Proyecto 1-Entrega 03

ESTRUCTURAS DE DATOS

Estudiantes:

Laura Valentina Ovalle Benítez

Valentina García Alfonso

Juan Miguel Zuluaga Suárez

Carlos Gabriel López

Nicolás Padilla Medina

PRESENTADO A:

John Jairo Corredor Franco

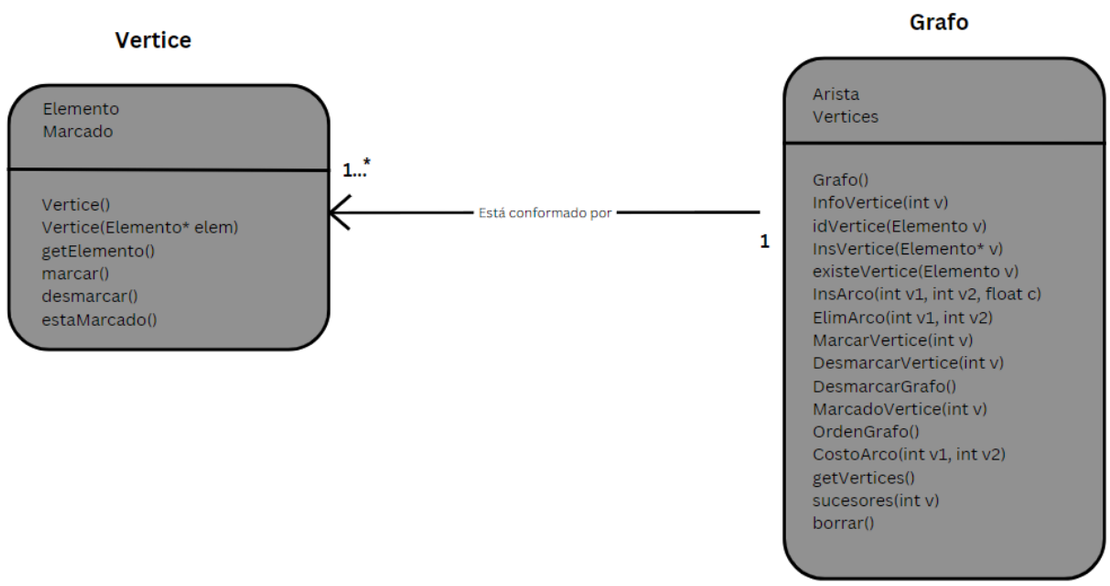
PONTIFICIA UNIVERSIDAD JAVERIANA

BOGOTÁ D.C

2023

Declaración de TAD's pertenecientes al segundo Componente

A continuación se presentan los TAD's que fueron añadidos al proyecto con el fin de cumplir con las funcionalidades propuestas en el "Componente 3":



Vértice

1. **Datos mínimos**
    a. elemento, apuntador de tipo "elemento", que corresponde al elemento que contiene el vértice del grafo
    b. marcado, booleano que indica si el vértice se encuentra marado o no
2. **Operaciones**
    a. Vertice(), constructor por defecto del vértice, no realiza ninguna operación
    b. Vertice(Elemento* elem), constructor sobrecargado que recibe como parámetro un elemento y lo asigna al atributo "elemento" del vértice.
    c. getElemento(), retorna el atributo de tipo elemento que se encuentra almacenado en el vértice
    d. marcar(), asigna un valor de verdadero para el atributo "marcado" del vértice
    e. desmarcar(), asigna un valor de faso para el atributo "marcado" del vértice
    f. estaMarcado(), retorna el atributo "marcado" del vértice

