UNIVERSIDAD VERACRUZANA

Facultad de Ingeniería Eléctrica y Electrónica

Periodo:

Agosto - Enero 2026

DISEÑO Y ANÁLISIS DE ALGORITMOS

Estudiantes:

García Martínez Yahaira

Gueixpal Ayala Juan David

Matrícula:

s23002505

s23002503

Académico:

Fleitas Toranzo Yadira

Proyecto

05/12/2025 Veracruz, Boca del Rio.

índice

Inicialización y análisis de datos

Heap Sort

El algoritmo heap sort también conocido como ordenamiento por montículo es un algoritmo que se basa en comparaciones, donde el elemento mayor queda siempre en la raíz y después movemos repetidamente el elemento mayor al final.

Como lo utilizamos en nuestro proyecto:

1.- Construimos un montículo: Insertamos todos los elementos de la población uno por uno mediante la función 'hPush' que tenemos en nuestra librería heap, la estructura interna del heap utiliza un arreglo dinámico de punteros tipo void* y organizamos los datos de forma que el nodo padre tiene mayor prioridad que sus hijos gracias a la función 'criteria'.

2.- Extracción: Una vez ya tengamos el montículo hecho, extraemos la raíz esta como ya sabemos tiene el elemento prioritario utilizando 'hPop'. Al extraer la raíz, el montículo se reorganiza automáticamente para traer el siguiente valor prioritario a la cima, estos elementos extraídos se almacenan en un nuevo arreglo 'sorteArr'.

Justificación

Seleccionamos "HeapSort" como uno de los algoritmos de ordenamiento O (n log n) ya que nos garantiza que tiene un tiempo de ejecución de O (n log n) en el peor, mejor y caso promedio en comparación de otros algoritmos como el de Quicksort que puede tener degradaciones según su tipo de caso. Nuestra implementación de Heap Sort es iterativa ya que usa ciclos while y for para flotar y hundir nodos evitando el riesgo de desbordamiento de pila, en algoritmos como Mergesort que son recursivos podrían ocurrir Stack Overflow al procesar cantidades muy grandes de población, además, gracias a el uso de punteros void* y la función 'criteria' este algoritmo nos ayuda a ordenar por riesgo, nombre o ID sin reescribir código haciéndolo modular.

Cálculo de eficiencia y complejidad

```
PERSON **heapSort(PERSON **population, int n)
{
    // Heapsort es sencillo

    // 1.- Creo mi heap
    HEAP *h = initHeap(n, compare);
    if (!h) return NULL;

    // 2.- Inserto los datos en el heap segun mi criterio
    for (int i = 0; i < n; i++)
    {
        if (population[i] != NULL)
            hPush(h, population[i]);
    }

    // 3.- Creo un array exactamente igual
    PERSON **sortedArr = (PERSON**)malloc(sizeof(PERSON*) * n);
    if (!sortedArr) return NULL;

    // 4.- Hago pop sobre el nuevo array en orden numerico
    //     el heap los acomoda automaticamente
    for (int i = 0; i < n; i++)
    {
        sortedArr[i] = (PERSON*)hPop(h);
    }

    free(h->elements);
    free(h);


    return sortedArr;
}

int compare(void *person1, void *person2) // Callback necesario en el
{
    return criteria((PERSON*)person1, (PERSON*)person2, eCrit);
}
```

t1 t2 t3 t1 t5 t6

$$T(n) = t1 + \sum_{i=0}^{n-1}(Thpush(i) + t2) + t3 + \sum_{i=0}^{n-1}(Thpop(i) + t4) + t5 + t6$$

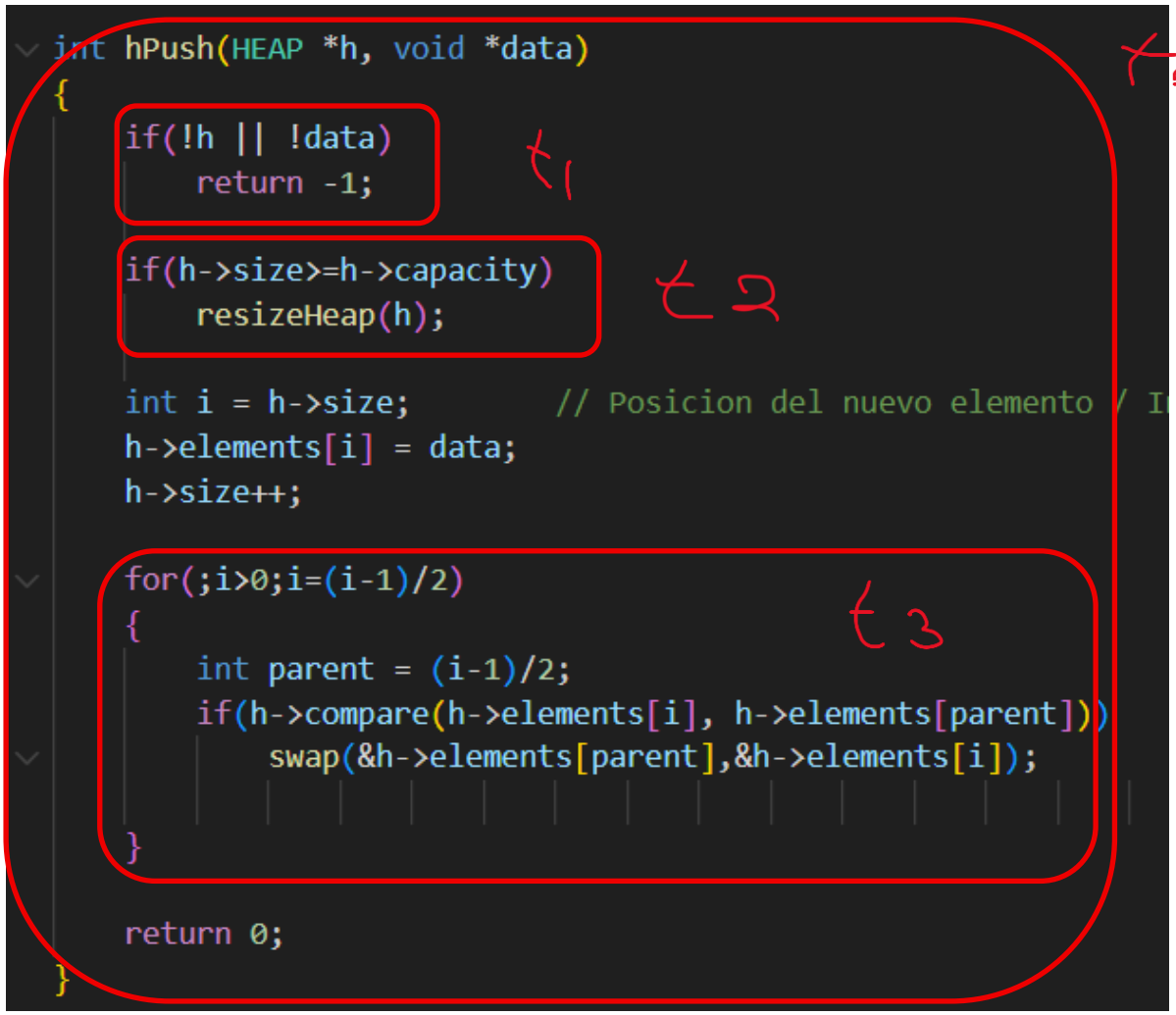
$$T(n) = t1356 + (nlogn - n + \frac{2}{1}logn) + nt2) + (\log n + nt4)$$

$$k1 = t1356 \quad k2 = 1 \;\; k3 = t2 + t4 - 1 \;\; k4 = \frac{3}{2}$$

$$T(n) = k1 + k2\, n\, log\, n + k3n + k4 \log n$$

$$\lim_{n\to\infty}\frac{T(n)}{n\,log\,n}=\lim_{n\to\infty}\frac{k1+k2\,n\,long\,n+k3n+k4\log n}{n\,log\,n}$$
$$=\lim_{n\to\infty}\frac{k1}{n\,log\,n}+k2+\frac{k3}{log\,n}+\frac{k4}{n}=\lim_{n\to\infty}\frac{k1}{\infty}+k2+\frac{k3}{\infty}+\frac{k4}{\infty}=0+0+k2+0=k2$$

$T(n)\ \in\ O(n\,log\,n)$



$$T(n)=t1+\ t2+\sum_{i=1}^{log\,n}(t3)+t4$$

$$T(n)=t124+\sum_{i=1}^{log\,n}(t3)$$

$$T(n)=t124+t3*\ log\,n$$

$$T(n)=t124+t3*\ log\,n$$

$k1 = t124 \quad k2 = t3$

$T(n) = k1 + k2\ log \quad n$

$$\lim_{n\to\infty} \frac{T(n)}{log\ n} = \lim_{n\to\infty} \frac{k1 + k2\ log\ n}{log\ n} = \lim_{n\to\infty} \frac{k1}{log\ n} + k2 = \lim_{n\to\infty} \frac{k1}{\infty} + k2 = 0 + k2 = k2$$

$T(n) \in O(log\ n)$

```
void *hPop(HEAP *h)
{
    if(!h || h->size == 0)
        return NULL;

    int i = 0;                          // el inicio para el hundimiento
    void *temp = h->elements[i];        // Guardamos el dato

    h->elements[i] = h->elements[h->size-1];
    h->size--;

    int hSelect,l,r;

    while(i < h->size/2)
    {
        l = 2*i+1;
        r = 2*i+2;

        if(r<h->size)
            hSelect = h->compare(h->elements[l],h->elements[r]) ? l : r;
        else
            hSelect = l;

        if(h->compare(h->elements[hSelect],h->elements[i])) // Callback que
        {                                                   // Al hundir le
            swap(&h->elements[hSelect],&h->elements[i]);    // Si lo es, se
            i = hSelect;
            continue;
        }
        break;
    }

    return temp;
}
```

t1 t2 t3 t4

$$T(n) = t1 + \ t2 + \sum_{i=1}^{log\ n} (t3) + t4$$

$$T(n) = t124 + \sum_{i=1}^{log\ n} (t3)$$

$T(n) = t124 + t3 * log\ n$

$T(n) = t124 + t3 * log\ n$

$k1 = t124 \quad k2 = t3$

$T(n) = k1 + k2\ log\ n$

$$\lim_{n\to\infty}\frac{T(n)}{log\ n}=\lim_{n\to\infty}\frac{k1+k2\ log\ n}{log\ n}=\lim_{n\to\infty}\frac{k1}{log\ n}+k2=\lim_{n\to\infty}\frac{k1}{\infty}+k2=0+k2=k2$$

$T(n)\ \in\ O(log\ n)$

MergeSort

Este algoritmo de ordenamiento llamado mezcla es basado en el paradigma divide y vencerás. Contamos con la función 'mSort' que divide recursivamente el arreglo de personas hasta que cada sub-arreglo contenga un solo elemento, un arreglo de elementos se considera intrínsecantemente ordenado y utilizando la función 'merge' combinamos los sub-arreglos ordenados de vuelta, mezclándoles de manera que la lista resultante se mantenga ordenada. Es en esta fase mezcla donde realizamos la mayor parte del trabajo de comparación y reordenamiento, garantizando que el resultado sea correcto.

