INFORME APP 3

Lenguajes y Paradigmas de la Programación

Sección 2 - Justo Vargas

Integrantes:

Alonso Paniate

Felipe Retamal

Informe Final: App #3 - El Bosque de las Runas Mágicas

1. Introducción

El presente informe detalla el proceso de desarrollo, las decisiones de diseño y el funcionamiento de la aplicación "El Bosque de las Runas Mágicas". El objetivo del proyecto fue desarrollar en Haskell una solución puramente funcional para encontrar el camino de máxima energía a través de una matriz de runas, partiendo de la esquina superior izquierda a la inferior derecha.

El desarrollo enfrenta desafíos significativos relacionados con el rendimiento y la compatibilidad de librerías, los cuales fueron superados mediante un proceso iterativo de optimización y depuración. La solución final es una aplicación robusta y funcional que cumple con todos los requisitos especificados en el enunciado del proyecto.

2. Diseño de la Solución y Decisiones Tomadas

El diseño de la aplicación evolucionó para superar los desafíos técnicos encontrados, principalmente el rendimiento del algoritmo de búsqueda.

2.1. Diagnóstico Inicial: Problema de Rendimiento

La primera versión del código, aunque funcional para matrices pequeñas (3x3), se "congelaba" o demoraba un tiempo inaceptable para matrices de 6x6. El análisis del problema reveló que se estaba utilizando un algoritmo de búsqueda "a ciegas" o de fuerza bruta. Este enfoque explora una cantidad exponencial de caminos posibles (explosión combinatoria), lo que lo vuelve computacionalmente inviable para grillas de mayor tamaño.

2.2. Decisión de Diseño: Adopción del Algoritmo A* (A-estrella)

Para solucionar el problema de rendimiento, se decidió reemplazar el algoritmo original por **A***, un algoritmo de búsqueda informada y estándar en la industria para encontrar caminos óptimos.

*¿Por qué A?** A diferencia de una búsqueda a ciegas, A* utiliza una **función heurística** para estimar cuán "prometedor" es un camino. En cada paso, prioriza las rutas que no solo tienen una buena energía acumulada, sino que también parecen estar más cerca de la meta con un mayor potencial de ganancia. Esto le permite "podar" ramas de búsqueda no prometedoras y encontrar la solución óptima de manera drásticamente más eficiente.

Nuestra heurística se definió como: heurística = (distancia_manhattan_a_la_meta) * (valor_maximo_de_runa)

Esta heurística es "admisible" (nunca sobreestima el costo real), lo que garantiza que A* siempre encontrará el camino óptimo.

2.3. Desafío Técnico: Problemas con la Librería Data.Heap

La implementación más eficiente de A* utiliza una **cola de prioridad**, comúnmente implementada con una estructura de datos Heap. Inicialmente, se intentó usar la librería heap de Haskell. Sin embargo, este enfoque derivó en una frustrante serie de errores de compilación. Los errores indican que la versión de la librería instalada en el entorno de desarrollo tenía una interfaz (API) diferente y en conflicto con la documentación y los ejemplos estándar.

Tras múltiples intentos fallidos por adaptar el código a la API de la librería, se tomó una decisión crucial para garantizar la estabilidad y funcionalidad del proyecto.

2.4. Decisión Final: Reemplazo del Heap por una Lista Ordenada

Para eliminar la dependencia problemática y garantizar que el programa compilara y funcionara, se decidió reemplazar el Heap por una **lista ordenada** estándar de Haskell (Data.List).

- **Ventajas:**
    1. **Estabilidad y Compatibilidad:** Eliminó todos los errores de compilación al no depender de librerías externas con APIs inconsistentes.
    2. **Funcionalidad Garantizada:** El algoritmo A* sigue funcionando correctamente. La lista se mantiene ordenada por prioridad en cada paso.
- **Desventajas:**
    1. **Rendimiento:** Ordenar una lista (O(n log n)) es menos eficiente que insertar en un Heap (O(log n)). Este es el motivo por el cual la versión final, aunque funcional, todavía experimenta una demora notable en matrices de 6x6. Es el "precio" que se pagó por tener una solución estable y funcional que cumpliera el plazo.

