El vehículo “Curiosity”: Implementando Estructuras de Datos.

De Aza M. Tomás, Galindo M. Sofía

I. RESUMEN

El artículo discute el vehículo explorador "Curiosity" de la NASA, un robot que realiza experimentos científicos en Marte para buscar condiciones pasadas o presentes que sean favorables para la vida y puedan conservar registros de vida. La movilidad del explorador es esencial para su misión, y se mueve a través del terreno marciano utilizando secuencias de comandos enviadas desde la Tierra o de manera autónoma utilizando mapas de profundidad generados por cámaras en su frente. Una mala planificación de la trayectoria puede poner en riesgo la misión, por lo que se utiliza un mecanismo de retroalimentación y validación basado en fotografías tomadas por las cámaras del vehículo y análisis del suelo marciano.

El artículo también describe un proyecto para construir un sistema que simule las actividades e interacciones entre el "Curiosity" y su centro de control de misión en la NASA. Los tipos de datos enviados al robot para su misión incluyen comandos de desplazamiento, que le permiten moverse y analizar el suelo marciano, y datos de puntos de interés, que incluyen ubicaciones geográficas de componentes o elementos encontrados en el terreno.

II. ANÁLISIS DEL PROBLEMA

El problema en cuestión involucra el desplazamiento del vehículo explorador "Curiosity" en Marte y su interacción con el centro de control de misión en la NASA. A continuación se presenta un análisis del problema teniendo en cuenta el uso de listas, estructuras lineales, grafos y quad trees.

1. Listas y Estructuras Lineales:

Las listas y otras estructuras lineales, como pilas y colas, pueden utilizarse para gestionar las secuencias de comandos enviadas desde la Tierra a "Curiosity". Cada comando puede ser almacenado como un elemento en la lista o estructura lineal, y el robot puede procesar los comandos uno por uno en orden. También se podrían usar listas para almacenar las fotos tomadas por "Curiosity" y para manejar los datos de los puntos de interés encontrados en el terreno marciano.

2. Grafos:

Los grafos pueden ser útiles para representar el terreno marciano y para planificar las rutas que "Curiosity" debe seguir. Los nodos del grafo podrían representar ubicaciones geográficas específicas en Marte, y las aristas podrían representar los posibles caminos entre estas ubicaciones. Los pesos de las aristas podrían representar la dificultad o el tiempo requerido para moverse entre dos ubicaciones. Este enfoque podría ayudar a "Curiosity" a planificar la ruta más eficiente para cumplir su misión, o a encontrar rutas alternativas en caso de que el camino inicialmente planeado sea intransitable.

3. Quad Trees:

Los Quad Trees son estructuras de datos de árbol utilizadas comúnmente para manejar datos espaciales bidimensionales. En este caso, podrían utilizarse para representar los mapas de profundidad que "Curiosity" genera con sus cámaras. Cada nodo en el Quad Tree podría representar una región específica del mapa, y los cuatro hijos de cada nodo podrían representar los cuatro cuadrantes de esa región. Esto permitiría a "Curiosity" buscar y analizar eficientemente las áreas de interés en sus mapas de profundidad

III. MANEJO DE LA INFORMACIÓN

Estamos proponiendo un sistema avanzado que simulará la interacción y las operaciones del vehículo "Curiosity" de la NASA con su centro de control de misión. Para el éxito del proyecto, se ha definido una serie de comandos, cada uno de los cuales se explica a continuación:

- **cargar_comandos nombre_archivo:** Este comando permite al sistema leer y cargar correctamente los comandos de desplazamiento de un archivo específico.
- **cargar_elementos nombre_archivo:** Este comando habilita al sistema para leer y cargar correctamente los datos de los puntos de interés contenidos en un archivo específico.
- **agregar_movimiento tipo_mov** magnitud unidad_med: A través de este comando, el sistema puede agregar un nuevo comando de movimiento a la lista de comandos de "Curiosity".
- **agregar_analisis tipo_analisis objeto** Este comando permite al sistema agregar un nuevo comando de análisis a la lista de comandos de "Curiosity".
- **agregar_elemento tipo_comp tamaño unidad_med coordX coordY** Este comando permite al sistema agregar un nuevo punto de interés a la lista de "Curiosity".
- **guardar tipo_archivo nombre_archivo** Este comando permite al sistema guardar la información solicitada en un archivo específico.
- **simular_comandos coordX coordY** Este comando permite al sistema simular los comandos de movimiento a partir de la posición actual del vehículo "Curiosity".