Justificación

Utilizamos MergeSort porque necesitamos un algoritmo estable en esta situación, lo que significa que, si dos personas tienen el mismo valor de riesgo, su orden relativo original se mantiene en el arreglo ordenado. Consideramos el Mergesort para este subproblema en lugar de Quicksort debido a su estabilidad independientemente de la distribución de datos y utilizamos una copia superficial 'mockUp' de los punteros de las estructuras PERSON antes de la recursión para aislar el proceso de ordenamiento de la lista original.

Cálculo de eficiencia y complejidad

```
int mSort(PERSON **arr, int low, int high)
{
    if(low < high)
    {
        int m = low + (high - low) / 2;

        mSort(arr, low, m);
        mSort(arr, m + 1, high);
        merge(arr, low, m, high);
    }
    return 0;
}
```

$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \rightarrow ecuación$

$T(1) = O(1) \rightarrow Caso\ base$

$K = 1$	$T(n) = 2T\left(\frac{n}{2}\right) + n$	$T(n/2) = 2T(n/4) + (n/2)$
$K = 2$	$T(n) = 2[2T(n/4) + (n/2)] + n$ $T(n) = 4T(n/4) + 2n$	$T(n/4) = 2T(n/8) + (n/4)$
$K = 3$	$T(n) = 4[2T(n/8) + (n/4)] + 2n$ $T(n) = 8T(n/8) + 3n$	$T(n/8) = 2T(n/16) + (n/8)$
$K = 4$	$T(n) = 8\left[2T(n/16) + (\frac{n}{8})\right] + 3n$ $T(n) = 16T(n/16) + 4n$	$T(n/16) = 2T(n/32) + (n/16)$

ecuación general

$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$

ecuación caso base

$T(n) = 2^k T(1) + kn$

Buscamos K

$2^k T\left(\frac{n}{2^k}\right) + kn = 2^k T(1) + kn$

$T\left(\frac{n}{2^k}\right) = T(1)$

$\frac{n}{2^k} = 1$

$n = 2k$

$k = log_2 n$

sustituimos k

$T(n) = 2^{log_2 n} T\left(\frac{n}{2^{log_2 n}}\right) + log_2 n * n$

$T(n) = nT\left(\frac{n}{2^{log_2 n}}\right) + log_2 n * n$

$T(n) = n + n\ log_2 n$

ecuación de recurrencia

$T(n) = n + n\ log_2 n$

cota de complejidad

$T(n) = n + n\ log_2 n$

$$\lim_{n\to\infty}\frac{T(n)}{n\,log_2\,n}=\lim_{n\to\infty}\frac{n+n\,log_2\,n}{n\,log_2\,n}=\lim_{n\to\infty}\frac{n+n\,log_2\,n}{n\,log_2\,n}=0+1=1$$

$T(n)\in O(n\,log_2\,n)$

```
int merge(PERSON **arr, int low, int m, int high)
{
    int i, j, k;
    int n1 = m - low + 1;
    int n2 = high - m;

    PERSON **L = (PERSON**)malloc(sizeof(PERSON*)*n1);
    PERSON **R = (PERSON**)malloc(sizeof(PERSON*)*n2);

    for(i=0; i < n1; i++)
        L[i] = arr[low + i];
    for(j=0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = low;

    while(i < n1 && j < n2)
    {
        if(criteria(L[i], R[j], eCrit))
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while(i<n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    while(j<n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }

    free(L);
    free(R);

    return 0;
}
```

t1

t2

t3

t4

t5

t6

$T(n) = t6 + n(t1 + t2 + t3 + t4 + t5)$

$k1 = t6\,, \qquad k2 = (t1 + t2 + t3 + t4 + t5)$

$T(n) = k1 + nk2$

$$\lim_{n\to\infty} \frac{T(n)}{n} = \lim_{n\to\infty} \frac{k1 + nk2}{n} = \lim_{n\to\infty} \frac{k1}{n} + k2 = 0 + k2 = k2$$

$T(n) \in O(n)$

➔ Complejidad general $O(n\ log_2\ n)$ para todos los casos

Quick Sort

El ordenamiento rápido o Quick sort es un algoritmo de ordenamiento con bases similares a las del mergesort con el famoso divide y vencerás, este algoritmo es popular por su eficiencia en la práctica, elegimos un pivote en nuestra implementación se eleige el último elemento del sub-arreglo ‘arr[high]’ como el pivote, la partición ‘partition’ es una de las fases cruciales para reordenar el sub-arreglo de modo que los elementos con prioridad menor que el pivote quedan a la izquierda y todos los elementos con prioridad mayor o igual queden a la derecha. Al finalizar la partición, el pivote se coloca en su posición correcta. Gracias a nuestr ‘qSort’ ordenamos recursivamente los sub-arreglos a la izquierda y a la derecha del pivote.

Justificación

Como ventaja de este algoritmo a comparación de los dos anteriores es la velocidad el Quick sort es generalmente más rápido, porque tiene una menor sobrecarga y mejor aprovechamiento de memoria. aunque estamos utilizando memoria auxiliar con ‘mockUp’ inicialmente, el algoritmo base de partición ‘partition’ es un ordenamiento in-place, lo que lo hace muy eficiente en términos de memoria de pila en comparación con la recursión de Merg sort y al igual que los algoritmos anteriores, utilizamos ‘criteria’ y el criterio global ‘eCrit’ para ordenar dinámicamente por riesgo, nombre o ID.

Cálculo de eficiencia y complejidad

```
int qSort(PERSON **arr, int low, int high)
{
    if(low>=high)
        return 0;

    int p = partition(arr, low, high);
    qSort(arr,low,p-1);
    qSort(arr,p+1,high);
    return 0;
}
```

mejor caso

$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \rightarrow ecuación$

$T(1) = O(1) \rightarrow Caso\ base$

$K = 1$	$T(n) = 2T\left(\frac{n}{2}\right) + n$	$T(n/2) = 2T(n/4) + (n/2)$
$K = 2$	$T(n) = 2[2T(n/4) + (n/2)] + n$ $T(n) = 4T(n/4) + 2n$	$T(n/4) = 2T(n/8) + (n/4)$
$K = 3$	$T(n) = 4[2T(n/8) + (n/4)] + 2n$ $T(n) = 8T(n/8) + 3n$	$T(n/8) = 2T(n/16) + (n/8)$
$K = 4$	$T(n) = 8\left[2T(n/16) + (\frac{n}{8})\right] + 3n$ $T(n) = 16T(n/16) + 4n$	$T(n/16) = 2T(n/32) + (n/16)$

ecuación general

$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$

ecuación caso base

$T(n) = 2^k T(1) + kn$

Buscamos K

$2^k T\left(\frac{n}{2^k}\right) + kn = 2^k T(1) + kn$

$T\left(\frac{n}{2^k}\right) = T(1)$

$\frac{n}{2^k} = 1$

$n = 2k$

$k = log_2 n$

sustituimos k

$T(n) = 2^{log_2 n} T\left(\frac{n}{2^{log_2 n}}\right) + log_2 n * n$

$T(n) = nT\left(\frac{n}{2^{log_2 n}}\right) + log_2 n * n$

$T(n) = n + n\, log_2 n$

ecuación de recurrencia

$T(n) = n + n\, log_2 n$

cota de complejidad

$T(n) = n + n\, log_2 n$

$$\lim_{n\to\infty} \frac{T(n)}{n\, log_2 n} = \lim_{n\to\infty} \frac{n + n\, log_2 n}{n\, log_2 n} = \lim_{n\to\infty} \frac{n + n\, log_2 n}{n\, log_2 n} = 0 + 1 = 1$$

$T(n) \in O(n\, log_2 n)$

caso promedio

$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \rightarrow ecuación$

$T(1) = O(1) \rightarrow Caso\ base$

$K = 1$	$T(n) = 2T\left(\frac{n}{2}\right) + n$	$T(n/2) = 2T(n/4) + (n/2)$
$K = 2$	$T(n) = 2[2T(n/4) + (n/2)] + n$ $T(n) = 4T(n/4) + 2n$	$T(n/4) = 2T(n/8) + (n/4)$
$K = 3$	$T(n) = 4[2T(n/8) + (n/4)] + 2n$ $T(n) = 8T(n/8) + 3n$	$T(n/8) = 2T(n/16) + (n/8)$
$K = 4$	$T(n) = 8\left[2T(n/16) + (\frac{n}{8})\right] + 3n$ $T(n) = 16T(n/16) + 4n$	$T(n/16) = 2T(n/32) + (n/16)$

ecuación general

$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$

ecuación caso base

$T(n) = 2^k T(1) + kn$

Buscamos K

$$2^k T\left(\frac{n}{2^k}\right) + kn = \ 2^k T(1) + kn$$

$$T\left(\frac{n}{2^k}\right) = T(1)$$

$$\frac{n}{2^k} = 1$$

$$n = 2k$$

$$k = \ log_2 n$$

sustituimos k

$$T(n) = \ 2^{log_2 n} T\left(\frac{n}{2^{log_2 n}}\right) + log_2 n * n$$

$$T(n) = \ nT\left(\frac{n}{2^{log_2 n}}\right) + log_2 n * n$$

$$T(n) = \ n + n\ log_2 n$$

ecuación de recurrencia

$$T(n) = \ n + n\ log_2 n$$

cota de complejidad

$$T(n) = \ n + n\ log_2 n$$

$$\lim_{n\to\infty} \frac{T(n)}{n\ log_2 n} = \lim_{n\to\infty} \frac{n + n\ log_2 n}{n\ log_2 n} = \lim_{n\to\infty} \frac{n + n\ log_2 n}{n\ log_2 n} = \ 0 + 1 = 1$$

$$T(n) \in O(n\ log_2 n)$$

peor caso

$T(n) = \ T(n-1) + \ O(n) \rightarrow ecuación$
$T(1) = \ O(1) \rightarrow Caso\ base$

