Introducción a la Computación / Tópicos de Programación

Trabajo Práctico

Fecha de entrega: Lunes 2 de diciembre de 2019

Ejercicio 1: Divide & Conquer

Se tiene un archivo de texto con un par de números (enteros o de punto flotante) por cada línea, que representan puntos en el plano 2D. Los números pueden ser positivos o negativos. Por ejemplo, son entradas válidas:

```
1.3 4.4          -0.5 5.5          -10 -6.1
-1.0 5.3         5 5.5             21.1 14.4
0.0 2.3          7.5 10.8          60 -50
```

Se necesita, a partir de un archivo en este formato, obtener el par de puntos diferentes cuya distancia euclídea sea mínima. La distancia euclídea entre los pares de puntos (x_0, y_0) y (x_1, y_1) se define de la siguiente manera:

$$\text{distancia}((x_0, y_0), (x_1, y_1)) \equiv \|(x_1 - x_0, y_1 - y_0)\| = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$

Se pide:

1. Implementar una función `listaDePuntos(fn)` que tome como entrada un archivo, cuyo nombre está indicado por `fn`, que contiene una lista de puntos en el formato especificado, y devuelva una lista de *tuplas* con los valores de cada punto del archivo.

   Ayuda 1: una tupla con valores `x` e `y` se define en `Python` como `(x, y)`.

2. Implementar una función `distanciaMinima(l)` que tome como entrada una lista como la definida en el punto anterior y devuelva cuál es el par cuya distancia es mínima. En el caso del ejemplo, el par de distancia mínima es el $(-1{,}0; 5{,}3), (-0{,}5; 5{,}5)$, cuya distancia es 0,5385. La estrategia a utilizar es la de *fuerza bruta*, es decir calcular las distancias entre todos los pares posibles de puntos. Como era de esperar, resulta en una complejidad $O(n^2)$, siendo n la cantidad total de puntos.

3. El algoritmo anterior puede ser mejorado a $O(n \log^2 n)$ si se utiliza la técnica de *divide & conquer*.
   Diseñar e implementar una función `distanciaMinimaDyC(l, algoritmo)` siguiendo esta técnica, teniendo en cuenta el esquema descripto a continuación:

   a) Preprocesamiento: ordenar la lista de puntos por su coordenada x. Implementar *merge sort* y *upsort*, y decidir qué algoritmo utilizar en función de la cadena indicada por el parámetro `algoritmo`:

   - `'merge'`: utiliza *merge sort*.
   - `'up'`: utiliza *upsort*.
   - `'python'`: utiliza el sort de Python. [1]

   b) Dividir la lista de puntos en dos mitades, cortando sobre el eje x.

   c) Buscar recursivamente las distancias menores en ambas mitades, y quedarse con la menor de ambas.

   d) Combinar el resultado anterior con el de entre pares de puntos que se encuentran cada uno en una mitad diferente.
   Ayuda 2: puede utilizar la menor de las distancias encontradas en los pasos recursivos como cota para determinar cuánto debe alejarse hacia cada lado en la coordenada x de la recta que corta ambas mitades, ya que la lista de puntos se encuentra ordenada en x.
   Se puede pensar que las soluciones posibles se categorizan en tres grupos:

   1) los pares de puntos que se encuentran en el lado izquierdo
   2) los pares de puntos que se encuentran en el lado derecho
   3) los pares de puntos tales que van de un punto del lado izquierdo a un punto del lado derecho.

[1] Python tiene un metodo *sort* para listas. Por ejemplo una lista l se puede ordenar usando *l.sort()*.

El paso *combine* involucra encontrar potenciales soluciones que se encuentren en la categoría 3. La restricción de distancia se da por una propiedad matemática de las distancias euclídeas, ya que es imposible encontrar un par de puntos en esa categoría que se encuentre más cerca que los pares de puntos en las categorías 1 y 2 si alguno de los dos puntos que constituyen el par se encuentra más alejado de la mitad que la menor de las distancias mínimas encontradas recursivamente.