Además, se ha definido un tipo abstracto de datos (TAD) Comandos que incluye los datos y operaciones mínimas necesarias para el componente. Los detalles del TAD Comandos se proporcionan a continuación:

Datos mínimos para todo el componente: nombre_archivo, tipo_mov, magnitud, unidad_med, tipo_analisis, objeto, comentario, tipo_comp, tamaño, unidad_med, coordX, coordY, tipo_archivo, coeficiente_conectividad, coordX1, coordX2, coordY1, coordY2.

Se manipularon las siguientes estructuras de datos:

- **sComandos:** Es una estructura que representa un comando en el sistema. Tiene dos constructores: uno para comandos de movimiento y otro para comandos de análisis. Los comandos de movimiento tienen un tipo de movimiento, una magnitud y una unidad de medida. Los comandos de análisis tienen un tipo de análisis, un objeto y un comentario. También tiene un constructor por defecto.
- **sCuriosity:** Es una estructura que representa el rover Curiosity. Tiene dos constructores: uno por defecto y otro que recibe una lista de comandos y una lista de elementos. El rover tiene una orientación y puede ejecutar una serie de comandos y mantener una lista de elementos.
- **sElemento:** Es una estructura que representa un elemento en el sistema. Esta estructura no está definida en el código proporcionado, pero se asume que se utiliza en otros lugares del programa para representar elementos específicos.

Implementación en código:

- **bool hasMoreTokens(std::stringstream& ss, char delimiter) :** Verifica si hay más tokens en un stringstream basado en un delimitador.
- **int countLines(std::string fileName) :** Cuenta el número de líneas en un archivo.
- **void escribir_archivo_comandos(std::string filename, std::liststd::string internal)** : Escribe los elementos de una lista en un archivo de comandos.
- **void escribir_archivo_elementos(std::string filename, std::liststd::string internal)** : Escribe los elementos de una lista en un archivo de elementos.
- **std::vectorstd::string TokenizarEntradaUsuario (std::string entrada) :** Tokeniza la entrada del usuario en un vector de strings.
- **std::vector<Point> puntoMaxyMin(std::list<sElemento> internal) :** Encuentra los puntos máximos y mínimos en términos de coordenadas x e y en una lista de elementos.

IV. Planeación de Desplazamientos

Para el segundo componente del sistema, nuestro objetivo es usar una estructura de datos jerárquica que almacene datos geométricos para analizar los puntos de interés en el cielo marciano. Así, facilitaremos la futura planificación automática de desplazamientos. Los comandos que se implementarán como parte de este componente son:

- **ubicar_elementos:** Este comando utiliza la información de los puntos de interés almacenada en memoria para colocarlos en una estructura de datos jerárquica adecuada, que permitirá realizar consultas geográficas sobre estos elementos.
- **en_cuadrante coordX1 coordX2 coordY1 coordY2:** Este comando utiliza la estructura creada con el comando anterior para retornar una lista de los componentes o elementos que están dentro del cuadrante geográfico descrito por los límites de coordenadas en x y y. Para poder realizar la búsqueda por cuadrantes, es necesario haber ejecutado el comando ubicar_elementos.

Para implementar estos comandos, utilizaremos la estructura de datos conocida como Quadtree. Un Quadtree es una estructura de árbol en la cual cada nodo tiene exactamente cuatro hijos: noroeste, noreste, suroeste y sureste. Los Quadtrees son la estructura de datos espacial preferida para consultas de rango en dos dimensiones.

Los detalles de los comandos son los siguientes:

- **ubicar_elementos:** Este comando utiliza la información de los puntos de interés almacenada en memoria para ubicarlos en una estructura de datos jerárquica adecuada, que permita realizar consultas geográficas sobre estos elementos. En caso de que alguno de los elementos no pueda ser agregado adecuadamente, el sistema generará un mensaje de error, pero continuará procesando el resto de los elementos almacenados en memoria.
- **en_cuadrante coordX1 coordX2 coordY1 coordY2:** Este comando permite utilizar la estructura creada con el comando anterior para devolver una lista de los elementos que se encuentran dentro del cuadrante geográfico descrito por los límites de coordenadas en x y y. Es necesario haber ejecutado el comando ubicar_elementos para poder realizar la búsqueda por cuadrantes. Los límites de coordenadas deben garantizar que coordX1<coordX2 y coordY1<coordY2.

- **Los datos mínimos para todo el componente incluyen:** coordX1, coordX2, coordY1, coordY2.
- **Las operaciones mínimas para todo el componente incluyen:** ubicar_elementos, en_cuadrante.