Grafo

1. **Datos mínimos**
    a. Arista, lista de tipo "pair" (de dos elementos) que contiene en la primera posición el coste o distancia del vértice actual con su vértice vecino, y en la segunda posición el id del vértice vecino al cual corresponde la distancia antes mencionada
    b. vertices, lista de vértices que contiene cada uno de los vértices del grafo
2. **Operaciones**
    a. Grafo(), constructor por defecto del grafo, no realiza ninguna operación
    b. InfoVertice(int v), recibe un entero que corresponde al ID de un vértice, lo busca en la lista de vértices, y una vez lo encuentra retorna el atributo de tipo "elemento" que se encuentra almacenado actualmente en el vértice
    c. idVertice(Elemento v), recibe un parámetro de tipo "Elemento" y busca en la lista de vértices el vértice que contenga este elemento con la finalidad de retornar su ID
    d. InsVertice(Elemento* v), crea un nuevo vértice a partir del parámetro de tipo "elemento" que recibe, es decir, llama al constructor sobrecargado del vértice. Una vez el vétice está creado lo inserta en la lista de vértices
    e. existeVertice(Elemento v), recibe un parámetro de tipo "Elemento" y busca en la lista de vértices para comprobar si ya existe alguno que contenga este elemento, en caso de que sí se retorna un valo de "verdadero", en caso contrario se retorna un valor de "falso"

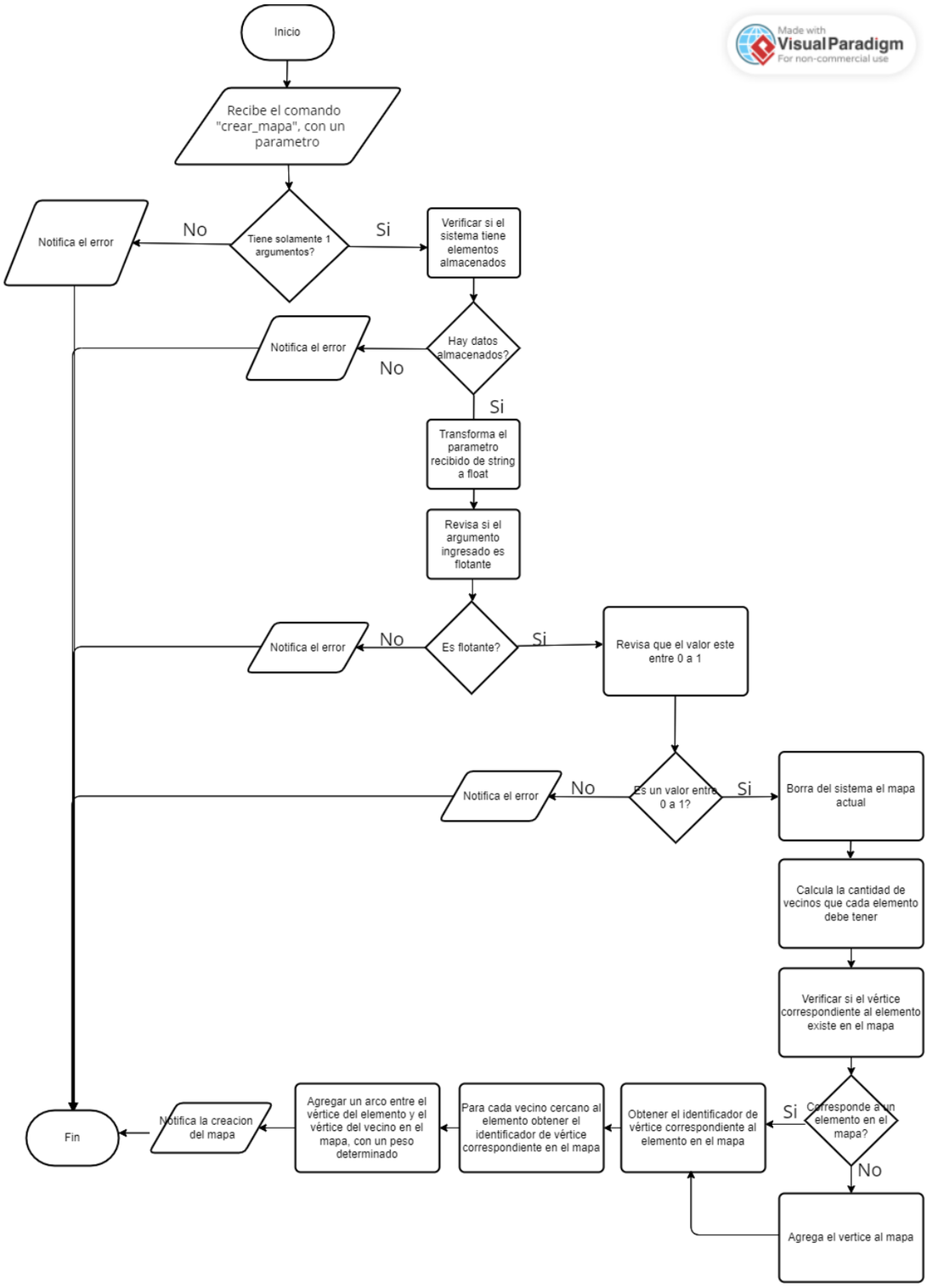
f. InsArco(int v1, int v2, float c), crea un nuevo elemento de tipo “Arista” buscando en la lista de aristas hasta encontrar la posición que corresponde al parámetro “v1”, y allí inserta el parámetro “v2” en la primera posición y “c” en la segunda, siendo respectivamente el vértice de destino y la distancia del vértice de origen al de destino.
g. ElimArco(int v1, int v2), elimina la arista que conecta al vértice con ID “v1” y el vértice con ID “v2”
h. MarcarVertice(int v), llama a la función “marcar()” del vértice que posea el ID que recibe como parámetro con el nombre de “v”, con la finalidad de que su atributo “marcado” ahora tenga un valor de verdadero
i. DesmarcarVertice(int v) llama a la función “desmarcar()” del vértice y se utiliza para desmarcar un vértice específico en el grafo.
j. DesmarcarGrafo(), se utiliza para desmarcar todos los vértices del grafo.
k. MarcadoVertice(int v), es un método de la clase Grafo que se utiliza para verificar si un vértice específico está marcado en el grafo.
l. OrdenGrafo(), es un método constante de la clase Grafo que devuelve el número de vértices en el grafo.
m. CostoArco(int v1, int v2), se utiliza para determinar el costo de un arco entre dos vértices dentro del grafo.
n. getVertices(), devuelve una lista de punteros a elementos (Elemento*), representando los elementos asociados a cada vértice del grafo.
o. sucesores(int v), devuelve una lista de enteros que representa los sucesores de un vértice específico en el grafo.
p. borrar(), se encarga de eliminar todas las aristas y vértices existentes en el grafo, dejándolo vacío.