4. Generar un gráfico en el cual se analice el tiempo de ejecución variando el tamaño del conjunto de puntos de entrada, midiendo los dos algoritmos implementados, incluyendo una curva por cada algoritmo de ordenamiento en el caso del algoritmo de *divide & conquer*. Sacar conclusiones con respecto a la performance del algoritmo de *divide & conquer* con los distintos algoritmos de ordenamiento utilizados.

   Ayuda 3: Para tomar tiempos se puede usar `time.time()` (del módulo `time`) y para generar puntos de entrada se puede usar `random.random()` (del módulo `random`) que genera números aleatorios entre 0 y 1 (ajustar al rango de valores que necesite por medio de operaciones aritméticas).

Ejercicio 2: Python & Numba

Una imagen digital en mapa de bits o imagen ráster es un archivo compuesto esencialmente por una matriz bidimensional donde cada elemento es un punto de color llamado píxel. Cada píxel está codificado por uno o más valores numéricos. En las imágenes en escala de grises cada píxel está representado mediante un byte, por tanto cada punto puede albergar una de 256 posibles intensidades de gris. En las imágenes en color RGB, cada píxel se compone de tres valores numéricos, cada uno de un byte y representando la intensidad de un color base: rojo (Red), verde (Green) y azul (Blue). Las imágenes se despliegan en el monitor de la computadora también a través de píxeles, si amplificamos un sector de nuestra pantalla (podemos hacerlo mediante una lupa) se puede apreciar cada píxel como la composición de los tres colores nombrados.

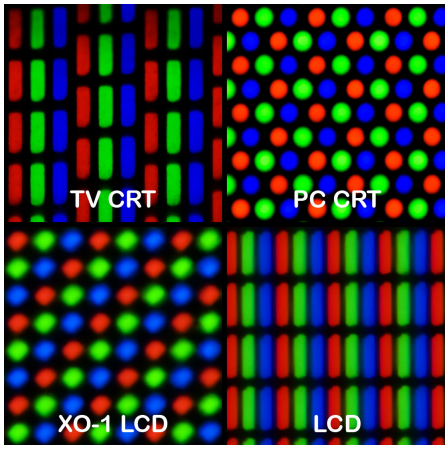

Figura 1: Visualización de la composición de los colores en diferentes tipos de pantallas (fuente: `https://en.wikipedia.org/wiki/Pixel_geometry`)

Para almacenar imágenes digitales en disco se utilizan distintos formatos gráficos, de los cuales el más conocido es el JPEG. Cada formato tiene diferentes características que indican cómo y dónde se debe guardar cada píxel de la imagen en un archivo. Estos formatos son complicados de interpretar, pero al ser problemas muy comunes existen bibliotecas en cada lenguaje para solucionarlos.

El procesamiento de imágenes digitales es un problema computacionalmente costoso para el que implementar su solución completamente en `Python` puede resultar ineficiente. Para no perder tanta eficiencia, pero de todas formas aprovechar las características que tiene `Python`, el lenguaje ofrece mecanismos alternativos como `Numpy`.

En nuestro problema trabajaremos leyendo las imágenes a través de las bibliotecas `PIL` (Python Image Library) que se incluyen en `SciPy` y resolveremos la implementación de dos filtros gráficos en `Python` *puro*, con `Numpy` y `Numba`. La matriz bidimensional que representa a la imagen estará implementada sobre matrices de `NumPy` en el caso de `Python`. En el caso de las imágenes RGB, cada elemento de la lista contiene los tres valores numéricos correspondientes.