El último ajuste fue añadir deriving (Eq, Ord) a la definición del tipo de datos Estado, para que Haskell supiera cómo desempatar y ordenar dos estados con la misma prioridad.

3. Explicación del Funcionamiento de la Aplicación

La aplicación se ejecuta como un programa de consola que recibe dos argumentos: la matriz del bosque (en formato de lista de listas JSON) y la energía inicial del mago.

3.1. Estructuras de Datos Principales

- **Estado:** Representa una "foto" del mago en un momento dado. Almacena:
  - posAct: La coordenada (fila, columna) actual.
  - energia: La energía acumulada hasta ese punto.
  - camino: La lista de coordenadas visitadas.
  - mask: Un "mapa de bits" (Word64) que registra de forma ultra eficiente las celdas ya visitadas para evitar ciclos.
- **PriorityQueue:** Es una lista de tuplas (Prioridad, Estado). Se mantiene siempre ordenada para que la cabeza de la lista sea el estado más prometedor a explorar.

3.2. Flujo de Ejecución

1. **Inicio (main)**: El programa lee y valida los argumentos de la línea de comandos.
2. **Llamada a bestFirst**: Se inicia el proceso de búsqueda. Se calcula el valor de la runa más alta (para la heurística) y se crea el estado inicial en la esquina (0,0). Este primer estado se inserta en la cola de prioridad.
3. **Bucle de Búsqueda (search)**: Esta es la función recursiva que impulsa el algoritmo: a. Extrae el estado con mayor prioridad de la cola. b. **¿Es la meta?** Si la posición actual es la esquina inferior derecha (n-1, n-1), se ha encontrado el camino óptimo. El programa termina y devuelve la lista de coordenadas y la energía final. c. **¿Ya se visitó este estado con más energía?** Se comprueba en un Map si ya hemos llegado a esta misma celda, por el mismo camino (mask), pero con una energía superior. Si es así, se descarta esta ruta y se continúa con el siguiente estado de la cola. d. **Expansión (expandir)**: Si el camino es válido, se generan todos los posibles "hijos" (movimientos a celdas adyacentes válidas). e. **Actualización de la Cola**: Para cada hijo, se calcula su nueva energía, su nueva prioridad (energía + heurística) y se inserta en la cola de prioridad. Luego, la lista se vuelve a ordenar. f. **Recursión**: Se vuelve a llamar a search con la nueva cola de prioridad.
4. **Impresión de Resultados**: Una vez que search devuelve un resultado, main lo formatea y lo imprime en la consola. Si la cola de prioridad se vacía y nunca se llega a la meta, significa que no existe un camino válido.

4. Cumplimiento de Requerimientos del Proyecto

Para realizar la App 3, se buscó cumplir con todos los requerimientos pedidos cumpliendo con:

- **Paradigma Funcional y Lenguaje** :
    1. **Programación Funcional Pura:** ✅ Sí. No hay estado mutable ni efectos secundarios (salvo la impresión final). Todas las transformaciones de datos crean nuevas estructuras inmutables.
    2. **Solución Recursiva:** ✅ Sí. La función search es el núcleo recursivo del programa.
    3. **Lenguaje Haskell:** ✅ Sí. El proyecto está implementado íntegramente en Haskell.
- **Requerimientos Funcionales**:
    1. **Matriz NxN:** ✅ Soportado por el tipo Bosque = [[Int]].
    2. **Movimientos (Derecha, Abajo, Diagonal, Izquierda, Arriba):** Implementados en la lista movimientos. La restricción de no visitar celdas se gestiona con la máscara de bits (visMask).
    3. **Costo Diagonal:** ✅ Se aplica un costo de 2 puntos si el movimiento es (1,1).
    4. **Trampa de Valor 0:** ✅ Se aplica una penalización de 3 puntos si el valor de la runa es 0.
    5. **Suma/Resta de Energía:** ✅ La energía se actualiza en cada paso con eNew = eAct - costo_movimiento + valor_runa.
    6. **Energía Inicial:** ✅ Se recibe como argumento de entrada.
    7. **Camino con Mayor Energía:** ✅ Garantizado por el algoritmo A*, que encuentra el camino óptimo.
    8. **Invalidación por Energía < 0:** ✅ Cualquier camino que resulte en energía negativa se descarta inmediatamente (if eNew < 0 then Nothing).
- **Interfaz Gráfica Opcional (GUI)**:
    - **Desarrollo en Python con Tkinter:** ✅ Adicionalmente, se desarrolló una interfaz gráfica de usuario utilizando Python y su librería estándar Tkinter. Esta GUI actúa como un *frontend* o cliente para el ejecutable de Haskell.
    - **Funcionamiento:** La interfaz permite al usuario ingresar la matriz y la energía inicial de forma visual. Al presionar el botón, el script de Python ejecuta el App3.exe como un subproceso, pasándole los datos. Luego, captura la salida de texto de la consola, la procesa y muestra los resultados

