Práctica 4: Rush Hour

Carmen Toribio Pérez(22M009) y Marcos Carnerero Blanco(22M039)

Introducción

En esta práctica hemos modelado un clasificador y visualizador de mapas del juego Rush Hour, para lo cual hemos implementado un resolvedor de mapas a base del algoritmo A*. Este juego consiste en un tablero de 6x6 donde los jugadores deben mover vehículos para liberar el coche objetivo, inicialemente en el extremo izquierdo del mapa y tercera fila. El sistema calcula la dificultad de cada tablero ponderando el número mínimo de movimientos necesarios para alcanzar la solución, los coches en el tablero y su disposición inicial.

Algoritmo A* y su implementación

Nuestro resolvedor utiliza el algoritmo A* para encontrar la secuencia óptima de movimientos que resuelve el tablero. A* combina la búsqueda por costo uniforme y la búsqueda heurística, utilizando una función de evaluación $f(n) = g(n) + h(n)$, donde:

- **Función heurística**: Cantidad de véhiculos bloqueando el camino del coche objetivo + 1, ya que como mínimo se necesitarán estos movimientos para liberar el coche, lo cual hace a esta función una heurística aceptable.
- **Costo acumulado**: Número de movimientos realizados
- **Cola de prioridad**: Usamos un **Data.Set** que ordenamos según el valor de $f(n)$, para asegurar que siempre procesamos el nodo con menor costo estimado primero.

Algorithm 1 A* para Rush Hour

Inicializar algoritmo con el estado inicial del tablero
while queden estados por explorar **do**
  if nodo es solución **then**
    return g(n)
  end if
  for movimiento válido no visitado previamente **do**
    Generar estados sucesores con los movimientos posibles de los vehículos
    if estado no visitado **then**
      Insertar el hijo a la frontera de búsqueda con $f(n) = g(n-1) + h(n) + 1$, donde $g(n-1)$ es el costo del padre y $h(n)$ es la heurística del hijo, con un paso más al haber movido un vehículo.
    end if
  end for
end while

Calculo de la dificultad

Una vez tenemos la solución del tablero, calculamos su dificultad mediante la siguiente fórmula:

$$0.7.\text{stepScore}^{1.4} + 0.2.\sqrt{\text{movedCarsScore}} + 0.1.\text{simmetryScore}$$

En la cual cada termino representa:

- **stepScore**: Número de movimientos necesarios para resolver el tablero, que se obtiene directamente del algoritmo A*.
- **movedCarsScore**: Cantidad de coches que se han movido al menos una vez durante la resolución del tablero, lo cual nos da una idea de la complejidad del tablero.
- **simmetryScore**: Grado de simetría del tablero, calculado como la cantidad de coches que están en posiciones simétricas respecto a uno de los ejes del tablero.

Visualización del tablero

Hemos optado por implementar una visualización sencilla del tablero utilizando la librería `Gloss`, con la interfaz incluyendo controles básicos para pausar, reiniciar y salir del juego. Para la selección de colores hemos usado un generador de numeros pseudoaleatorios, de forma que cada coche tiene un color diferente, y el coche objetivo es siempre rojo. La visualización se actualiza cada medio segundo para mostrar el movimiento de los vehículos hasta alcanzar la solución.

Librerías usadas

Hemos usado varias Librerías de Haskell para facilitar el desarrollo del sistema:

Librería gráfica: Gloss

Para la facilitar la visualización del tablero y la animación de movimientos, hemos utilizado la librería `Gloss`. Esta librería permite crear gráficos de forma sencilla y eficiente, ideal para simulaciones como la nuestra. Nos interesa principalmente para:

- **Visualización del tablero** Nos permite representar el estado del tablero de forma gráfica, mostrando los vehículos con colores y su disposición en el mapa.
- **Animación de movimientos con `animate`** Actualizamos el estado del tablero cada medio segundo, mostrando cómo se mueven los vehículos hasta alcanzar la solución.
- **Interfaz de usuario con controles básicos** Podemos pausar, reiniciar y salir del juego mediante teclas, lo que mejora la experiencia del usuario.

Esta librería aún no permitiendo atajos de teclado o uso del portapapeles, lo cual hubiese sido útil para una mejor interacción con el usuario, pidiendo el mapa a resolver por la ventana, su faciliddad de implementación y su integración con Haskell nos hizo decantarnos por ella.

