NOMBRE: Diego Fluxá

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructura de Datos y Algoritmos — 2' 2020

Informe Tarea 0

1 Complejidades de Eventos

i. GROW:
Para el comando GROW (líneas 71-94 de main.c), podemos observar que en la línea 74 se define una variable (cur) que consideraremos que corresponde a un tiempo constante y, por lo tanto, será $\mathcal{O}(1)$. Luego, en las líneas 77 y 90 se lee una línea del archivo, lo que también consideraremos constante y por ende de complejidad $\mathcal{O}(1)$.

Ahora, en la línea 80 nos encontramos con un for, el cual recorre nuestro mould hasta llegar a la célula que crecerá. Entonces, si tenemos un mould de n células y nos ponemos en el peor de los casos, el cual sería tener un mould completamente lineal y que tuvieramos que recorrer todos las células para hacer crecer la última, este for se ejecutaría n veces, por lo que posee complejidad $\mathcal{O}(n)$. Es importante mencionar que dentro de este for solo se lee una linea del archivo y se define una variable, lo que ya mencionamos que tiene complejidad $\mathcal{O}(1)$.

Por último, en la línea 93 se ejecuta la función *mould_add_cell*, por lo que tendremos que pasar a analizar la complejidad de esta función. $Mould\_add\_cell$, se encuentra definida en las 45-53 de mould.c y podemos observar que posee sólo 2 acciones, sumar 1 al id del mould que será $\mathcal{O}(1)$ y la creación de una nueva célula al utilizar la función *cell_init*, la cual no sabemos su complejidad por lo que tendremos que analizarla rápidamente. Entre las línas 6-13 de cell.c podemos encontrar la definición de la función *cell_init*, donde nos podemos dar cuenta con facilidad que ocupará un tiempo constante, razón por la cual su complejidad será $\mathcal{O}(1)$. Esto último se debe a que esta función solo solicita espacio de memoria y define a una variable célula y sus atributos para posteriormente retornar el puntero a dicha célula, y todos estas acciones tomarán tiempos constantes.

Finalmente, la complejidad del evento GROW corresponderá a:

$$\text{Eventos constantes} + for$$

$$\mathcal{O}(1) + \mathcal{O}(n)$$

Aquí, como sabemos, se ignoran las constantes y se mantiene el mayor.

Entonces, habremos demostrado que la complejidad del evento GROW corresponde a $\mathcal{O}(n)$.

ii. BUD:

Para el comando BUD (líneas 197-221 de main.c) podemos notar que es muy similar en estructura a GROW con la excepción de que en vez de utilizar la función *mould_add_cell* utiliza la función *cell_delete*.

Entonces, al igual que GROW todos los eventos hasta antes de *cell_delete* serán constantes a excpeción del for utilizado para llegar a la célula de interés, el cual ya hemos demostrado que posee complejidad $\mathcal{O}(n)$.

Ahora, procederemos a analizar la función *cell_delete* definida en las líneas 48-63 de cell.c. Esta función, tiene por objetivo eliminar del mould a una célula específica la cual se obtiene mediante su célula madre y el índice de la célula a eliminar, y también se libera su espacio en memoria. Para esto, la función también debe eliminar y liberar el espacio de todos los hijos de la célula objetivo, razón por la cual se implementa un for que recorre todos los indices de hijos de la célula y en caso de existir un hijo se le aplica la misma función *cell_delete* a esa célula hija (función recursiva).

En el peor de los casos, la función *cell_delete* tendrá que recorrer para toda célula del mould a excpeción de la célula raiz (por la definición de esta función no se puede aplicar a la raiz porque no posee célula madre) cada uno de sus posibles hijos (las células poseen 10 espacios para hijos). Por lo tanto, al recorrerse para las $n-1$ células (se excluye la raíz) cada uno de sus espacios para hijos, la complejidad de esta función será $\mathcal{O}(10 * (n-1)) = \mathcal{O}(10n - 10)$ lo que sabemos que, para efectos del cálculo de complejidad, es equivalente a $\mathcal{O}(n)$.

Finalmente, tendremos que la complejidad de nuestro evento BUD estará dado por la suma de las complejidades de los eventos constantes, nuestro for para llegar a la célula objetivo y la función *cell_delete*. Es decir:

$$\mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(10 * (n-1))$$

Se desprecian las constantes y se toma $\mathcal{O}(10 * (n-1) = \mathcal{O}(n)$ y tenemos,

$$\mathcal{O}(n) + \mathcal{O}(n)$$

$$2 * \mathcal{O}(n)$$

$$\mathcal{O}(n)$$

Por lo tanto, hemos demostrado que la complejidad del algoritmo BUD corresponde a $\mathcal{O}(n)$.

iii. CLONE:

Para el comando CLONE (líneas 97-138 de main.c), se puede observar que, a parte de los eventos constantes, realiza el mismo for presente en GROW y BUD pero realiza 2 distintos para encontrar la célula a clonar y donde se clonará. Además, termina utilizando la función *mould_copy_cell* definida en las líneas 17-37 de main.c.