$K = 1$	$T(n) = T(n-1) + n$	$T(n-1) = T(n-2) + (n-1)$
$K = 2$	$T(n) = [T(n-2) + (n-1)] + n$ $T(n) = T(n-2) + (n-1) + n$ $T(n) = T(n-2) + 2n - 1$	$T(n-2) = T(n-3) + (n-2)$
$K = 3$	$T(n) = [T(n-3) + (n-2)]2n - 1$ $T(n) = T(n-3) + 3n - 3$	$T(n-3) = T(n-4) + (n-3)$
$K = 4$	$T(n) = [T(n-4) + (n-3)] + 3n - 3$ $T(n) = T(n-4) + 4n - 6$	$T(n-4) = T(n-5) + (n-4)$

ecuación general

$$T(n) = \ T(n-k) + kn - \frac{k(k-1)}{2}$$

ecuación caso base

$$T(1) = T(1) + kn - \frac{k(k-1)}{2}$$

Buscamos K

$$T(n-k) + kn - \frac{k(k-1)}{2} = T(1) + kn - \frac{k(k-1)}{2}$$

$T(n-k) = T(1)$

$n - k = 1$

$n - 1 = k$

$k = n - 1$

sustituimos k

$$T(n) = (1) + (n-1)n - \frac{(n-1)(n-1-1)}{2}$$

$$T(n) = 1 + n^2 - n - \frac{(n-1)(n-2)}{2}$$

$$T(n) = 1 + n^2 - n - \frac{n^2}{2} + \frac{3n}{2} - 1$$

$$T(n) = \frac{n^2}{2} + \frac{1n}{2}$$

ecuación de recurrencia

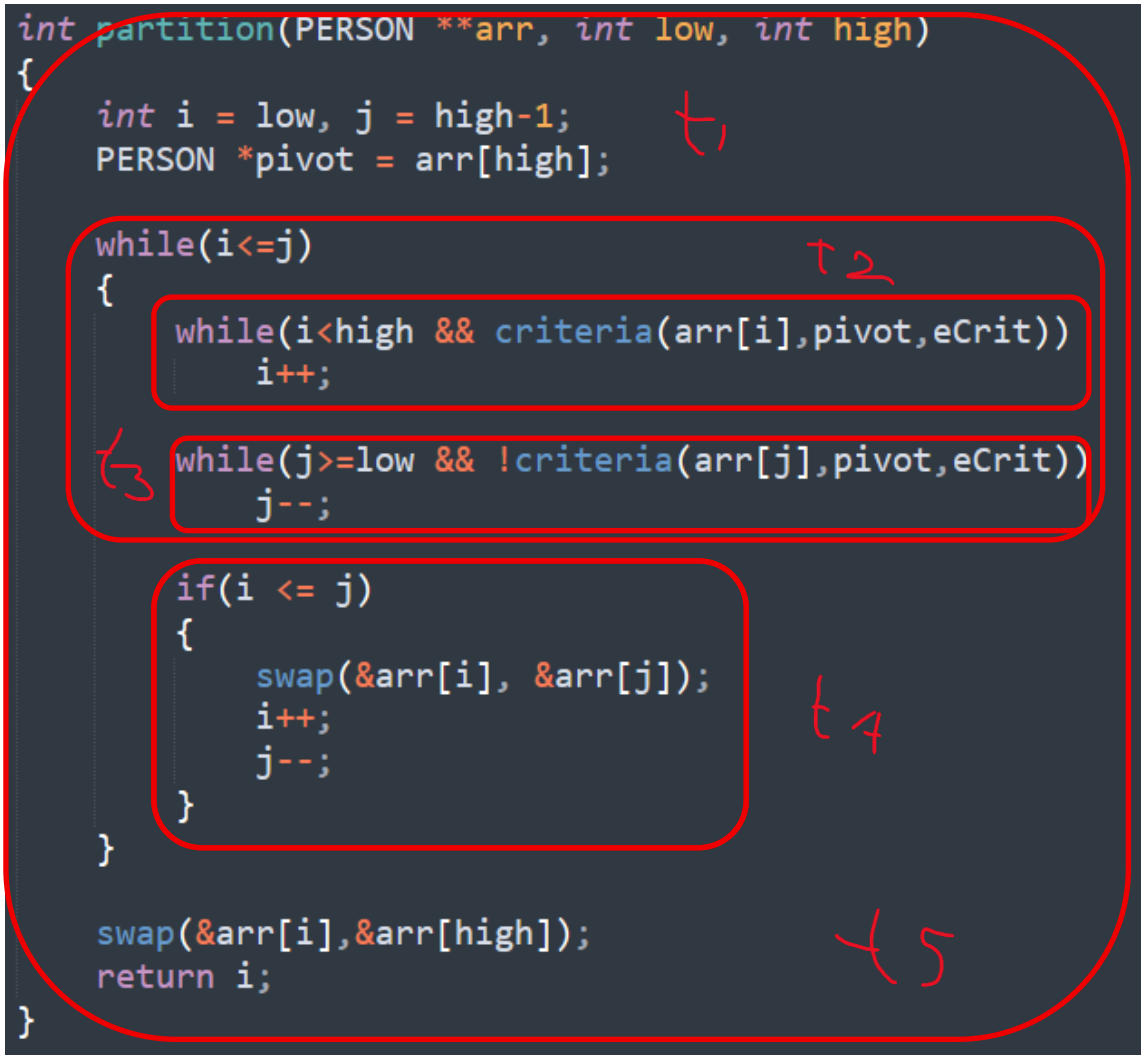
$$T(n) = \frac{n^2}{2} + \frac{1n}{2}$$

cota de complejidad

$$T(n) = \frac{n^2}{2} + \frac{1n}{2}$$

$$\lim_{n\to\infty} \frac{T(n)}{n^2} = \lim_{n\to\infty} \frac{\frac{n^2}{2} + \frac{n}{2}}{n^2} = \lim_{n\to\infty} \frac{n^2}{2n^2} + \frac{n}{2n^2} = \lim_{n\to\infty} \frac{1}{2} + \frac{1}{2n} = \frac{1}{2}$$

$T(n) \in O(n^2)$



$T(n) = t5 + n(t1 + t2 + t3 + t4)$

$k1 = t5\,, \qquad k2 = (t1 + t2 + t3 + t4)$

$T(n) = k1 + nk2$

$$\lim_{n\to\infty} \frac{T(n)}{n} = \lim_{n\to\infty} \frac{k1 + nk2}{n} = \lim_{n\to\infty} \frac{k1}{n} + k2 = 0 + k2 = k2$$

$T(n) \in O(n)$

➔ Complejidad general $(n\ log_2\ n)$ para mejor y caso promedio y en peor caso $O(n^2)$

Detección de brotes

BSF

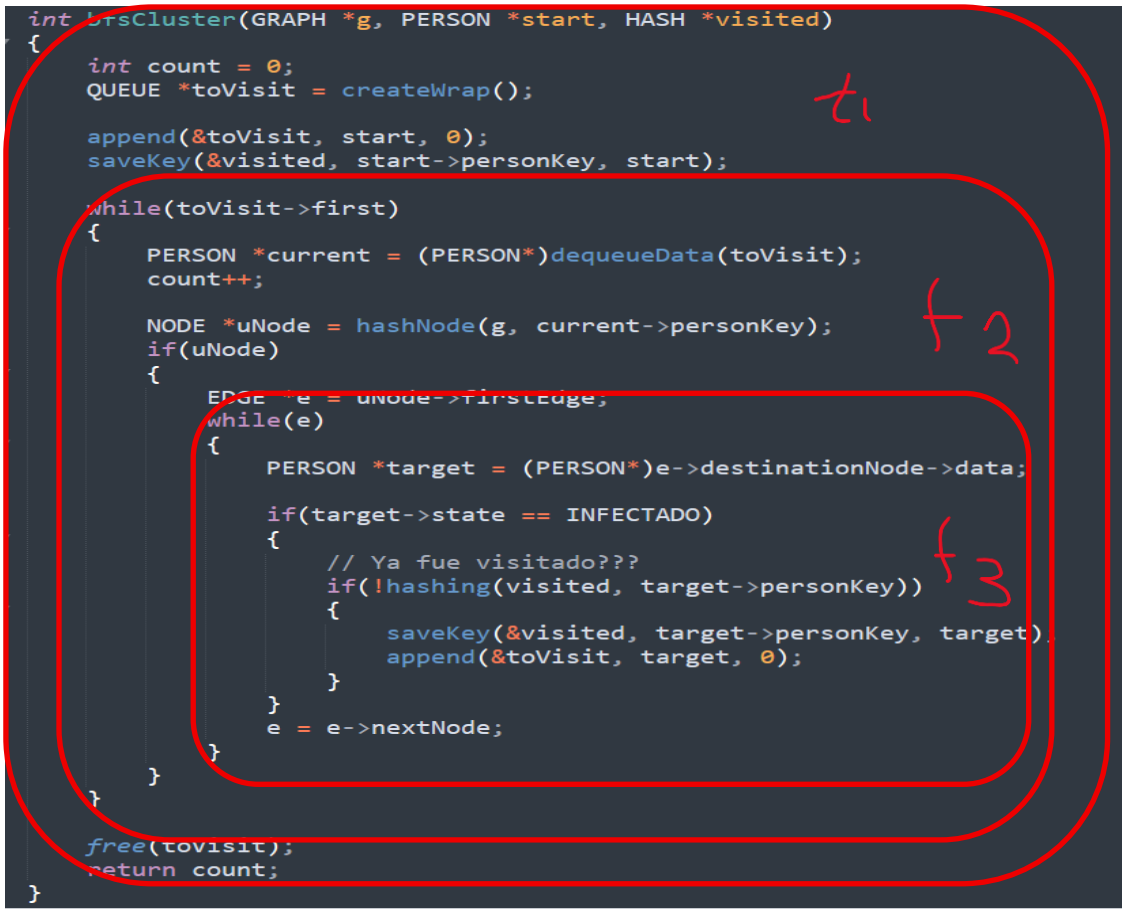
En este módulo utilizamos la búsqueda en amplitud para identificar y medir la extensión de los focos de infección dentro de la ciudad. En el código la función 'detectOutbreak' es la principal que itera sobre 'infectedList' lista de todos los

infectados de la ciudad, utilizamos un control de vistas utilizando hash en ‘visited’ para registrar a cada persona que ya ha sido contada en un clúster esto nos garantiza que un brote grande solo se procese una vez, utilizando ‘bsfCluster’ si una persona infectada no ha sido visitada, se inicia un BSF desde ella, contamos con una condición de propagación ‘solo entre vecinos infectados’ target->state == INFECTADO . Si un vecino es susceptible o vacunado el BFS se detiene en ese punto, delimitando la frontera del brote activo, este proceso se repite hasta que todos los infectados hayan sido asignados a un clúster. La función principal retorna el tamaño del brote más grande encontrado.

justificación

Decidimos que el algoritmo perfecto para esta situación es el BSF por la forma rápida de garantizar que se visitan todos los nodos conectados por aristas siendo muy eficiente en cabio la búsqueda por DFS es un tanto menos intuitivo, además explora todos los caminos mientras que BSF encuentra el camino más corto.