Diagramas de Flujo

1. **Crear Mapa**

Nombre:	comando **crear_mapa**
Objetivo en Contexto	El objetivo de este comando es utilizar la información de puntos de interés almacenada en memoria para ubicarlos en un grafo teniendo en cuenta el coeficiente de conectividad dado. Se busca que cada elemento esté conectado a los demás elementos más cercanos a él, utilizando la distancia euclidiana como medida de cercanía y peso de la conexión. El comando tiene como objetivo generar un mapa donde cada elemento tenga n vecinos, determinado por el coeficiente de conectividad especificado.
ENTRADAS	Se recibe el comando de crear_mapa y el coeficiente de conectividad que se usara
Pre-Condiciones *(Escenario de éxito)*	El comando “crear_mapa” debe tener exactamente dos palabras, donde la primera debe ser “crear_mapa” y la segunda debe ser el valor del coeficiente de conectividad.
SALIDAS	<u>Si el comando es correcto:</u> Si hay elementos cargados, y se ingresaron los datos de forma correcta, la funcion crearia el mapa e imprimiria”El mapa se ha generado exitosamente. Cada elemento tiene " << vecinos << " vecinos.” <u>Si el comando es incorrecto(Se le pasa un numero diferente de un parametro):</u> Se termina el programa y se muestra en pantalla: “.Se requiere el coeficiente conectividad" <u>Si no hay elementos:</u> Se termina el programa y se muestra en pantalla: “La informacion requerida no esta almacenada en memoria." <u>Error con el coeficiente de conectividad:</u> Sucede cuando el coeficiente de conectividad no es un valor entre 0 y 1, y imprime:”El coeficiente de conectividad debe tener un valor entre 0 y 1.” <u>Si el argumento no es flotante:</u> Se imprime en pantalla “El coeficiente de conectividad debe tener un valor entre 0 y 1”
Post-Condiciones	Se imprime que la funcion se ejecuto correctamente, y se crea el mapa de elementos