En primer lugar, como ya hemos visto, cada for posee una complejidad de $\mathcal{O}(n)$. Es importante mencionar que esto sigue siendo factible porque podría querer copiarse la última célula de un mould líneal como hijo de esa misma célula por lo que se estarían recorriendo todas las células ambas veces.

Luego, debemos pasar a analizar la complejidad de *mould_copy_cell*. Esta función, en caso de existir la célula a copiar, retorna el puntero a una nueva copia de dicha célula. Para crear est nueva célula utiliza recursión para recorrer todas las hijas de la célula original y también copiarlas (y recursivamente con las hijas de esas). Por esto, la complejidad de esta función en el peor de los casos corresponderá a copiar

la célula raiz, por lo que se tendrán que recorrer todas las n células para realizar la copia correctamente.

Finalmente, la complejidad del evento CLONE estará dado por la suma entre la complejidad de los eventos constantes, de ambos for y de la función *mould_copy_cell*. Es decir:

$$\mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n)$$

$$\mathcal{O}(1) + 3 * \mathcal{O}(n)$$

$$\mathcal{O}(n)$$

Por lo tanto, hemos demostrado que la complejidad del algoritmo CLONE corresponde a $\mathcal{O}(n)$.

iv. CROSSOVER:

En el comando CROSSOVER (líneas 141-194 de main.c), se tienen 2 for similares a los analizados anteriormente para encontrar las células destino los cuales vimos que tenían complejidad $\mathcal{O}(n)$. Además de esto posee únicamente acciones para leer líneas de archivos, definir variables y alterar atributos de células, eventos que ya mostramos que son constantes y, por ende, poseen complejidad $\mathcal{O}(1)$.

Luego, la complejidad del evento CROSSOVER estará dada por la suma entre la complejidad de los eventos constantes y de ambos for. Es decir:

$$\mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(n)$$

$$2 * \mathcal{O}(n)$$

$$\mathcal{O}(n)$$

Podemos darnos cuenta que esto es coherente debido a que, al estar intercambiando de lugar 2 células (junto con sus hijas), no es necesario eliminar células ni recorrer el mould por alguna otra razón además de encontrar las células a intercambiar.

Finalmente, hemos demostrado que la complejidad del evento CROSSOVER corresponde a $\mathcal{O}(n)$.

v. ABSORB:

En el comando ABSORB (líneas 224-278) existen múltiples eventos constantes, un for para obtener la célula madre, otro for para obtener la célula descendiente de esta que pasará a conectarse con la primera, y una función de *cell_delete* para eliminar las células que fueron absorbidas y liberar su memoria. Entonces, la complejidad de este evento estará dada por la suma de la complejidad de cada uno de los sub-eventos mencionados anteriormente. Es decir:

$$\mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n)$$

$$\mathcal{O}(1) + 3 * \mathcal{O}(n)$$

$$\mathcal{O}(n)$$

Por lo tanto, hemos demostrado que la complejidad del algoritmo ABSORB corresponde a $\mathcal{O}(n)$.

Es importante mencionar que la complejidad de los for no será exactamente $\mathcal{O}(n)$ debido a que el recorrido del segundo for ocurre desde la célula madre, razón por la cual la complejidad de uno estará condicionado por el otro, entonces en el peor de los casos, en conjunto su complejidad será $\mathcal{O}(n)$. Sin embargo, sabemos que al ser un polinomio de grado 1 se puede considerar simplemente $\mathcal{O}(n)$ para cada

uno.

Además, la demostración de la complejidad de la función *cell_delete* la demostramos anteriormente.

vi. OBSERVE:

En el comando OBSERVE (líneas 224-278), además de múltiples eventos constantes, existe un for para encontrar la célula objetivo (al igual que en el resto de los comandos) y finalmente se llama a la función *cell_observe*, la cual no hemos analizado anteriormente por lo que tenemos que hacer esto para encontrar la complejidad de OBSERVE.

Entre las líneas 15 y 45 de cell.c encontramos definida la función *cell_observe*, la cual tiene por objetivo crear un archivo .txt que represente el estado del mould a partir de una célula entregada. Para esto, se utiliza un for para escribir el espaciado que corresponderá a tres espacios por nivel en el que se encuentra la célula. Entonces, en el peor caso tendríamos un mould lineal donde existirían el mismo número de niveles que de células, es decir, n. Por lo tanto, este for se ejecutará como máximo n veces y tendrá complejidad $\mathcal{O}(n)$.

Luego existe un if y otras acciones de escritura en el archivo y de definición de variables que toman un tiempo constante y entonces tendrán complejidad $\mathcal{O}(1)$.