(energía final y camino) de forma amigable, incluyendo una animación del camino óptimo sobre la grilla.

- **Propósito:** Aunque no era un requisito estricto del *backend* funcional, esta GUI cumple con el objetivo general de crear una aplicación usable y demuestra una excelente práctica de interoperabilidad entre un núcleo lógico funcional (Haskell) y una capa de presentación imperativa (Python).

5. Reflexiones Finales / Autoevaluación

- **¿Qué fue lo más desafiante de implementar en paradigma funcional?** El mayor desafío no fue tanto la lógica funcional en sí (pensar recursivamente o manejar la inmutabilidad), sino la **gestión del ecosistema de Haskell**. Depurar los errores de compilación causados por las inconsistencias de la librería heap fue extremadamente frustrante y consumió una cantidad considerable de tiempo. Demuestra que, aunque el lenguaje sea muy robusto, las dependencias pueden ser un punto débil, al igual que en otros ecosistemas de programación.
- **¿Qué aprendizajes surgieron del proyecto?**
    1. **La importancia de la elección del algoritmo:** Quedó demostrado de forma práctica cómo un cambio de algoritmo (de fuerza bruta a A*) puede cambiar el problema.
    2. **Depuración y resiliencia:** El proyecto fue una lección intensiva sobre cómo interpretar errores de compilación de Haskell y sobre la importancia de tener un "plan B" (como reemplazar la librería heap) cuando una dependencia se vuelve un obstáculo insuperable.
    3. **Trade-offs en ingeniería:** La decisión de usar una lista ordenada en vez de un heap es un ejemplo perfecto de un *trade-off* o compromiso de ingeniería: se sacrificó rendimiento máximo a cambio de estabilidad, compatibilidad y, lo más importante, un producto final funcional.

6. Explicación de Uso de IA

Para el desarrollo de este proyecto se utilizó un Gemini y ChatGPT, que desempeñaron un rol crucial como herramienta de apoyo y depuración.

- **¿Qué tipo de ayuda proporcionó las herramientas?**
    1. **Diagnóstico y Sugerencia de Algoritmo:** La IA analizó el código inicial, diagnosticó correctamente el problema de rendimiento y sugirió la implementación del algoritmo A* como la solución óptima.
    2. **Generación de Código:** Proporcionó la primera versión del código optimizado con A* y la librería heap.
    3. **Depuración Iterativa:** Fue fundamental durante el proceso de compilación. Ante cada nuevo error reportado, el asistente analizaba el mensaje de GHC y proporcionaba una versión corregida del código, abordando los problemas de tipo, API y compatibilidad de las librerías.
    4. **GUI en Pýthon:** Ayuda para la creación de la GUI de la aplicación tomando como base la App3 en Haskell y el elegir la librería Tkinter.
    5. **Validación de resultados:** Ayuda para pedirle Matrices con el resultado ya obtenido, para que el programa entregar el mismo resultado y validar que este funcionara correctamente.
    6. **Solución Final:** Cuando la librería heap demostró ser inviable, el asistente propuso y generó la solución final que reemplaza el heap por una lista ordenada, resolviendo definitivamente los problemas de compilación.

- **¿Cómo se validaron o contrastaron las sugerencias?** La validación fue un proceso práctico y empírico. Cada versión del código sugerida por el asistente fue compilada y ejecutada en el entorno de desarrollo local. Los errores resultantes de la compilación se reportaban de vuelta al asistente, lo que permitía un ciclo de depuración iterativo. La sugerencia final se validó cuando el programa compiló exitosamente y produjo los resultados correctos para las matrices de prueba, demostrando su funcionalidad.