Cálculo de eficiencia y complejidad



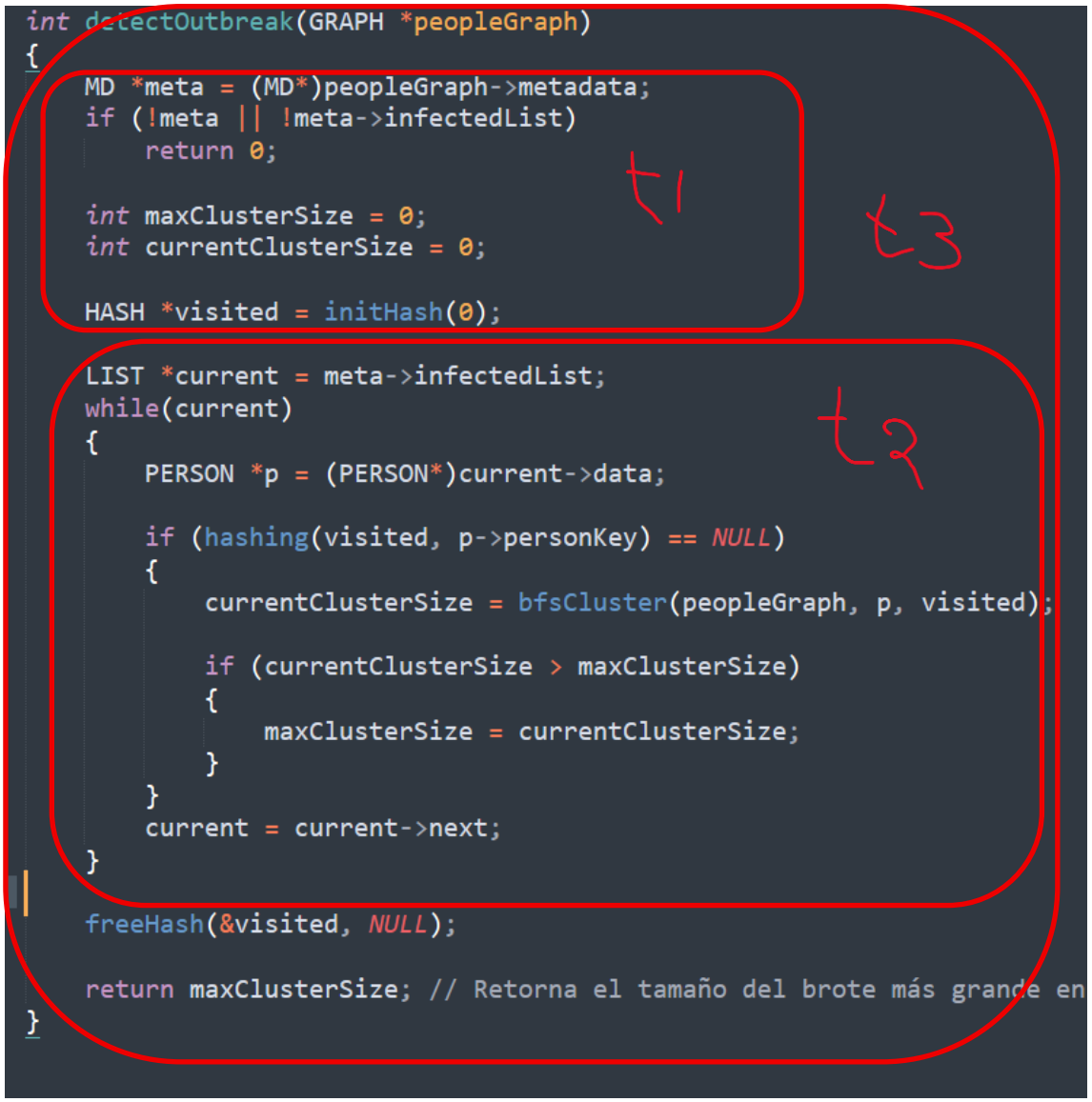
$T(n) = t1 + n\,t2 + n\,t3$

$k1 = t1 \quad k2 = t2 + t3$

$T(n) = k1 + nk2$

$$\lim_{n\to\infty} \frac{T(n)}{n} = \lim_{n\to\infty} \frac{k1 + nk2}{n} = \lim_{n\to\infty} \frac{k1}{n} + k2 = 0 + k2 = k2$$

$T(n) \in O(n)$




```
int detectOutbreak(GRAPH *peopleGraph)
{
    MD *meta = (MD*)peopleGraph->metadata;
    if (!meta || !meta->infectedList)
        return 0;

    int maxClusterSize = 0;
    int currentClusterSize = 0;

    HASH *visited = initHash(0);

    LIST *current = meta->infectedList;
    while(current)
    {
        PERSON *p = (PERSON*)current->data;

        if (hashing(visited, p->personKey) == NULL)
        {
            currentClusterSize = bfsCluster(peopleGraph, p, visited);

            if (currentClusterSize > maxClusterSize)
            {
                maxClusterSize = currentClusterSize;
            }
        }
        current = current->next;
    }

    freeHash(&visited, NULL);

    return maxClusterSize; // Retorna el tamaño del brote más grande en
}
```

$T(n) = t1 + n\,t2 + n\,t3$

$k1 = t1 \quad k2 = t2 + t3$

$T(n) = k1 + nk2$

$$\lim_{n\to\infty} \frac{T(n)}{n} = \lim_{n\to\infty} \frac{k1 + nk2}{n} = \lim_{n\to\infty} \frac{k1}{n} + k2 = 0 + k2 = k2$$

$T(n) \in O(n)$

➔ Complejidad general de O(n)

Propagación Temporal (Simulación)

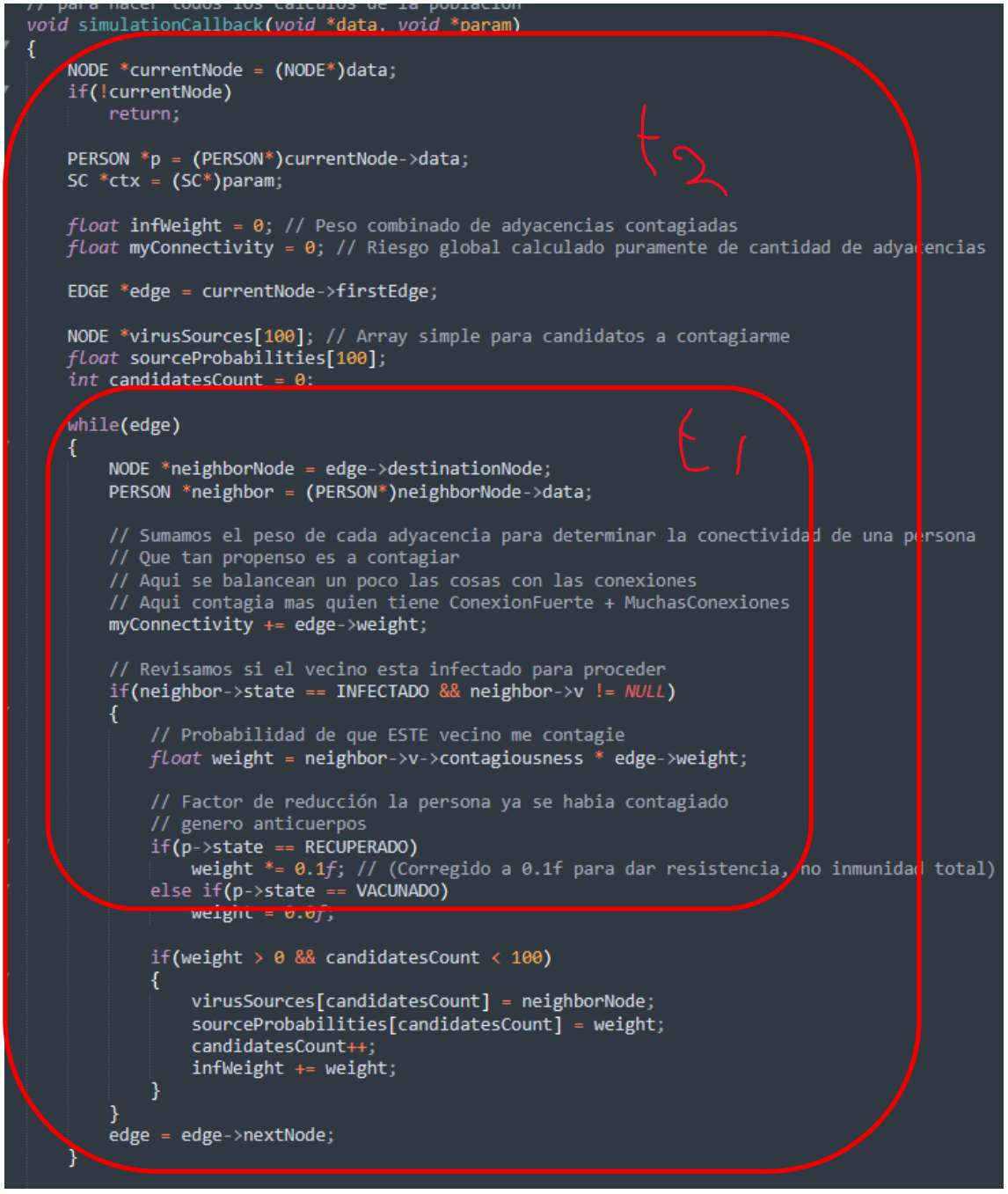
BSF

El algoritmo utiliza un BFS modificado que opera en el grafo de personas. Su objetivo principal es garantizar una exploración sistemática y completa de la red de contactos sin enfocarnos en encontrar la ruta más corta, sino evaluar los nodos. Se gestiona el orden de visita de los nodos que significa una evaluación a los vecinos más cercanos antes de pasar a los más lejanos, utilizamos el BSF para garantizar la iteración de las personas y el contacto por día 'cbCalculation' avanza la simulación un día actualizando el estado de cada persona en función de las probabilidades de contagio, muerte y mutación de virus, nuestro 'stepSimulation' llama a 'traverseGraphParameter' para ejecutar el 'SimulationCallback' para garantizar con esto que cada persona en el grafo esta siendo evaluada en cada ciclo.