1. Implementar una función en `Python` `gray_filter(img)` que tome una imagen en colores `img` y devuelva una nueva imagen con la misma cantidad de filas y columnas pero con una sola componente de color, con el resultado de aplicar la siguiente transformación al espacio de color YUV (quedándose únicamente con la componente de luma Y):

$$res_{i,j} = 0{,}3 \cdot img_{i,j}.\mathbf{r} + 0{,}6 \cdot img_{i,j}.\mathbf{g} + 0{,}11 \cdot img_{i,j}.\mathbf{b}$$

   Ayuda: Para acceder a cada componente de color **r**, **g** o **b**, utilizar el índice de la tercera dimensión de la matriz `img` ($0 = \mathbf{r}$, $1 = \mathbf{g}$, $2 = \mathbf{b}$). Por ejemplo, para acceder al valor de la componente verde del pixel $(2, 2)$, utilizar `img[2,2,1]`.

2. Implementar una función en `Python` `blur_filter(img)` que tome una imagen en escala de grises `img` y devuelva una nueva imagen del mismo tamaño con el resultado de aplicar un filtro de difuminado sobre la imagen original. Dicho filtro se corresponde a aplicar la siguiente matriz de convolución de 3×3 a cada elemento de `img`:

$$\begin{pmatrix} 0 & 0{,}25 & 0 \\ 0{,}25 & 0 & 0{,}25 \\ 0 & 0{,}25 & 0 \end{pmatrix}$$

   El cálculo anterior equivale a calcular la siguiente expresión para cada elemento i, j tal que $0 < i < filas(img) - 1 \wedge 0 < j < columnas(img) - 1$:

$$res_{i,j} = \frac{img_{i-1,j} + img_{i+1,j} + img_{i,j-1} + img_{i,j+1}}{4}$$

Por simplicidad, los elementos no alcanzados por la condición anterior deben estar en negro (valor 0).

3. Agregar las directivas de `Numba` para que las dos funciones anteriores utilicen la maquinaria de *just-in-time-compiling* (`jit`) que provee Numba.

4. Modificar el punto anterior para que `Numba` utilice paralelismo (`Parallel=True`). Probar las funciones anteriores y verificar con `htop` que efectivamente se están utilizando todos los núcleos disponibles en la máquina. Por medio de la variable de ambiente `NUMBA_THREADING_LAYER` debería ser posible bajar la cantidad de núcleos en uso, probar poner un valor menor a la cantidad de núcleos disponibles y verificar con `htop` que ocurra.

5. Realizar el siguiente experimento desde el código en `Python`:

   a) Elegir una imagen y cargarla[2].

   b) Medir el tiempo que demora convertir la imagen cargada a escala de grises, y luego aplicar el filtro de difuminado, utilizando la implementación en `Python`.

   c) Medir el tiempo que demora realizar la misma operación indicada por el ítem anterior, pero invocando a su implementación en `Numba`.

   d) Medir el tiempo que demora realizar la misma operación indicada por el ítem anterior, pero invocando a su implementación en `Numba` en paralelo modificando la variable de ambiente `NUMBA_THREADING_LAYER` con 1, 2, 4 y, de poder acceder a máquinas con mayor cantidad de núcleos, 8, 16, 32, 64.

   e) Almacenar en disco ambas imágenes difuminadas[3]. Verificar que el resultado obtenido coincida.

6. Comparar los resultados obtenidos en el experimento anterior en ambas implementaciones, repitiéndolo para imágenes de distintos tamaños.

7. Muchas veces resulta importante analizar cómo se comporta una aplicación paralela a medida que se incorpora un mayor número de recursos para solucionar el mismo problema. Esto se denomina **escalabilidad** y se presenta graficando el tiempo que se demora en resolver un problema dado en función de la cantidad de recursos computacionales utilizados.
   Presentar los resultados de `Numba` paralelo graficando el tiempo que demora con un núcleo, luego con dos y así para la cantidad disponible de núcleos a los que tenga acceso. Normalmente es posible notar cómo la mejora que se consigue al agregar mayor número de recursos, va disminuyendo. Realizar el gráfico y evaluar el comportamiento usando imágenes de distintos tamaños.