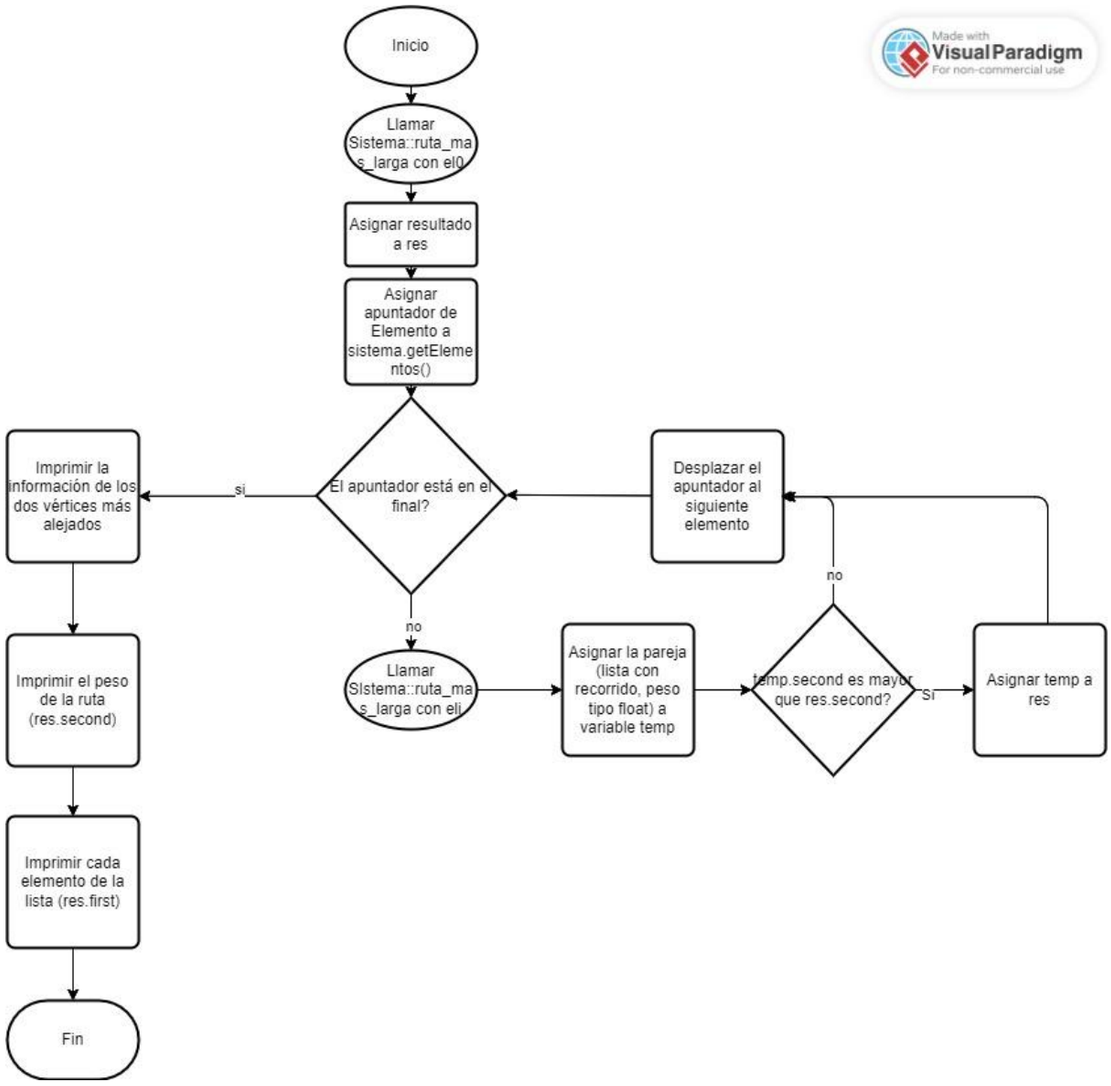
(Escenario de éxito)	



2. Ruta más larga

Nombre:	comando **ruta_mas_larga**
Objetivo en Contexto	El objetivo de este comando es encontrar la ruta más larga existente dentro de un grafo. Esta, siendo la ruta entre dos vértices, que pase por otros k vértices y que el peso de esas k+1 aristas, sea el mayor posible.