Funciones generales del quadtree:

- **void split()**: Método privado de la clase Quadtree que se utiliza para dividir el nodo en cuatro hijos. Calcula las coordenadas y crea los cuatro hijos del nodo actual.
-

- **Quadtree(int level, Point topLeft, Point bottomRight):** Constructor de la clase Quadtree que recibe el nivel del árbol, las coordenadas del punto superior izquierdo y las coordenadas del punto inferior derecho. Inicializa los miembros de datos del nodo y establece los punteros a los hijos en nullptr.
- **void insert(sElemento point):** Método para insertar un punto en el árbol. Si el nodo actual ya tiene hijos, el punto se inserta en el hijo correspondiente. Si el número de puntos en el nodo supera el límite máximo y la profundidad del árbol no ha alcanzado el nivel máximo, se realiza la división del nodo y se reinsertan los puntos en los hijos correspondientes.
- **std::vector<sElemento> query(float x1, float y1, float x2, float y2, Quadtree* pQuadtree):** Función para buscar los elementos dentro de un área específica definida por las coordenadas x1, y1, x2 y y2. Se realiza una verificación para determinar si el área de búsqueda está completamente fuera del área del nodo actual. Luego, se verifica si los puntos en el nodo actual están dentro del área de búsqueda y se agregan al resultado. Si el nodo actual tiene hijos, se realiza una búsqueda recursiva en cada uno de ellos y se agrega el resultado al vector result.
- **bool esQuadTreeVacio(Quadtree* nodo):** Función para comprobar si un Quadtree está vacío. Verifica si el nodo y todos sus hijos son nulos.
- **int getIndex(sElemento point):** Método privado para determinar en qué hijo se encuentra un punto. Utiliza las coordenadas del punto y el punto medio vertical y horizontal del nodo para determinar el cuadrante en el que se encuentra el punto.
- **void imprimirQuadtree(Quadtree* pQuadtree):** Método para imprimir el contenido del Quadtree en el que se llama. Imprime los puntos almacenados en el nodo actual y luego realiza una llamada recursiva para imprimir los puntos de sus hijos.

V. Recorridos Entre Puntos de Interes

En este tercer componente, se busca utilizar la información de puntos de interés para crear un mapa basado en la representación de grafos. Para esto se desarrollarán dos comandos principales.

Los detalles de los comandos son:

- **crear_mapa coeficiente_conectividad:** Este comando utiliza la información de puntos de interés almacenada en memoria para ubicarlos en una estructura no lineal y conectarlos entre sí teniendo en cuenta el coeficiente de conectividad dado. Este coeficiente de conectividad representa la cantidad de vecinos que puede tener cada punto de interés, medido como una fracción del total de elementos en el mapa. El objetivo es que cada punto esté conectado a los puntos más cercanos a él, medidos a través de la distancia euclidiana. Esta distancia también se utiliza como el peso o etiqueta de la conexión entre los puntos. Si el mapa se genera con éxito, se muestra un mensaje de éxito. De lo contrario, se indica que no hay información almacenada en memoria.
- **ruta_mas_larga:** Este comando, una vez creado el mapa, permite identificar los dos puntos más alejados entre sí de acuerdo a las conexiones generadas. El comando retorna los elementos más alejados de acuerdo a las conexiones que se encuentran en el mapa, no los elementos que estén a mayor distancia euclidiana entre sí. Al encontrar esa ruta más larga dentro del mapa, el comando imprime en pantalla los elementos de origen y destino, la longitud total de la ruta, y la secuencia de elementos que hay que seguir para ir del elemento origen al elemento destino. Si no se ha generado el mapa previamente, se mostrará un mensaje indicando que el mapa no ha sido generado todavía.

- **Los datos mínimos para todo el componente incluyen:** coeficiente_conectividad.
- **Las operaciones mínimas para todo el componente incluyen:** crear_mapa, ruta_mas_larga.

Implementacion del Código:

- **Grafo::Grafo(float coeficiente) :** Constructor de la clase Grafo que recibe un coeficiente y asigna valores iniciales a los atributos.
- **void Grafo::insertarNodo(std::vector<sElemento> elemento) :** Inserta nodos en el grafo a partir de un vector de elementos.
- **float Grafo::DistanciaEuclidiana(Nodo* elemento1, Nodo* elemento2) :** Calcula la distancia euclidiana entre dos nodos del grafo.
- **void Grafo::Conectar() :** Establece las conexiones entre los nodos del grafo basado en un coeficiente y una fórmula de distancia.
- **void Grafo::imprimir() :** Imprime en consola los nodos del grafo y sus conexiones adyacentes.
- **void Grafo::DFS(Nodo* nodoActual, std::unordered_map<Nodo*, bool>& visitado, std::vector<Nodo*>& rutaActual, float pesoTotal, float& pesoMaximo, std::vector<Nodo*>& rutaMasLarga) :** Realiza una búsqueda en profundidad en el grafo para encontrar la ruta más larga entre dos nodos.
- **void Grafo::ruta_mas_larga() :** Encuentra la ruta más larga entre dos nodos del grafo y la imprime en consola, junto con los puntos de interés más alejados en términos de aristas