Justificación

El uso de BSF es indispensable para esta fase, es un método para garantizar que cada nodo y cada arista sean inspeccionados exactamente una vez por día, lo utilizamos porque es más necesario que otro algoritmo como Dijkstra que es innecesario, porque para este módulo nuestro objetivo es evaluar el estado del nodo y no encontrar el camino más optimo, por eso rechazamos la idea de utilizar Dijkstra.

Cálculo de eficiencia y complejidad



```
// para hacer todos los calculos de la poblacion
void simulationCallback(void *data, void *param)
{
    NODE *currentNode = (NODE*)data;
    if(!currentNode)
        return;

    PERSON *p = (PERSON*)currentNode->data;
    SC *ctx = (SC*)param;

    float infWeight = 0; // Peso combinado de adyacencias contagiadas
    float myConnectivity = 0; // Riesgo global calculado puramente de cantidad de adyacencias

    EDGE *edge = currentNode->firstEdge;

    NODE *virusSources[100]; // Array simple para candidatos a contagiarme
    float sourceProbabilities[100];
    int candidatesCount = 0;

    while(edge)
    {
        NODE *neighborNode = edge->destinationNode;
        PERSON *neighbor = (PERSON*)neighborNode->data;

        // Sumamos el peso de cada adyacencia para determinar la conectividad de una persona
        // Que tan propenso es a contagiar
        // Aqui se balancean un poco las cosas con las conexiones
        // Aqui contagia mas quien tiene ConexionFuerte + MuchasConexiones
        myConnectivity += edge->weight;

        // Revisamos si el vecino esta infectado para proceder
        if(neighbor->state == INFECTADO && neighbor->v != NULL)
        {
            // Probabilidad de que ESTE vecino me contagie
            float weight = neighbor->v->contagiousness * edge->weight;

            // Factor de reducción la persona ya se habia contagiado
            // genero anticuerpos
            if(p->state == RECUPERADO)
                weight *= 0.1f; // (Corregido a 0.1f para dar resistencia, no inmunidad total)
            else if(p->state == VACUNADO)
                weight = 0.0f;

            if(weight > 0 && candidatesCount < 100)
            {
                virusSources[candidatesCount] = neighborNode;
                sourceProbabilities[candidatesCount] = weight;
                candidatesCount++;
                infWeight += weight;
            }
        }
        edge = edge->nextNode;
    }
```

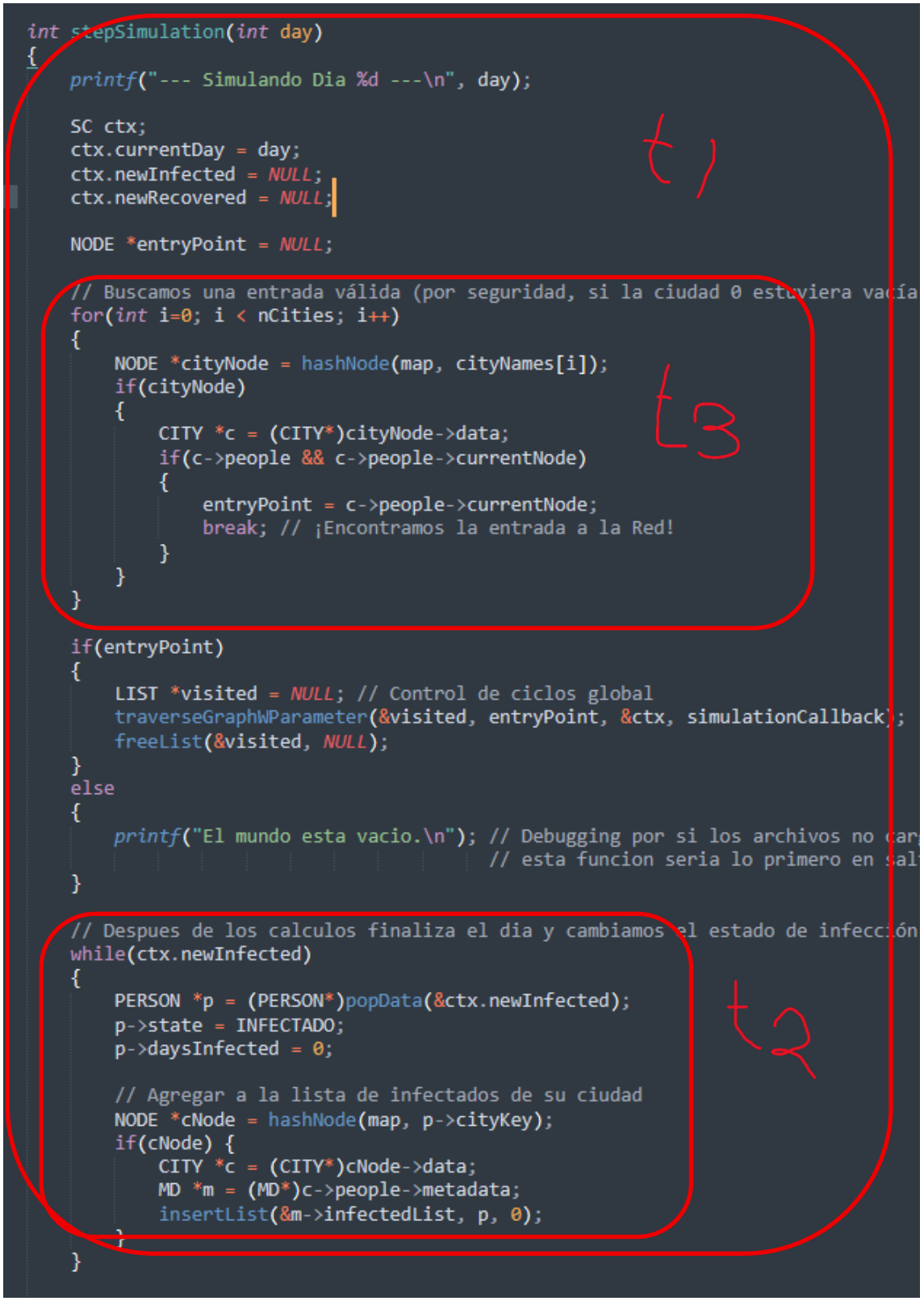
$T(n) = nt1 + t2$

$k1 = t1 \quad k2 = t2$

$T(n) = nk1 + k2$

$$\lim_{n\to\infty} \frac{T(n)}{n} = \lim_{n\to\infty} \frac{k2 + nk1}{n} = \lim_{n\to\infty} \frac{k2}{n} + k1 = 0 + k1 = k1$$

$T(n) \in O(n)$



$T(n) = t1 + n\,t2 + n\,t3$

$k1 = t1 \quad k2 = t2 + t3$

$T(n) = k1 + nk2$

$$\lim_{n\to\infty} \frac{T(n)}{n} = \lim_{n\to\infty} \frac{k1 + nk2}{n} = \lim_{n\to\infty} \frac{k1}{n} + k2 = 0 + k2 = k2$$

$T(n) \in O(n)$

➔ Complejidad general de O(n)

Minimización del Riesgo Total

Greedy con Max-Heap

La función ‘applySmartQuarantine’ implementa la estrategia voraz ‘Greedy’ para seleccionar individuos y aislarlos. El objetivo es obtener la máxima reducción de riesgo global por unidades de presupuesto, asumimos que ‘p->globalRisk’ ya ha sido calculado previamente, construimos un Heap de máximos ‘riskHeap’ utilizando el comparador ‘compareRisk’ el cual nos garantiza que la persona con el mayor riesgo global siempre está en la cima. gracias a la selección greey consumimos o extraemos repetidamente a la persona de la cima, dado que heap siempre devuelve al candidato más peligroso y se marca a la persona como ‘ p->quarantine = true’
y se mueve a nuestro array de cuarentena para una gestión rápida.

Utilizamos una función auxiliar que es nuestro criterio de recuperación, define la prioridad para ordenar a las personas en cuarentena (recuperados o fallecidos para su liberación inmediata) y si el estado es igual utilizamos ‘daysInfected’ para ver quien ha estado enfermo más tiempo.

Justificación

se utiliza el paradigma de Greedy por la naturaleza del problema de suma de riesgos, permitiendo tomar soluciones óptimas y sumándole el rol que cumple el Max-heap para comparar y ejecutar Greedy con eficiencia ya que la búsqueda heap cuesta o (1), utilizar alguna otra alternativa como Quicksort o Mergesort necesitamos ordenar la población lo cual nos da más trabajo y es ineficiente.