ENTRADAS	Se recibe el comando de ruta_mas_larga
Pre-Condiciones *(Escenario de éxito)*	El comando “ruta_mas_larga” requiere que el comando “crear_mapa” haya sido ejecutado con éxito.
SALIDAS	<u>Si el comando es correcto:</u> En una situación adecuada, el programa imprime los siguientes resultados en consola: “Los puntos de interés más alejados entre si son: <<punto 1>> y <<punto 2>>.” “La ruta que los conecta tiene una longitud total de <<longitud>> y pasa por los siguientes elementos:” “<<Lista de elementos que recorre>>”” <u>Si el comando recibe algún parámetro:</u> Si el comando es recibido con algún parámetro extra (diferente del nombre del comando” el programa arroja una excepción de tiempo de ejecución e imprime: “No se requieren argumentos” <u>Si el comando “crear_mapa” no ha sido ejecutado con éxito:</u> Si el programa detecta que el mapa está vacío, por ende no ha sido ejecutado el comando crear_mapa, arroja una excepción de tiempo de ejecución e imprime: “El mapa no ha sido generado todavía (con el comando crear_mapa)” <u>Si algún elemento recibido no existe en el mapa:</u> Si el programa detecta que alguno de los vértices no está incluído en el mapa, arroja una excepción de tiempo de ejecución e imprime: “El elemento no se encuentra en el mapa”
Post-Condiciones *(Escenario de éxito)*	Se imprime: “Los puntos de interés más alejados entre si son: <<punto 1>> y <<punto 2>>.” “La ruta que los conecta tiene una longitud total de <<longitud>> y pasa por los siguientes elementos:” “<<Lista de elementos que recorre>>”” No se retorna nada ni se altera espacio en memoria


Inicio
Llamar Sistema::ruta_mas_larga con el0
Asignar resultado a res
Asignar apuntador de Elemento a sistema.getElementos()
El apuntador está en el final?
si
Imprimir la información de los dos vértices más alejados
Imprimir el peso de la ruta (res.second)
Imprimir cada elemento de la lista (res.first)
Fin
no
Llamar SIstema::ruta_mas_larga con eli
Asignar la pareja (lista con recorrido, peso tipo float) a variable temp
temp.second es mayor que res.second?
Si
Asignar temp a res
no
Desplazar el apuntador al siguiente elemento
Made with VisualParadigm
For non-commercial use

3. Plan de pruebas

Con el fin de realizar el plan de pruebas de las nuevas funciones implementadas en el proyecto (“crear_mapa” y “ruta_mas_larga”), se cargarán 5 elementos del archivo de prueba ‘elementosp.txt’ para verificar el correcto funcionamiento del código. Cabe resaltar que los 5 elementos son insertados usando el siguiente formato:

Tipo de elemento|Tamaño|Distancia|CentroX|CentroY

A continuación se listan los elementos que se encuentran en el archivo “elementosp.txt”:

crater|1|metros|2.0|2.0
crater|1|metros|3.0|3.0
monticulo|1|metros|6.0|8.0
crater|2.5|metros|12.50|7.5
duna|2|metros|18.00|14.00

A continuación se listan los coeficientes que se utilizarán en conjunto con el comando “crear_mapa”:

- crear_mapa 0.1
- crear_mapa 1
- crear_mapa -1
- crear_mapa 0.4

Tabla de prueba:

