TALLER N°1

RUSH HOUR SOLUCION

usach

JAIME RIQUELME OLGUIN

TALLER DE PROGRAMACIÓN | 02 OCTUBRE 2023

TALLER N°1

Explicación breve del algoritmo:

El algoritmo es básicamente un solucionador el juego “Rush Hour”, utilizando una matriz de 6x6 para representar la situación de los autos en un estacionamiento. Cada auto tiene su propia información, como su posición, en qué dirección se está moviendo y cuánto espacio ocupa. El algoritmo utiliza "estados" para mantener un registro de cómo están dispuestos los autos, qué movimientos se han realizado y cuánto "cuesta" llegar a ese punto. Usa una cola de prioridad para asegurarse de que siempre está revisando los escenarios (estados) más prometedores primero, basándose en una métrica llamada heurística.

El punto de partida, o estado inicial, se crea a partir de un archivo y se coloca en la cola. Luego, el algoritmo entra en un ciclo de revisar el estado más prometedor, ver si es la solución y, si no lo es, generar nuevos estados moviendo los autos de diferentes maneras. Si un nuevo estado parece útil y no se ha revisado antes, se añade a la cola para explorarlo más tarde. Si se encuentra una solución, se muestra un recorrido de los movimientos realizados para llegar allí. Y así seguimos, buscando la solución moviendo autos aquí y allá, explorando diferentes posibilidades hasta encontrar el camino para sacar el auto rojo del estacionamiento o hasta que no queden más opciones para explorar.

heurística o técnicas utilizadas:

Para resolver el problema de "Rush Hour", se implementó el algoritmo A*. Este algoritmo busca el camino más corto entre diferentes "Estados" utilizando una cola de prioridad. Gracias a esta estructura, es posible enfocarse siempre en los "Estados" con el menor coste o la menor heurística, dependiendo de la elección realizada.

En este caso, la estructura de datos seleccionada para la cola de prioridad fue un árbol tipo "Heap". La elección de este árbol se basa en una heurística que ordena los elementos de tal manera que aquel con la menor heurística siempre se ubique en la primera posición. Por ende, al extraer (pop) un elemento de la cola, siempre se obtiene el "Estado" más prometedor.

La heurística específica que se utilizó para ordenar esta cola de prioridad considera dos componentes:

1. El costo acumulado que ha incurrido un "Estado" hasta el momento (es decir, la cantidad de movimientos realizados).
2. La distancia lineal que falta para que el auto rojo alcance la salida (ignorando cualquier obstáculo en su camino).

De esta manera, la combinación de ambos factores garantiza que se prioricen aquellos "Estados" que, no solo han realizado menos movimientos, sino que también están potencialmente más cerca de la solución final.

Funcionamiento del programa:

El problema se representa mediante una matriz de enteros que refleja la disposición de los autos. Cada vehículo en la matriz se identifica por un ID único y se describe por su posición (X, Y), dirección, longitud y un ID que se le asigna conforme se agrega al vector.

Se define un "estado" que engloba:

- Un vector de autos.
- La operación realizada para alcanzar dicho estado.
- El auto al que se le aplicó la operación.
- Heurística, costo acumulado de movimientos y referencia al estado padre.

Para gestionar estos estados, se emplea una cola de prioridad (Heap) que los ordena según su heurística. Al extraer un elemento (pop), se obtiene siempre el estado con la mejor heurística.

El proceso inicia creando un estado basado en un archivo de entrada, que proporciona la lista de autos (siendo el primero el rojo). A medida que se procesa cada vehículo, se crea su objeto y se agrega al vector inicial. Posteriormente, se configura el tablero (de tamaño fijo 6x6) y se establece el estado inicial, con su vector y un padre nulo.

La función "resolver", perteneciente a la clase "tablero", se invoca con el estado inicial. Se añade este estado a la cola y se configura el tablero con la disposición inicial de autos. Luego, se entra en un bucle que opera mientras la cola tenga estados.

Dentro del bucle:

1. **Extracción del Estado Óptimo:**
    - Extraer el estado óptimo de la cola de estados por procesar.
2. **Verificación y Solución:**
    - Evaluar si el estado extraído representa la solución, es decir, si el auto rojo se encuentra en la posición final.
    - En caso afirmativo, invocar la función mostrarSolucion, que realizará lo siguiente:
        i. Navegar recursivamente a través de los estados anteriores, accediendo al "padre" de cada estado.
        ii. Una vez que se llega a un estado sin "padre" (es decir, NULL), presentar recursivamente la operación realizada y el auto al que se aplicó la operación, remontándose desde la solución hasta el estado inicial.
3. **Generación de Nuevos Estados:**
    - Si el estado extraído no es una solución:
        i. Explorar cada auto del vector de estado actual para evaluar su movilidad.
        ii. Si un auto puede moverse:
            - Crear una copia del vector de estado actual.
            - Actualizar la posición del auto móvil.
            - Generar un nuevo estado con la configuración actualizada, para cada auto del vector.