Cálculo de eficiencia y complejidad

```
int releaseRecovered(CITY *c)
{
    if(!c)
        return -1;
    MD *meta = (MD*)c->people->metadata;
    if(meta->quarantined == 0)
        return -1;

    // Borrado seguro al reves
    for(int i = meta->quarantined - 1; i >= 0; i--)
    {
        PERSON *p = meta->quarantineArray[i];

        // Checamos si ya se recuperó o falleció
        if(p->state == RECUPERADO || p->state == FALLECIDO)
        {
            p->quarantine = false;

            // Eliminar del Array (Swap con el último, esto deja el espacio libre como
            meta->quarantineArray[i] = meta->quarantineArray[meta->quarantined - 1];
            meta->quarantineArray[meta->quarantined - 1] = NULL;
            meta->quarantined--;
        }
    }

    if(meta->quarantined > 1)
    {
        // Configuramos el criterio global antes de ordenar
        eCrit = 4;

        // Llamamos a tu mSort recursivo directamente sobre el array de cuarentena
        // mergeSort devuelve una copia asi que no lo necesitamos aqui
        mSort(meta->quarantineArray, 0, meta->quarantined - 1);
    }

    return 0;
}
```

$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \rightarrow ecuación$

$T(1) = O(1) \rightarrow Caso\ base$

$K = 1$	$T(n) = 2T\left(\frac{n}{2}\right) + n$	$T(n/2) = 2T(n/4) + (n/2)$
$K = 2$	$T(n) = 2[2T(n/4) + (n/2)] + n$ $T(n) = 4T(n/4) + 2n$	$T(n/4) = 2T(n/8) + (n/4)$
$K = 3$	$T(n) = 4[2T(n/8) + (n/4)] + 2n$ $T(n) = 8T(n/8) + 3n$	$T(n/8) = 2T(n/16) + (n/8)$
$K = 4$	$T(n) = 8\left[2T(n/16) + (\frac{n}{8})\right] + 3n$ $T(n) = 16T(n/16) + 4n$	$T(n/16) = 2T(n/32) + (n/16)$

ecuación general

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

ecuación caso base

$$T(n) = 2^k T(1) + kn$$

Buscamos K

$$2^k T\left(\frac{n}{2^k}\right) + kn = 2^k T(1) + kn$$

$$T\left(\frac{n}{2^k}\right) = T(1)$$

$$\frac{n}{2^k} = 1$$

$$n = 2k$$

$$k = log_2 n$$

sustituimos k

$$T(n) = 2^{log_2 n} T\left(\frac{n}{2^{log_2 n}}\right) + log_2 n * n$$

$$T(n) = nT\left(\frac{n}{2^{log_2 n}}\right) + log_2 n * n$$

$$T(n) = n + n\, log_2 n$$

ecuación de recurrencia

$$T(n) = n + n\, log_2 n$$

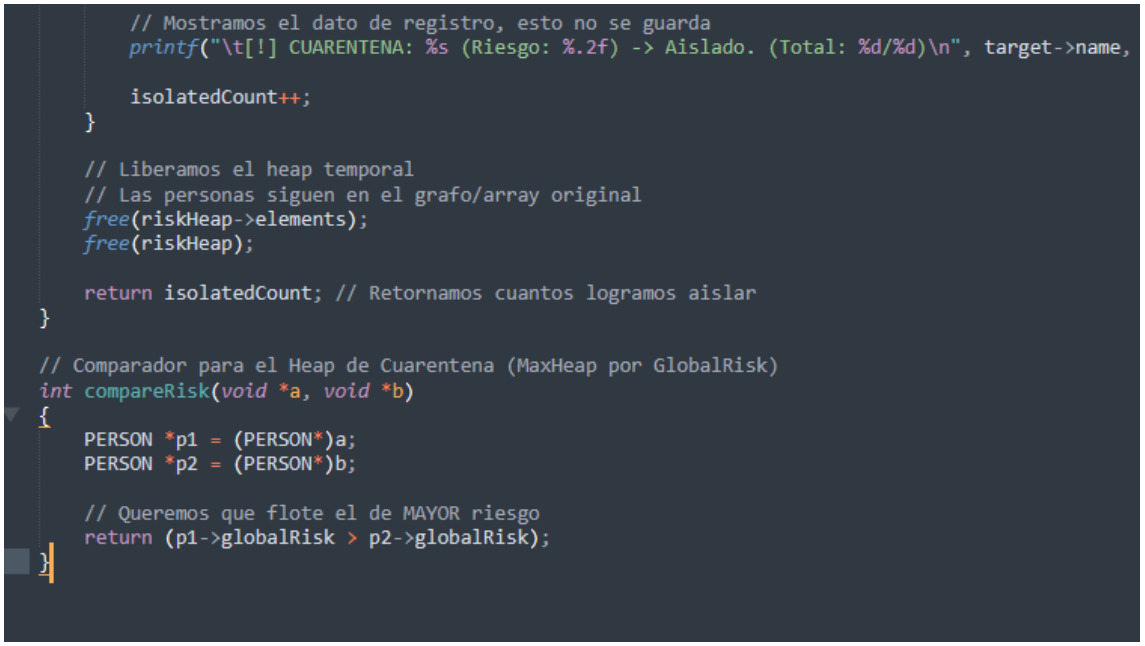
cota de complejidad

$$T(n) = n + n\, log_2 n$$

$$\lim_{n\to\infty} \frac{T(n)}{n\, log_2 n} = \lim_{n\to\infty} \frac{n + n\, log_2 n}{n\, log_2 n} = \lim_{n\to\infty} \frac{n + n\, log_2 n}{n\, log_2 n} = 0 + 1 = 1$$

$$T(n) \in O(n\, log_2 n)$$

```c
int applySmartQuarantine(CITY *c, int budget)
{
    if(!c || budget <= 0)
        return -1;

    MD *meta = (MD*)c->people->metadata;
    int maxCapacity = (int)(meta->nPopulation * 0.3); // Capacidad total del array (30%)

    // Si ya esta lleno el bunker, no cabe nadie mas
    if(meta->quarantined >= maxCapacity)
        return 0;

    // Inicializamos el heap temporal
    HEAP *riskHeap = initHeap(meta->nPopulation, compareRisk);
    if(!riskHeap)
        return -1;

    // Llenado del Heap
    // Recorremos el array de toda la poblacion linealmente
    for(int i = 0; i < meta->nPopulation; i++)
    {
        PERSON *p = meta->population[i];

        // Solo consideramos candidatos validos:
        // - Vivos
        // - Que no esten en cuarentena
        // - Con riesgo > 0 (si alguien no tiene riesgo no sirve de nada aislarlo)
        if(p->state != FALLECIDO && !p->quarantine && p->globalRisk > 0)
        {
            hPush(riskHeap, p); // Empujamos el nuevo propenso
        }
    }

    printf("Analisis completado. Candidatos a cuarentena: %d\n", riskHeap->size);

    // Metemos en cuarentena a cuantos hayan sido designados
    int isolatedCount = 0;

    // Mientras tengamos presupuesto, candidatos y ESPACIO en el array
    while(isolatedCount < budget && riskHeap->size > 0 && meta->quarantined < maxCapacity)
    {
        PERSON *target = (PERSON*)hPop(riskHeap);

        // Aplicar aislamiento
        target->quarantine = true; // Un individuo solo puede salir de cuarentena si...
                                   // se recupero la persona (esto se gestiona en otra funció

        // Lo movemos al array de cuarentena para gestion rapida
        // Usamos el contador actual como indice y LUEGO incrementamos
        meta->quarantineArray[meta->quarantined] = target;
        meta->quarantined++;

        // Mostramos el dato de registro, esto no se guarda
        printf("\t[!] CUARENTENA: %s (Riesgo: %.2f) -> Aislado. (Total: %d/%d)\n", target->na
```



```
            // Mostramos el dato de registro, esto no se guarda
            printf("\t[!] CUARENTENA: %s (Riesgo: %.2f) -> Aislado. (Total: %d/%d)\n", target->name,

            isolatedCount++;
        }

        // Liberamos el heap temporal
        // Las personas siguen en el grafo/array original
        free(riskHeap->elements);
        free(riskHeap);

        return isolatedCount; // Retornamos cuantos logramos aislar
    }

    // Comparador para el Heap de Cuarentena (MaxHeap por GlobalRisk)
    int compareRisk(void *a, void *b)
    {
        PERSON *p1 = (PERSON*)a;
        PERSON *p2 = (PERSON*)b;

        // Queremos que flote el de MAYOR riesgo
        return (p1->globalRisk > p2->globalRisk);
    }
```

Esta función es de complejidad O (n log n) esto se debe a que en el bucle for iteramos sobre toda la población (N) y cada operación hPush cuesta O(log n) el cálculo de eficiencia y las cotas de complejidad de hPop y hPush se encuentra en la parte superior en donde vimos el algoritmo Heap por eso sabemos que este apartado de código es de complejidad O (n log n).

➔ Complejidad general O (n log n)

Identificación de Rutas Críticas (Dijkstra)

Dijkstra

Utilizamos el algoritmo de Dijkstra para encontrar el camino de mayor probabilidad de infección entre dos personas ‘start’ y ‘target’ con la función ‘findCriticalPath’ el problema al que nos enfrentamos es que maximizar la probabilidad compuesta en pocas palabras la multiplicación de probabilidades de contagio se convierte en un problema de minimizar un costo que el formato de Dijkstra requiere un peso en la arista y el costo de la arista , al minimizar el costo maximizamos la probabilidad de contagio la ruta con el menor costo es la ruta con la mayor probabilidad de infección,

utilizamos como estructuras claves el min-heap para extraer el nodo con el menor costo acumulado más rápidamente y con las tablas hash almacenamos la distancia más corta conocida y el nodo padre.

Justificación

Utilizamos Dijkstra porque para este tipo de situaciones no tiene deficiencias es un algoritmo estándar eficiente para encontrar la ruta optima con costos positivos, no utilizamos BSF O DFS porque encontrar una ruta con menos contactos es su propósito, no puede manejar pesos de probabilidad, por lo que no va a encontrar una ruta con la mayor probabilidad acumulada.