[2] Utilizar `matriz = misc.imread(nombre)`.
[3] Utilizar `misc.imsave(nombre, matriz)`.

Ejercicio 3: Backtracking

Se tiene un laberinto de tamaño $N \times M$ (N filas por M columnas) y se desea simular el comportamiento que tendría una rata al intentar recorrerlo, hasta alcanzar un trozo de queso. Originalmente existía un programa completo que lo modelaba, con una interfaz gráfica que nos permitía visualizar la simulación, pero se perdió la implementación del `TAD Laberinto` en una actualización fallida de Ubuntu, por lo que es necesario reconstruirlo.

Después de arduas horas de analizar el código que sobrevivió, se logró reconstruir el esqueleto del TAD, pero sigue faltando el código que permite que todo funcione. El objetivo de este punto será encargarse de esta tarea.

Para que todo vuelva a funcionar, se pide completar los siguientes métodos de la clase Laberinto (en el archivo `laberinto.py`) para que cumplan con la funcionalidad requerida en cada caso.

Utilizaremos como base el código que se encuentra en el directorio `src_ej3` de los archivos adicionales del TP (ver apéndice A).

1. `cargar(self, fn)`: abre, lee, procesa el archivo de texto cuyo nombre se indica en la variable `fn`, cargando el laberinto allí descripto. El archivo debe respetar el formato indicado en el apéndice B. Al finalizar la carga, el laberinto debe encontrarse reseteado (ver método `resetear(self)`), y debe mover la rata a la posición $(0, 0)$ (borde superior izquierdo) y el queso a la posición $(N - 1, M - 1)$ (borde inferior derecho).

2. `tamano(self)`: devuelve una tupla de dos elementos, donde el primero representa la cantidad de filas y el segundo es la cantidad de columnas del laberinto.

3. `resetear(self)`: limpia el laberinto, desmarcando las posiciones que aparecen como visitadas o como parte de un camino.

4. `getPosicionRata(self)`: devuelve una tupla de dos elementos, donde el primero indica la fila y el segundo la columna donde se encuentra la rata.

5. `getPosicionQueso(self)`: devuelve una tupla de dos elementos, donde el primero indica la fila y el segundo la columna donde se encuentra el queso.

6. `setPosicionRata(self, i, j)`: cambia la posición de la rata a la fila i, columna j. Devuelve `True` si el cambio fue posible (es decir, si la posición indicada se encontraba dentro de los límites del laberinto), `False` en otro caso.

7. `setPosicionQueso(self, i, j)`: cambia la posición del queso a la fila i, columna j. Devuelve `True` si el cambio fue posible (es decir, si la posición indicada se encontraba dentro de los límites del laberinto), `False` en otro caso.

8. `esPosicionRata(self, i, j)`: devuelve `True` si el par ordenado determinado por (i, j) corresponde a la posición de la rata, `False` en caso contrario.

9. `esPosicionQueso(self, i, j)`: devuelve `True` si el par ordenado determinado por (i, j) corresponde a la posición del queso, `False` en caso contrario.

10. `get(self, i, j)`: devuelve una lista con 4 elementos *booleanos*, que indican si hay pared o no en cada uno de los 4 bordes de la celda ubicada en la fila i, columna j. El orden en el que deben aparecer los bordes es el siguiente: *Izquierda*, *Arriba*, *Derecha*, *Abajo*. Por ejemplo, si la celda pedida tiene paredes a su derecha y arriba, este método debe devolver `[False, True, True, False]`.

11. `getInfoCelda(self, i, j)`: devuelve un diccionario con información acerca de la celda ubicada en la posición (i, j). El diccionario debe tener dos claves: `visitada` y `caminoActual`. La clave `visitada` es un *booleano* que indica si la celda fue visitada por el backtracking, mientras que `caminoActual` indica si es parte del camino actual que está siendo probado por el backtracking. Por ejemplo, si la celda ya fue visitada por el backtracking y no es parte del camino actual, debe devolver `{'visitada': True, 'caminoActual': False}`.