4. **Gestión de Nuevos Estados:**
    - Asegurar que el nuevo estado generado no esté previamente en la cola de estados.
    - Si el estado es válido:
        i. Incrementar su costo acumulado.
        ii. Añadir el nuevo estado a la cola para futuras evaluaciones.

Este proceso se repite hasta encontrar la solución o agotar las posibilidades.

Aspectos de implementación y eficiencia:

La eficiencia de una solución no solo radica en el algoritmo empleado, sino también en la manera en la que se implementa y las estructuras de datos utilizadas. En el caso de la solución propuesta para el problema "Rush Hour", se pueden destacar varios factores que contribuyen a su eficiencia:

- **Uso del algoritmo A*:** El algoritmo A* es ampliamente reconocido por ser eficiente en la búsqueda de caminos en espacios de estados. Su capacidad para combinar el coste real desde el inicio hasta un estado dado, con una heurística que estima el coste desde ese estado hasta la meta, le permite seleccionar siempre la opción más prometedora para explorar.
- **Cola de prioridad con árbol Heap:** La elección de un árbol tipo "Heap" para la cola de prioridad es crucial. Esta estructura garantiza que se pueda insertar y extraer el estado con la mejor heurística en tiempo logarítmico, lo que acelera considerablemente el proceso de búsqueda.
- **Evitación de estados redundantes:** Antes de insertar un nuevo estado en la cola, se verifica si ya existe en ella. Esta comprobación es esencial para evitar trabajo innecesario. Si un estado ya se encuentra en la cola, cualquier nuevo estado idéntico que se genere posteriormente siempre será menos eficiente, ya que habrá requerido más movimientos para llegar al mismo punto. Al no agregar estos estados redundantes, se reduce el espacio de búsqueda y se acelera el proceso.
- **Heurística ad hoc:** La heurística diseñada para este problema, que combina el coste acumulado con la distancia lineal del auto rojo a la salida, es particularmente efectiva para guiar la búsqueda hacia la solución de manera rápida.

En resumen, la solución propuesta es eficiente gracias a la combinación de un algoritmo de búsqueda inteligente, estructuras de datos adecuadas y técnicas específicas para reducir el espacio de estados. Todo esto garantiza que se encuentre la solución en el menor tiempo posible y en los menores movimientos, utilizando menos recursos y evitando caminos y estados innecesarios.

Tabla de tiempos de ejecución archivos de prueba.

Nombre archivo	Tiempo (Segundos)
game1.txt, walls1.txt	0.713198
game2.txt, walls1.txt	3.566686
game3.txt, walls1.txt	3.11987
game4.txt, walls1.txt	1.57901
game5.txt, walls1.txt	3.84475
game6.txt, walls1.txt	1.94384

game7.txt, walls1.txt	25.5764
game8.txt, walls1.txt	152.052

Ejecución del Código:

Para poner en marcha el código, es imperativo operar en un entorno con el sistema operativo Linux y asegurarse de que todos los archivos .h y .cpp correspondientes a la solución desarrollada, así como los archivos .txt para pruebas, estén organizados en un mismo directorio. A continuación, se describe el proceso paso a paso:

1. **Preparación del Entorno:**
    - Abra una terminal de Linux y navegue hasta el directorio que contiene todos los archivos mencionados anteriormente.
2. **Compilación:**
    - Ejecute el comando make en la terminal para compilar todos los archivos necesarios del programa.
3. **Ejecución del Programa:**
    - Posteriormente, invoque ./main para iniciar la ejecución del programa.
    - Al comienzo, el programa solicitará, a través de la consola, el nombre del primer archivo, que debe contener la información sobre cada automóvil en el tablero.
    - A continuación, se le pedirá que proporcione el nombre del segundo archivo, que debe albergar los datos de cada pared en el tablero.
    - Nota: En caso de que alguno de estos archivos no se encuentre en el directorio, o si se introduce un nombre incorrecto, el programa se interrumpirá y emitirá un mensaje de error.
4. **Despliegue de Resultados:**
    - Si todo procede sin inconvenientes, el programa se ejecutará y presentará inicialmente el vector de autos y el tablero en su estado inicial, mostrando cada auto en su respectiva posición.
    - Al concluir, se mostrará el vector de autos finales, el tablero final, los pasos seguidos para resolver el tablero introducido y el tiempo que tardó en encontrar la solución óptima para dicho tablero.

Librerías Utilizadas:

El desarrollo del código implicó el uso de diversas librerías, entre las que se incluyen <sstream>, <iostream>, <stdlib>, <ctime>, <string> y <fstream>.