IX. . Salidas

En esta sección, exploraremos el funcionamiento detallado de los diversos comandos y funciones implementados en nuestro

programa. El propósito de esta exploración es proporcionar una comprensión clara de cómo cada comando interactúa con los datos y cómo estos comandos en conjunto permiten la manipulación eficaz de las estructuras de datos.

Uno de los aspectos críticos a tener en cuenta es que el resultado de muchos de estos comandos depende en gran medida de los datos específicos ingresados y ciertos parámetros asociados. Por lo tanto, la simulación que se presentará a continuación hará suposiciones sobre estos datos y parámetros para ilustrar el funcionamiento de las funciones

Estructura Comandos

```
0|avanzar|100|metros
0|girar|90|grados
0|avanzar|100|metros
0|girar|90|grados
0|avanzar|100|metros
0|girar|90|grados
0|avanzar|100|metros
```

Estructura Elementos

```
1 roca|0.25|metros|45|20
2 crater|0.3|metros|100|20
3 crater|2|metros|1|20
4 duna|0.5|metros|5|30
5 roca|0.7|metros|65|73
6 monticulo|1|metros|23|35
7 crater|4|metros|84|23
8 duna|1|metros|47|39
9 crater|4|metros|29|80
```

1ra Entrega Simulación de Comandos

```
$cargar_comandos Comandos.txt
7 Comandos cargados
$simular_comandos 0 0
Coordenadas actuales: 100 0
Orientacion actual: 90
Coordenadas actuales: 100 100
Orientacion actual: 180
Coordenadas actuales: 0 100
Orientacion actual: 270
Coordenadas actuales: 0 0
```

2da Entrega Procesamiento de Elementos

```
$cargar_elementos Elemento.txt
Los elementos han sido procesados exitosamente:
roca 0.25 metros 45 20
crater 0.3 metros 100 20
crater 2 metros 1 20
duna 0.5 metros 5 30
roca 0.7 metros 65 73
monticulo 1 metros 23 35
crater 4 metros 84 23
duna 1 metros 47 39
crater 4 metros 29 80
$en_cuadrante 0 0 100 100
roca 0.25 metros 45 20
crater 0.3 metros 100 20
crater 2 metros 1 20
duna 0.5 metros 5 30
roca 0.7 metros 65 73
monticulo 1 metros 23 35
crater 4 metros 84 23
duna 1 metros 47 39
crater 4 metros 29 80
```

3ra Entrega Ruta Más Larga

```
$ruta_mas_larga
(Resultado exitoso) Los puntos de interes mas alejados entre si en terminos de aristas son (29, 80) y (47, 39).
La ruta que los conecta tiene una longitud total de 8 aristas y pasa por los siguientes elementos:
(29, 80) (5, 30) (45, 20) (84, 23) (100, 20) (65, 73) (23, 35) (1, 20) (47, 39)
```

X. CONCLUSIÓN

En el contexto de solución al problema planteado, fue esencial la aplicación de diversas estructuras de datos y algoritmos eficientes, tales como estructuras lineales, grafos y quadtrees. Estas herramientas permitieron la representación, gestión y manipulación eficaz de los datos y relaciones espaciales asociados a la navegación de los robots en la superficie de Marte.

En términos de la organización y manipulación de datos, las estructuras lineales proporcionan una base sólida y eficiente para el almacenamiento y recuperación de datos. Esta eficiencia resulta crucial para manejar la información de la misión, manteniendo la coherencia y fiabilidad de los datos.

Por su parte, el uso de grafos fue de vital importancia para representar y manejar la información espacial y las relaciones entre diferentes puntos de interés. La capacidad de representar los puntos de interés como nodos en un grafo y las relaciones entre ellos como aristas permite modelar y manipular complejas relaciones espaciales de una manera intuitiva y eficiente.

Finalmente, la aplicación de quadtrees proporcionó una representación espacial eficiente que facilitó la búsqueda y la gestión de los puntos de interés en el espacio bidimensional.

En conclusión, la implementación efectiva de estas tres estructuras de datos y los algoritmos asociados a ellas, permitió la creación de una solución robusta para el problema propuesta en clase.