Librería de estructuras: Data

Por la dificultad del proyecto necesitamos poder gestionar eficientemente los estados del tablero y las posiciones de los vehículos. Para ello, hemos utilizado varias estructuras de datos proporcionadas por la librería `Data` de Haskell:

- `Data.Map` para el tratamiento de los estados del tablero, parsearlos, escribirlos y pasarlos como argumentos.
- `Data.Set` para gestión de posiciones ocupadas.
- `Data.List` para manipulación de listas y generación de movimientos válidos, con funciones como find.

Estructura usada

Módulos

Los módulos del proyecto están organizados de la siguiente manera:

Módulo	Función
Main	Inicialización de la ejecución e implementación específica de A*
AStar	Implementación abstracta del algoritmo
BoardUtils	Definición de tipos, parser de los mapas y movimientos posibles del tablero
Difficulty	Cálculo de la dificultad del tablero
Visualizer	Renderizado del mapa con Gloss y bucle de servicio

Tipos declarados

En cada modulo hemos definido los tipos necesarios para representar el estado del tablero, los vehículos y las métricas de dificultad. Hemos usado varios type para facilidad de lectura y mantenimiento del código, así como para mejorar la legibilidad. Los data son principalmente usados como structs de C, manteniendo en un solo tipo toda la información relevante para los calculos del modulo. A continuación mostramos la implementación:

```
module BoardUtils where

type Position = (Int, Int) -- (fila, columna)

data Orientation = Horizontal | Vertical deriving (Show, Eq, Ord)

data Car = Car
  { carId :: Char, -- Identificador del coche (A-Z)
    positions :: [Position], -- Lista de posiciones ocupadas por el coche
    orientation :: Orientation -- Orientación del coche (Horizontal o Vertical)
  }
  deriving (Show, Eq, Ord)

type Board = [Car] -- Lista de coches en el tablero

module Difficulty where

data SolutionMetrics = SolutionMetrics
  { steps :: Int, -- Número de movimientos en solución óptima
    movedCars :: Int, -- Cantidad de coches diferentes movidos
    carCount :: Int, -- Cantidad total de coches en el tablero
    symmetryScore :: Int -- Grado de simetría (0-100)
  }
  deriving (Show)

module Visualizer where

type Step = Int

data World = World
  { steps :: [Board] -- Lista de tableros que representan los pasos de la solución
  , current :: Step -- Índice del paso actual en la lista de pasos
  , playing :: Bool -- Indica si el visualizador está en modo "play" (reproducción automát
  , difficultyLabel :: String -- Etiqueta de dificultad del tablero
  }
```

Mejoras y optimizaciones

Por la naturaleza de la práctica hemos implementado una solución básica del problema, centrándonos en la facilidad de implementación y la claridad del código. Sin embargo, hemos identificado varias áreas de mejora y optimización en las que se podría trabajar:

- **Optimización de la heurística**: La heurística actual es simple y no tiene en cuenta la disposición de los vehículos no directamente bloqueantes.
- **Paralelización de la búsqueda**: Dado que A* explora múltiples nodos, se podría implementar una versión paralela del algoritmo para aprovechar mejor los recursos del sistema.
- **Mejora de la visualización**: Implementar controles más avanzados, como la posibilidad de avanzar o retroceder en los pasos de la solución, o incluso permitir al usuario interactuar con el tablero.
- **Optimización de la gestión de estados**: Utilizar estructuras de datos más eficientes para la gestión de los estados del tablero, como tries o tablas hash, podría mejorar el rendimiento en tableros díficiles.
- **Generación de tableros aleatorios**: Implementar un generador de tableros aleatorios con diferentes niveles de dificultad, para poder probar el sistema con una variedad de casos.
- **Mejora de la interfaz de usuario**: Implementar una interfaz gráfica más completa, con opciones para cargar tableros desde archivos o introducirlos manualmente, así como una mejor visualización de los movimientos realizados.

Conclusiones

El sistema logra calcular consistentemente la dificultad de tableros mediante A*, demostrando la efectividad de algoritmos heurísticos en problemas de planificación. Las estructuras inmutables de Haskell mostraron ventajas en la gestión segura del estado, aunque requirieron técnicas específicas de optimización. El lenguaje nos permitió implementar un sistema robusto y eficiente, con técnicas de orden superior que facilitan la modelización de este problema. La visualización con Gloss mejoró la experiencia del usuario, permitiendo observar el proceso de resolución en tiempo real.