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

Librerías usadas

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

0.1 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.

0.2 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

0.3 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

0.4 Tipos declarados

```
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

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)

type Step = Int

data World = World {steps :: [Board], current :: Step, playing :: Bool}
```

Problemas en el desarrollo

- **Representación eficiente del estado**: Solucionado con hashing de posiciones
- **Generación de movimientos válidos**: Implementación mediante guardas en Haskell
- **Rendimiento en grandes tableros**: Optimizado con poda de estados repetidos

1 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.