Cálculo de eficiencia y complejidad

```c
#include <math.h>
#include <float.h>


//MinHeap para dijkstra
int compareDijkstra(void *a, void *b)
{
    D_STATE *s1 = (D_STATE*)a;
    D_STATE *s2 = (D_STATE*)b;

    return (s1->cost < s2->cost);
}


LIST *findCriticalPath(NODE *start, NODE *target)
{
    if(!start || !target)
        return NULL;

    printf("[DIJKSTRA] Buscando camino critico de %s a %s...\n",((PERSON*)start->data)->name, ((PERSON*)target->data)->name);

    // Hash para arreglo de distancia y camino
    // el acceso con clave lo hace mas atractivo
    // en terminos de practicidad
    HASH *distMap = initHash(0);
    HASH *parentMap = initHash(0);

    // Heap de Prioridad
    // El tamaño inicial resulta un poco irrelevante ya que se redimensiona solo al llenarse
    HEAP *priorityQueue = initHeap(100, compareDijkstra);

    // Aqui se nota como brilla el hash NOMBRE:VALOR
    // no necesitamos trackear el id de nadie, a nivel logico
    // es mas obvio decirle a la computadora "Fulanito"
    // y sin perder ese valioso tiempo O(1)
    float *startDist = (float*)malloc(sizeof(float));
    *startDist = 0.0f;
    saveKey(&distMap, start->nodeKey, startDist);

    // Push inicial al heap
    D_STATE *initialState = createDstate(start,0);

    hPush(priorityQueue, initialState);

    NODE *current = NULL;
    int found = 0;

    while(priorityQueue->size > 0) // Siempre que haya nodos a explorar
    {
        D_STATE *state = (D_STATE*)hPop(priorityQueue);
        current = state->currentNode;
        float currentCost = state->cost;
        free(state); // Liberamos el wrapper, no el nodo

        // Si llegamos al destino, terminamos
```

```
    // Si llegamos al destino, terminamos
    if(current == target) // Comparacion de punteros directa
    {
        found = 1;
        break;
    }

    // Revisar si encontramos un camino mas corto a 'current'
    EHASH *dElem = hashing(distMap, current->nodeKey);
    if(dElem)
    {
        float bestKnown = *(float*)dElem->pair;
        if(currentCost > bestKnown)
            continue;
    }

    // Exploramos vecinos
    EDGE *edge = current->firstEdge;
    while(edge)
    {
        NODE *neighbor = edge->destinationNode;

        // Calculo del Costo de la Arista
        // Peso original (probabilidad): 0.1 a 1.0
        // Costo Dijkstra: -log(peso)
        // Evitamos log(0)

        float prob = edge->weight;
        if(prob <= 0.0001f) prob = 0.0001f;

        float edgeCost = -logf(prob);
        float newDist = currentCost + edgeCost;

        // Revisar distancia actual conocida
        float oldDist = FLT_MAX; // Distancia mas grande que cualquier otra
                                 // basicamente el centinela para intercambiar la distancia la primera vez
        EHASH *nElem = hashing(distMap, neighbor->nodeKey);
        if(nElem)
            oldDist = *(float*)nElem->pair;

        // Comprobamos si la distancia es correcta
        if(newDist < oldDist)
        {
            // Actualizamos distancia
            float *newDistPtr = (float*)malloc(sizeof(float));
            *newDistPtr = newDist;
            // saveKey simula actualizacion si ya existe
            // como el metodo de colisiones es chaining
            // el primer elemento del hash con la misma llave es el ultimo insertado
            saveKey(&distMap, neighbor->nodeKey, newDistPtr);

            // Actualizamos padre
            saveKey(&parentMap, neighbor->nodeKey, current);

            // Push al heap
            D_STATE *nextState = createDstate(neighbor,newDist);
```

```
                // Actualizamos padre
                saveKey(&parentMap, neighbor->nodeKey, current);

                // Push al heap
                D_STATE *nextState = createDstate(neighbor,newDist);
                hPush(priorityQueue, nextState);
            }

            edge = edge->nextNode;
        }
    }

    // Reconstruimos el camino
    LIST *pathList = NULL;

    if(found)
    {
        NODE *path = target;
        while(path)
        {
            // Insertamos al inicio de la lista para tener orden Origen -> Destino
            insertList(&pathList, path, 0);

            if(path == start)
                break;

            // Buscar al padre de 'path'
            // en nuestro mapa hash, decimos "Quien fue el que tuvo la menor distancia a este nodo?"
            // importante aclarar aqui que el nodo inicial no genero un padre para su llave, por lo que la
            EHASH *pElem = hashing(parentMap, path->nodeKey);
            if(pElem)
                path = (NODE*)pElem->pair; // Cuando lo encontramos actualizamos path y volvemos a intentar
            else
                path = NULL; // Camino roto (no deberia pasar si found=1)
        }
    }

    free(priorityQueue->elements);
    free(priorityQueue);

   // Liberamos el mapa de distancias (Borramos la tabla Y los floats)
    freeHash(&distMap, destroyFloat);

    // Liberamos el mapa de padres (Borramos la tabla pero NO los nodos, porque son del grafo)
    freeHash(&parentMap, NULL);

    return pathList;
}

void destroyFloat(void *data)
{
    if(data)
        free(data); // ¡Totalmente válido y seguro!
}
```

complejidad O((m+n)log n) podemos saber que tenemos complejidad O(n log n) debido a que utilizamos Heap sort (el caálculo se encuentra en ese apartado) obtenemos esa complejidad debido a que nuestro hPop se ejecuta m veces para extraer el nodo con prioridad y nuestro hPush se ejecuta hasta n veces para actualizar las distancias estas complejidades ya fueron calculadas anteriormente.

Clustering de Cepas

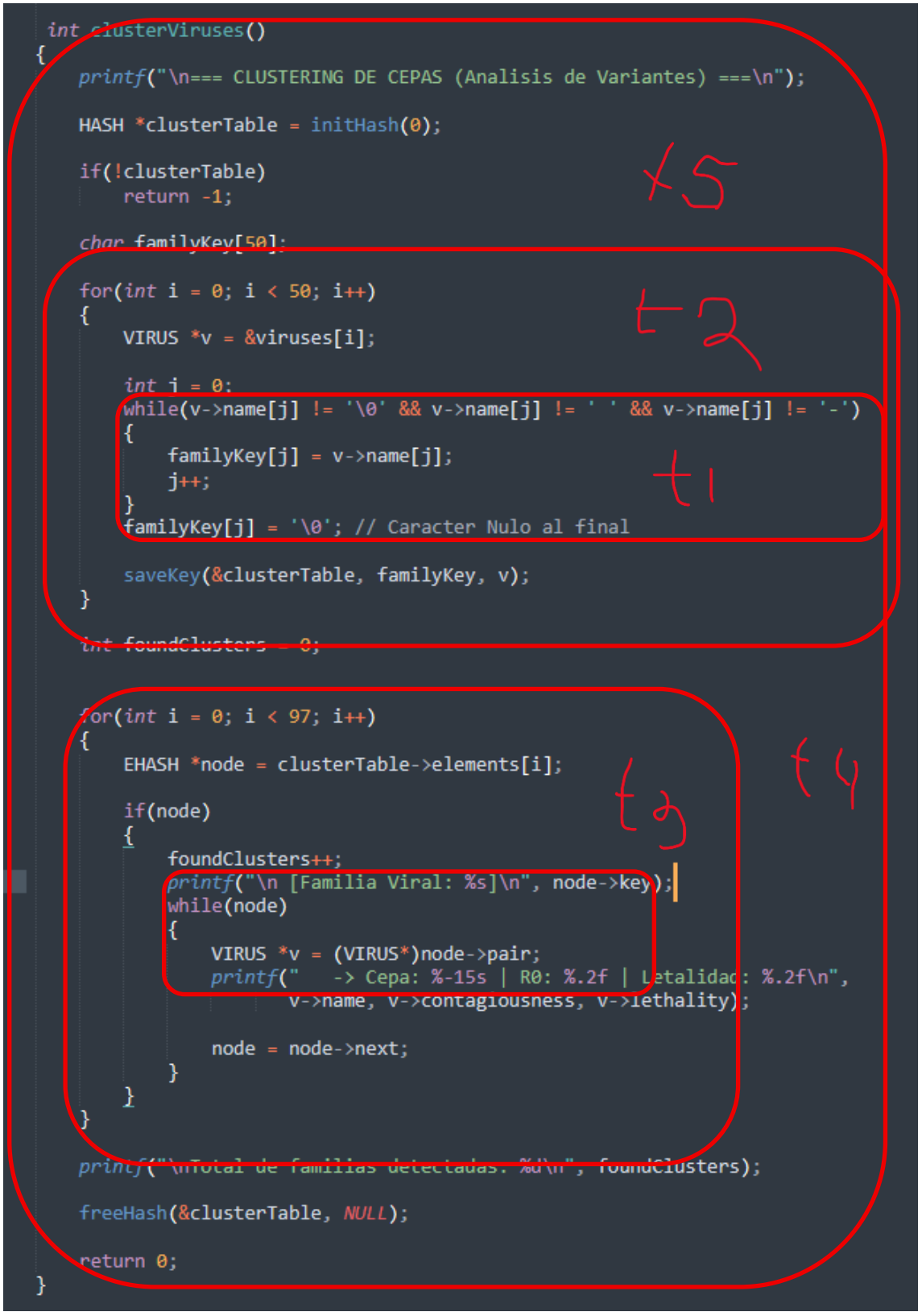
Clustering (tabla hash con encadenamiento)

La función ‘clusterViruses’ utiliza una tabla Hash con encadenamiento para agrupar las 50 cepas ‘viruses[50]’ en familias basándose en la primera palabra de su nombre. El código itera ssobre el nombre de cada virus hasta encontrar un espacio o guion, creando la clave de una familia ‘fmailyKey’, utilizando ‘saveKey’ para insertar el virus usando esta clave, si dos cepas comparten la misma clave ambas se hacen a la misma cubeta y se enlazan creando un clúster viral. la función recorre el arreglo interno de la tabla e itera sobre las listas enlazadas para imprimir cada familia y sus variantes.

Justificación

Este algoritmo es elegido porque es el más eficiente para agrupar N elementos basándose en una clave de longitud, evita una sobrecarga del ordenamiento en cambio sí utilizábamos una búsqueda lineal esta solución era más ineficiente ay que requiere que cada nuevo virus compare su clave con todos los virus previamente agrupados.