12. `resuelto(self)`: devuelve `True` si el laberinto está resuelto, es decir, si la posición de la rata es la misma que la del queso, y `False` en otro caso.

13. `resolver(self)`: resuelve el laberinto utilizando *backtracking*, comenzando por la posición en la que se encuentra la rata. Durante el proceso de resolución debe actualizar el estado de cada celda recorrida, de forma que quede marcado el camino actual, y en caso de volver hacia atrás por quedarse sin movimientos posibles, las celdas por las que vuelve deben ser marcadas como visitadas (y desmarcadas como parte del camino actual). Devuelve `True` si llega a la posición del queso, `False` si esto no es posible (si no existe un camino).

    Importante: Para ver actualizaciones en pantalla durante el backtracking, cada vez que cambie la posición de la rata o el estado de una celda es necesario llamar al método `self._redibujar()`.

Algunas notas adicionales que pueden ser útiles:

- Se recomienda programar los métodos pedidos en el orden en el que aparecen en el enunciado, de forma de poder ir probando de manera parcial la funcionalidad implementada.
- Para hacer pruebas se puede utilizar el modo interactivo del intérprete sobre el código de la clase laberinto (ejecutando `python3 -i laberinto.py`).
- Como guía para elegir la forma de estructurar los datos dentro del TAD, pensar en términos de hacer lo más sencillo posible devolver lo pedido en cada método.
- Ustedes deben elegir las variables para almacenar el estado del laberinto, que deberán ser marcadas como *privadas*, de manera de no utilizarse fuera de la clase.
- Pueden agregar métodos privados, pero no modificar la interface pública de la clase.

A. Cómo utilizar la interfaz gráfica del Ej. 3

La interfaz gráfica provista en la carpeta `src_ej3` del ZIP de archivos adicionales nos permite cargar, mostrar en pantalla y resolver laberintos, así como también visualizar el mecanismo de backtracking utilizado en la resolución. Para ejecutarla, en una terminal posicionada adecuadamente, ejecutar el siguiente comando: `python3 tplaberinto.py` (o bien `./tplaberinto.py`).

Esto inicializará el programa, mostrando una ventana con un recuadro negro en el lado izquierdo y varios botones y controles en el lado derecho. En el lado izquierdo se visualizará el laberinto una vez cargado.

Los botones tienen la siguiente funcionalidad:

- **Cargar**: abre un cuadro de diálogo que permite elegir un archivo con un laberinto a cargar.
- **Resolver**: resuelve el laberinto, llamando al método `resolver` de la clase Laberinto.
- **Resetear**: resetea el laberinto, llamando al método `resetear` de la clase Laberinto.
- **Salir**: sale del programa.

Los spinboxes *Posición inicial* y *Posición final* modifican la posición de la rata y del queso respectivamente. Si alguno de los dos botones **Modificar** se encuentra presionado, se activan los spinboxes correspondientes y además es posible mover la rata o el queso por el laberinto utilizando las flechas del teclado.

Finalmente, el slider *Velocidad de la animación* permite variar la velocidad con la que se redibuja el laberinto cada paso del backtracking. A efectos prácticos, se ve afectada la espera entre cada llamado a `self._redibujar()` y el siguiente paso.

B. Formato del laberinto

Un laberinto se representa por medio de un archivo de texto que tiene el siguiente formato:

1. La primer línea indica el tamaño del laberinto con la sintaxis:

   `Dim(`N`,`M`)`, donde N y M son números que indican la cantidad de filas y la cantidad de columnas respectivamente. Por ejemplo: `Dim(10,20)` indica un laberinto de 10 filas por 20 columnas.

