempos cuando se utilizan más de 8 núcleos. La figura 3, muestra todos los gráficos homogéneos debido a un tema de escala (hay mucha diferencia entre utilizar un sólo núcleo o paralelizar). En la tabla 2 puede consultarse alguno de los tiempos obtenidos para más detalles.

Núcleos / **Resolución**	**Tiempos de Ejecución (s)**				
	1	2	4	8	10
640x480	0.1994	0.00203	0.00208	0.00208	0.00204
1280x1024	0.6155	0.0015	0.0161	0.00165	0.00162
5120x4096	5.2527	0.01150	0.01042	0.01034	0.01058

Cuadro 2: Algunos ejemplos de tiempo de ejecución para el filtro blur

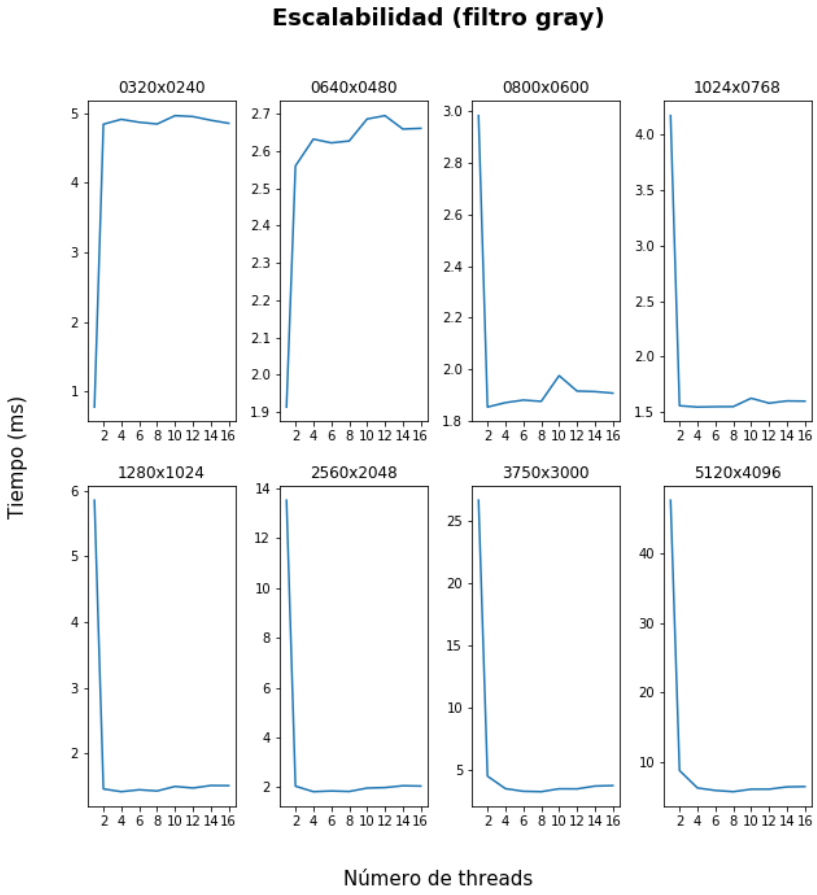

Figura 2: Gráficos de escalabilidad para el filtro gray

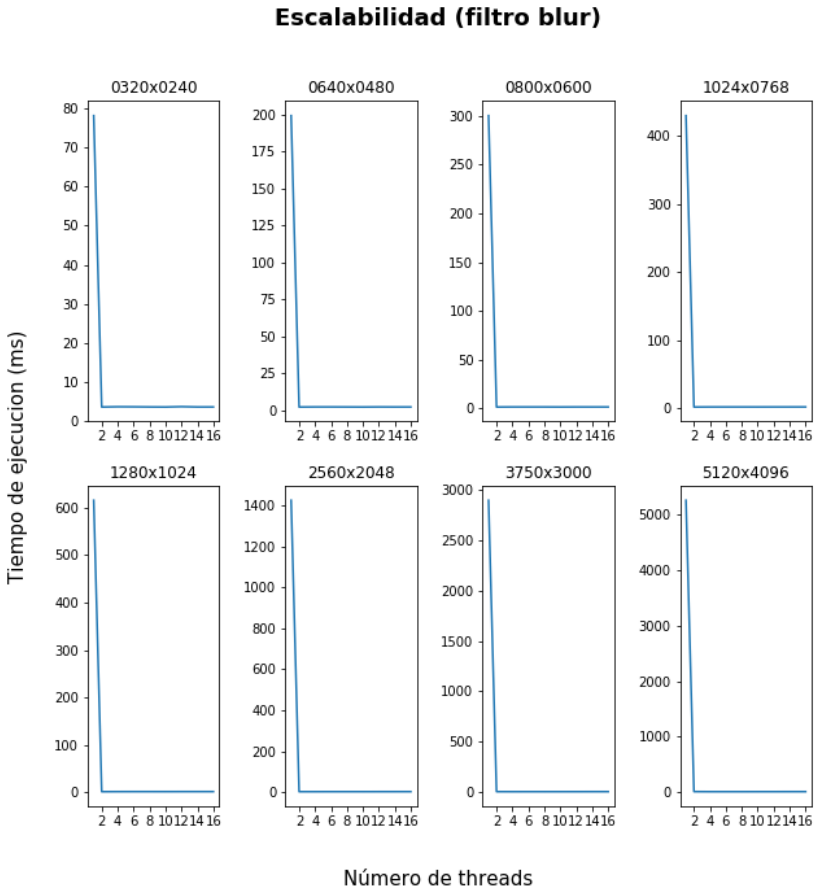

Figura 3: Gráficos de escalabilidad para el filtro blur

3. Ejercicio 3: Backtraking

Para resolver el ejercicio 3 se implementaron las funciones que se pedian en el enunciado y algunas funciones auxiliares ocultas. Dentro de las funciones auxiliares, un grupo se ocupa del tratamiento booleano de listas (`_not_lista(self, l1)`, `_and_listas(self, l1, l2)`, `_todo_False(self, l1)`, `_pos_true(self, l1)`. Un segundo grupo se ocupa de obtener una lista ordenada en las 4 direcciones de una casilla (i, j) de valores booleanos en funcion de si las casillas adyacentes fueron visitadas o son parte del camino (`_getCamino(self, i, j)`, `_getVisita(self, i, j)`. En tercer lugar se implementó la funcion `_mover(self, movimiento, i, j)` que tiene como entradas un numero entre 0 y 3 (movimiento) que indica la direccion en la que debe moverse y una coordenada (i, j), la posicion actual de la rata y ge-

nera un cambio de estado del programa (cambio en valores de atributos). Por ultimo, la funcion auxiliar `_notescapes(self)` resuelve el problema de que algunos laberintos no tienen paredes en los bordes y esto hacia que la rata tomara posiciones fuera del laberinto. Se implementó de esta manera porque nos pareció mas eficiente cerrar las paredes a poner un condicional mas en cada movimiento. Además, entendemos que las salidas del laberinto no cumplen ninguna función en la manera que está implementado el programa.

Con esta implementación el laberinto puede ser resulto en caso de que tenga una solución posible, o no en caso de que no la tenga al correrlo por consola interactiva. Desde la interfaz gráfica, el programa abra y carga el laberinto, pero al tocar el boton de resolver, la rata se mueve un casillero y obtenemos un error de qt que no pudimos solucionar (QObject::startTimer: timers cannot be started from another thread).