Posteriormente, existe otro for donde para cada hija de la célula objetivo se le aplica el comando OBSERVE (función recursiva). Entonces, para cada célula desccendiente de la célula inicial se ejecutará nuevamente la función *cell_observe*, es decir, en el peor caso se ejecutará este for n veces. Pero, vimos que esta función posee otro for de complejidad $\mathcal{O}(n)$ para imprimir el espaciado, el cual se ejecutará cada vez que se ejecute el segundo for. Por lo tanto, se ejecutará n veces el primer for.

Por consecuencia, la complejidad de la función *cell_observe* corresponderá a:

$$\mathcal{O}(1) + n * \mathcal{O}(n) + \mathcal{O}(n)$$

$$\mathcal{O}(1) + \mathcal{O}(n^2) + \mathcal{O}(n)$$

$$\mathcal{O}(n^2)$$

Finalmente, la complejidad del evento OBSERVE corresponderá a la suma de la complejidad de los sub-eventos constantes, del for y de la función *cell_observe*. Es decir:

$$\mathcal{O}(1) + \mathcal{O}(n^2) + \mathcal{O}(n)$$

$$\mathcal{O}(n^2)$$

Por lo tanto, hemos demostrado que la complejidad del comando OBSERVE corresponde a $\mathcal{O}(n^2)$.

2 Comparación con Python

A continuación, se procederá a comparar los tiempos de ejecución (tiempo real) entre el programa escrito en el lenguaje C y el escrito en Python. Para esto, se utilizarán los tests los cuales son: el test base *test.txt* entregado en el repositorio, el test *easy_mixed.txt* para representar todos los test easy, el test *medium_test_4.txt* como representante de los medium y el test *hard_test_4.txt* para los hard. Los últimos 3 tests están disponibles en SIDING.

Test	Python	C
test.txt	0m0.748s	0m0.035s
easy_mixed.txt	0m0.455s	0m0.008s
medium_test_4.txt	0m5.515s	0m1.006s
hard_test_4.txt	0m17.861s	0m3.004s

De aquí, podemos observar que para todos los tests el programa C es notoriamente más eficaz en términos de velocidad que el programa escrito en Python. Demorandose aproximadamente 21 veces menos en *test.txt*, 57 veces menos para *easy_mixed.txt*, 5,5 veces menos en *medium_test_4.txt* y 6 veces menos en *hard_test_4.txt*. Estas diferencias de rendimiento entre el programa en C y el programa en Python se deben principalmente a las características de estos lenguaje.
A continuación se pasará a nombrar algunas de estas diferencias que provocan que C tenga un mejor rendimiento que Python:

- **C es un lenguaje compilado** mientras que Python es interpretado. Esto significa que C posee un compilador que procesa el código antes de ser ejecutado y lo optimiza de ser posible (no siempre lo hace para algoritmos complejos). Por otro lado, Python debe interpretar el código cada vez que se ejecuta el programa y este no lo optimiza.

- **C es un lenguaje tipado** mientras que Python no lo es. Por esto, C no debe utilizar tiempo en determinar de que tipo es un dato mientras que Python si debe hacerlo.

- **Diferencias en Estructuras de Datos.** C se diferencia de Python en que no posee estructuras de datos por defecto, si no que uno debe realizar las acciones necesarias para implementarlas. Esto último implica tener que realizar un manejo de memoria que puede ser más complejo que realizarlo en Python pero sin dudas es más eficiente por el hecho de que uno tiene conciencia de que es lo que necesita para esa situación en específico y crea una estructura de datos adecuada para esto. Por otro lado, Python nos entrega estructuras de datos prefabricadas que son más amigables de utilizar, sin tener que definirlas ni manejar el espacio en memoria, pero implica una menor eficiencia porque estas estan fabricadas para adaptarse a múltiples situaciones. Esto último implica que se deban realizar múltiples acciones para asegurarse de esto que dañan a la eficiencia del programa. Por ejemplo, en Python para realizar listas se busca que estas sean de largo indefinido por lo que "por detras" se solicita una gran cantidad de memoria del HEAP y si esta se supera se debe pedir mas espacio en el HEAP y realizar un copiado de la lista, además de normalmente utilizar múltiples equivalentes a los arreglos de C para crear una sola lista. Esta y otras funcionalidades extra que nos entrega Python hace que tenga un peor rendimiento en términos de velocidad con respecto a C, donde las herramientas que se nos entregan son más básicas pero por esto mismo más eficientes pues solo harán lo que necesitemos.

Estas son algunas de las diferencias entre C y Python que provocan que el programa escrito en C tenga un mejor rendimiento. En síntesis, se debe a las características propias de los lenguajes y las herramientas y funcionalidades que entregan estos por default. Aquí Python busca ser más amigable y adaptable por lo que entrega más funcionalidades de avanzada complejidad, mientras que C se destaca por ser eficiente, por lo que entrega herramientas más básicas con las que uno puede realizar lo que busca con algo más de trabajo a cambio de gran eficiencia.