Cálculo de eficiencia y complejidad



$T(n) = t5 + n\,t2 + n\,t3 + n\,t1 + n\,t4$

$k1 = t5 \quad k2 = t2 + t3 + t1 + t4$

$T(n) = k1 + nk2$

$$\lim_{n\to\infty} \frac{T(n)}{n} = \lim_{n\to\infty} \frac{k1 + nk2}{n} = \lim_{n\to\infty} \frac{k1}{n} + k2 = 0 + k2 = k2$$

$T(n) \in O(n)$

Almacenamiento y Consulta (La "BD")

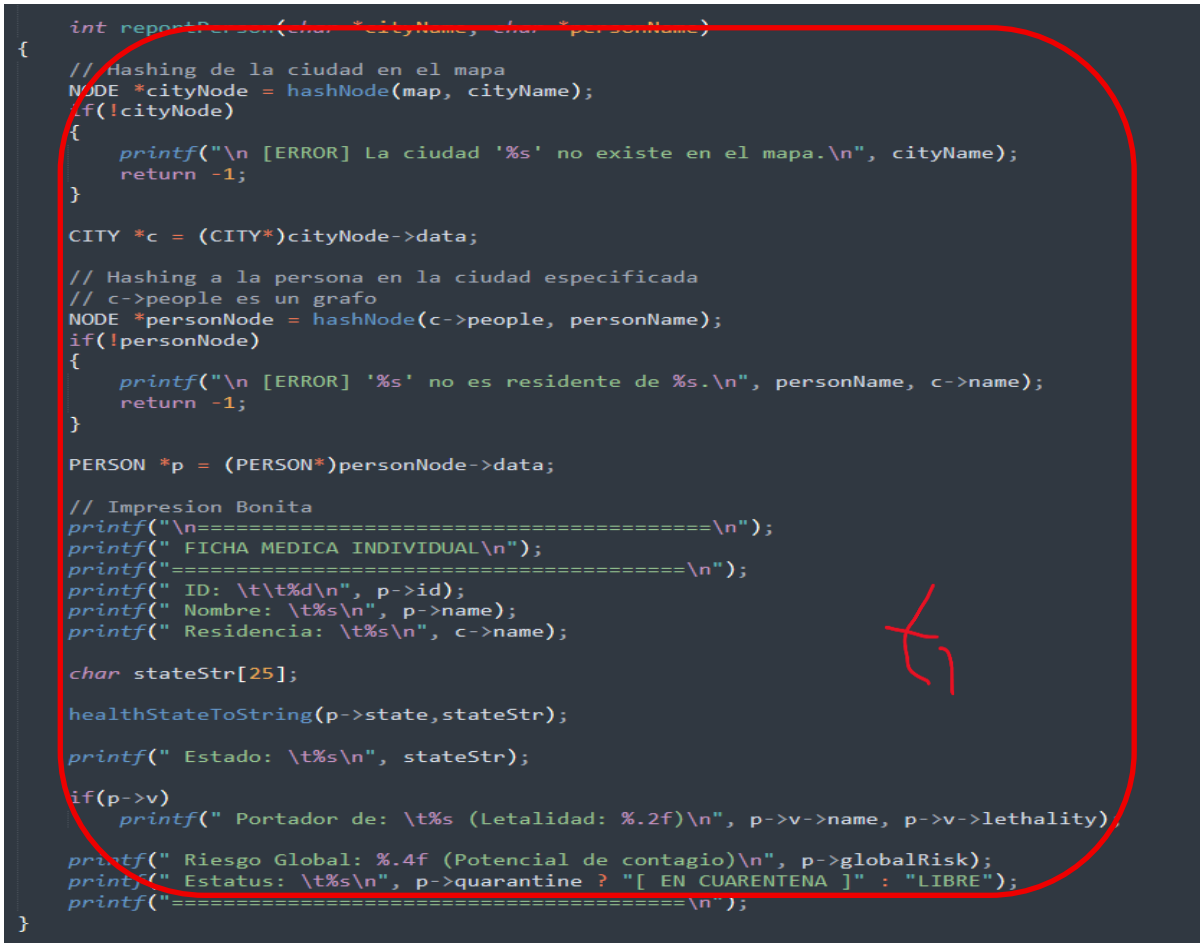
Hashing anidado

El este módulo utilizamos el diseño arquitectónico de la base de datos en la memoria para garantizar consultas de tiempo constante.

Justificación

Tenemos una eficiencia constante gracias a la base de datos de la memoria, tenemos las consultas individuales ‘reportPerson’ la función comienza buscando la ciudad en el grafo global utilizando el nombre de la ciudad como clave ‘hashNode(map, cityName)), una vez encontrado el nodo de la ciudad, accedemos a su grafo interno para buscar a la persona por su clave ‘hashNode(c->people, personName)’ por estos beneficios es que utilizamos hash por su nivel de complejidad en comparación con otros algoritmos como árbol binario de búsqueda que tiene una complejidad de O(log n) no era conveniente ni lo que se solicita en el trabajo.

Cálculo de eficiencia y complejidad

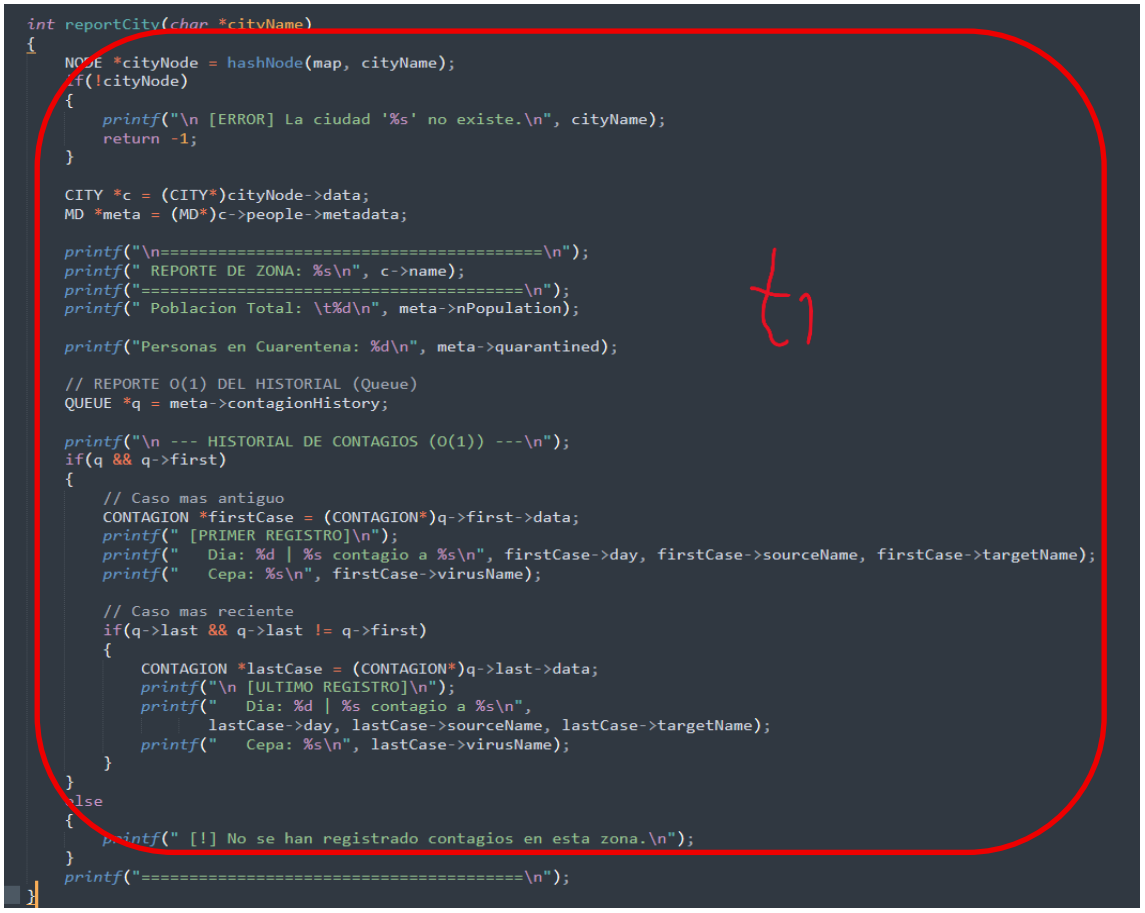


```
int reportPerson(char *cityName, char *personName)
{
    // Hashing de la ciudad en el mapa
    NODE *cityNode = hashNode(map, cityName);
    if(!cityNode)
    {
        printf("\n [ERROR] La ciudad '%s' no existe en el mapa.\n", cityName);
        return -1;
    }

    CITY *c = (CITY*)cityNode->data;

    // Hashing a la persona en la ciudad especificada
    // c->people es un grafo
    NODE *personNode = hashNode(c->people, personName);
    if(!personNode)
    {
        printf("\n [ERROR] '%s' no es residente de %s.\n", personName, c->name);
        return -1;
    }

    PERSON *p = (PERSON*)personNode->data;

    // Impresion Bonita
    printf("\n========================================\n");
    printf(" FICHA MEDICA INDIVIDUAL\n");
    printf("========================================\n");
    printf(" ID: \t\t%d\n", p->id);
    printf(" Nombre: \t%s\n", p->name);
    printf(" Residencia: \t%s\n", c->name);

    char stateStr[25];

    healthStateToString(p->state,stateStr);

    printf(" Estado: \t%s\n", stateStr);

    if(p->v)
        printf(" Portador de: \t%s (Letalidad: %.2f)\n", p->v->name, p->v->lethality);

    printf(" Riesgo Global: %.4f (Potencial de contagio)\n", p->globalRisk);
    printf(" Estatus: \t%s\n", p->quarantine ? "[ EN CUARENTENA ]" : "LIBRE");
    printf("========================================\n");
}
```

$T(n) = t1$

$$\lim_{n\to\infty} \frac{T(n)}{1} = \lim_{n\to\infty} \frac{1}{1} = 1$$

$T(1) \in O(1)$



```c
int reportCity(char *cityName)
{
    NODE *cityNode = hashNode(map, cityName);
    if(!cityNode)
    {
        printf("\n [ERROR] La ciudad '%s' no existe.\n", cityName);
        return -1;
    }

    CITY *c = (CITY*)cityNode->data;
    MD *meta = (MD*)c->people->metadata;

    printf("\n=======================================\n");
    printf(" REPORTE DE ZONA: %s\n", c->name);
    printf("=======================================\n");
    printf(" Poblacion Total: \t%d\n", meta->nPopulation);

    printf("Personas en Cuarentena: %d\n", meta->quarantined);

    // REPORTE O(1) DEL HISTORIAL (Queue)
    QUEUE *q = meta->contagionHistory;

    printf("\n --- HISTORIAL DE CONTAGIOS (O(1)) ---\n");
    if(q && q->first)
    {
        // Caso mas antiguo
        CONTAGION *firstCase = (CONTAGION*)q->first->data;
        printf(" [PRIMER REGISTRO]\n");
        printf("   Dia: %d | %s contagio a %s\n", firstCase->day, firstCase->sourceName, firstCase->targetName);
        printf("   Cepa: %s\n", firstCase->virusName);

        // Caso mas reciente
        if(q->last && q->last != q->first)
        {
            CONTAGION *lastCase = (CONTAGION*)q->last->data;
            printf("\n [ULTIMO REGISTRO]\n");
            printf("   Dia: %d | %s contagio a %s\n",
                   lastCase->day, lastCase->sourceName, lastCase->targetName);
            printf("   Cepa: %s\n", lastCase->virusName);
        }
    }
    else
    {
        printf(" [!] No se han registrado contagios en esta zona.\n");
    }
    printf("=======================================\n");
}
```

t1

$T(n) = t1$

$$\lim_{n\to\infty} \frac{T(n)}{1} = \lim_{n\to\infty} \frac{1}{1} = 1$$

$T(1) \in O(1)$

➔ Complejidad general de O (1)