Descripción del caso	Valores de entrada	Resultado Esperado	Resultado Obtenido
Sin crear Mapa	ninguno	figura 1	Exitoso
Coeficiente 0.1	0.1	figura 2	Exitoso
Coeficiente 1	1	figura 3	Exitoso
Coeficiente -1	-1	figura 4	Exitoso
Coeficiente 0.4	0.4	figura 5	Exitoso

Finalmente, se presenta el plan de pruebas para cada uno de los coeficientes

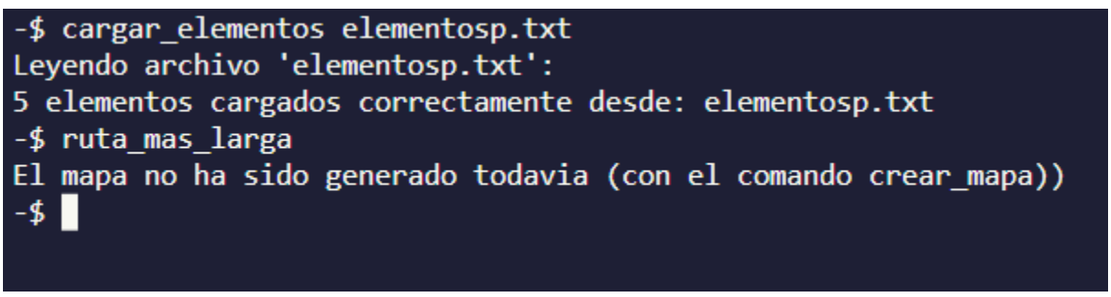


Figura 1. Prueba sin crear Mapa

```
-$ cargar_elementos elementosp.txt
Leyendo archivo 'elementosp.txt':
5 elementos cargados correctamente desde: elementosp.txt
-$ crear_mapa 0.1
El mapa se ha generado exitosamente. Cada elemento tiene 1 vecinos.
-$ ruta_mas_larga
Los puntos de interes mas alejados entre si son: 'duna 2 metros 18 14' y 'crater 1 metros 2 2'
La ruta que los conecta tiene una longitud total de 22.5475 y pasa por los siguientes elementos:
'duna 2 metros 18 14'
'crater 2 metros 12 7'
'monticulo 1 metros 6 8'
'crater 1 metros 3 3'
'crater 1 metros 2 2'
-$
```

Figura 2. Prueba coeficiente 0.1

```
-$ cargar_elementos elementosp.txt
Leyendo archivo 'elementosp.txt':
5 elementos cargados correctamente desde: elementosp.txt
-$ crear_mapa 1
El mapa se ha generado exitosamente. Cada elemento tiene 4 vecinos.
-$ ruta_mas_larga
Los puntos de interes mas alejados entre si son: 'crater 2 metros 12 7' y 'crater 1 metros 3 3'
La ruta que los conecta tiene una longitud total de 51.8949 y pasa por los siguientes elementos:
'crater 2 metros 12 7'
'monticulo 1 metros 6 8'
'crater 1 metros 2 2'
'duna 2 metros 18 14'
'crater 1 metros 3 3'
-$
```

Figura 3. Prueba coeficiente 1

```
-$ cargar_elementos elementosp.txt
Leyendo archivo 'elementosp.txt':
5 elementos cargados correctamente desde: elementosp.txt
-$ crear_mapa -1
El coeficiente de conectividad debe tener un valor entre 0 y 1
-$
```

Figura 4. Prueba coeficiente -1

```
Leyendo archivo 'elementosp.txt':
5 elementos cargados correctamente desde: elementosp.txt
-$ crear_mapa 0.4
El mapa se ha generado exitosamente. Cada elemento tiene 2 vecinos.
-$ ruta_mas_larga
Los puntos de interes mas alejados entre si son: 'crater 1 metros 3 3' y 'duna 2 metros 18 14'
La ruta que los conecta tiene una longitud total de 23.9276 y pasa por los siguientes elementos:
'crater 1 metros 3 3'
'crater 1 metros 2 2'
'monticulo 1 metros 6 8'
'crater 2 metros 12 7'
'duna 2 metros 18 14'
-$
```

Figura 5. Prueba coeficiente 0.4