2. Cada una de las líneas restantes indica una fila, en orden.

3. El formato de cada fila es el siguiente:

$$\overbrace{[0_{\texttt{left}}, 0_{\texttt{up}}, 0_{\texttt{right}}, 0_{\texttt{down}}] \cdots\cdots [\texttt{M} - 1_{\texttt{left}}, \texttt{M} - 1_{\texttt{up}}, \texttt{M} - 1_{\texttt{right}}, \texttt{M} - 1_{\texttt{down}}]}^{\text{M veces}}$$

   Cada uno de los bloques entre corchetes indica la descripción de la celda ubicada en la columna i-ésima de la fila a la que corresponde esa línea del laberinto. A su vez, cada uno de esos bloques tiene 4 elementos separados por comas, sin espacios, que pueden tener dos valores posibles, 1 o 0, según la presencia o ausencia de una pared en cada dirección (izquierda, arriba, derecha y abajo, que aparecen en ese orden). Por ejemplo, si una celda tiene paredes arriba y a la derecha, su descripción en el archivo es la siguiente: `[0,1,1,0]`.

Junto al código fuente de la GUI van encontrar varios ejemplos en este formato, de extensión `*.lab`, listos para probar con su implementación.

Condiciones de entrega:

- El trabajo se deberá realizar en grupos de dos (2) personas.
- Entregar una versión impresa **y** un archivo comprimido ZIP que contenga los archivos fuentes correspondientes a cada ejercicio y un informe (en formato PDF).
- La versión impresa debe incluir tanto el código como el informe (no incluir en la impresión el código ya provisto, como la interfaz gráfica del laberinto).
- Se evaluará la correctitud del código producido, su claridad y legibilidad; y el uso de la herramienta `git`. Esto es, no debe subirse sólo la versión final, sino usar el repositorio para subir versiones intermedias, con descripciones de `commit` significativas. De no hacerlo así, el trabajo quedará desaprobado. En el repositorio también se deberá incluir el archivo fuente del informe.

- El informe se realizará en LaTeXy deberá incluir:
  - **Ej. 1**: decisiones de diseño y cómo correr las pruebas necesarias para generar el gráfico incluido en el informe.
  - **Ej. 2**: resultados de los experimentos, qué pruebas realizaron y cómo se corren. La evaluación de los métodos usando las distintas opciones de Numba y, al considerar paralelismo, graficar tiempo de cómputo en función de la cantidad de núcleos utilizados (1, 2, 3 y 4).
  - **Ej. 3**: decisiones de diseño (atributos que hayan definido en la clase `Laberinto`, cómo se guardan los datos que cargan, estrategia del backtracking, etc.)
  - Cualquier otra aclaración que consideren pertinente.
- Los archivos fuentes deberán tener comentarios necesarios, aunque sin dificultar la lectura del código.
- Los gráficos deberán realizarse con Matplotlib.
- Una vez terminado, deben enviar un mail de aviso a la dirección de correo electrónico de los docentes de la materia: **icb-doc@dc.uba.ar**. Deberán poner como *subject*:
  [Grupo ⟨NN⟩: ⟨Nombre grupo⟩] TP - ⟨Apellido1 LU1⟩ - ⟨Apellido2 LU2⟩. Por ejemplo, un grupo podría ser:
  [Grupo 22: "Les salieris de Turing"] TP - Gómez 334/99 - Coria 671/01
- El cuerpo del mail de aviso debe indicar el comando que tiene que ejecutar el equipo docente para clonar el repositorio donde se encuentra el TP.
- **Nota**: se puede tomar coloquio individual a criterio de los docentes sobre cualquier tema del TP.
- **Importante**: Sólo se admite la entrega por medio de `Gitlab`. Se deberá crear un proyecto nuevo, en el cual deberán asegurarse de agregar al equipo docente con permisos de lectura (rol *Reporter*), según los usuarios en el Campus. Se descargará la última versión de los archivos directamente de allí.

  No se aceptarán las entregas sólo por mail o sólo en forma impresa.