UNIVERSIDAD SIMÓN BOLÍVAR
Dpto. de Computación y Tecnología de la Información
CI-3661 - Laboratorio de Lenguajes de Programación

Proyecto I

monad-wordle - Una implementación en Haskell de Wordle para dos

Autor:
Nestor González 16-10455
Christopher Gómez 18-10892

Profesor:
Fenando Torre

20 de julio de 2022

1. Introducción

El siguiente informe consiste en dar un vistazo a una implementación en Haskell del famoso juego Wordle, presentando las decisiones de diseño tomadas al programar la solución y los algoritmos usados para cada modo de juego, para obtener de ello conclusiones que nos permitan dilucidar las características más relevantes y distintivas del paradigma de programación funcional.

Wordle es un juego en línea que se popularizó alrededor de octubre del 2021, que consiste en adivinar en 6 intentos o menos una palabra de 5 letras planteada por la computadora, dadas unas pistas que indican cuáles letras de la adivinación son acertadas y si están o no en la posición correcta. El juego se basa en otro llamado "Toros y Vacas", y la implementación presentada trata sobre este, donde además se introduce un modo en el que el jugador piensa una palabra y la computadora intenta adivinarla.

Este último modo añade a la implementación una complejidad adicional, que es la de diseñar un solucionador que intentará adivinar la palabra del usuario. Para este solucionador se usa un algoritmo de Minimax para optimizar la adivinación, que se explicará en la sección de diseño de la solución, junto a los demás modos y las optimizaciones y problemas surgidos al modelar y programar.

Luego, el objetivo de implementar `monad-wordle` consiste en conocer las herramientas que ofrece la programación funcional, y en especial Haskell, para la resolución de problemas, así como descubrir nuevas maneras de plantearlos y pensar sobre ellos, explotando las bondades que ofrece este paradigma.

2. Diseño de la solución

`monad-wordle` se divide en los modos *mentemaestra* y *descifrador*, que son, respectivamente, el modo de Computadora vs. Usuario y Usuario vs. Computadora. El código fuente se encuentra dividido en los siguientes módulos:

- `Wordle.Mastermind`: Contiene la función que ejecuta el modo *mentemaestra*, usa las implementaciones de `Wordle.Utils.Checkers` para pedir una palabra del usuario, validarla e indicar los aciertos en el formato indicado, llevando control de las vidas y el historial.
- `Wordle.Decoder`: Contiene la función que ejecuta el modo decodificador, usa las implementaciones de `Wordle.Utils.Minimaxer` para pedir hacer una adivinación, pedir una pista al usuario y generar una siguiente.
    - `Wordle.Utils.IOHelpers`: Contiene funciones de ayuda para las operaciones de entrada-salida, que involucra la carga del archivo de palabras, la escogencia de una palabra al azar y la impresión del historial para el modo *mentemaestra*.
    - `Wordle.Utils.Checkers`: Contiene las funciones que validan y evalúan la adivinación del usuario en el modo *mentemaestra*.
    - `Wordle.Utils.Minimaxer`: Contiene las funciones que implementan el agente de Minimax solucionador del modo *descifrador*.

Luego, la primera consideración de eficiencia tomada en la implementación de `monad-wordle` fue con respecto al cargado de la lista de palabras del archivo dado.

Dado que el archivo contiene alrededor de 10,500 palabras, y constantemente se realizan búsquedas de palabras en ambos modos de juego, se decidió usar la estructura de datos `Data.Set` de Haskell para almacenarlas en memoria principal. La implementación interna de `Data.Set` se trata de un árbol binario autobalanceable, y el archivo de palabras está ordenado en orden lexicográfico ascendente, lo cual permite incluso construir el árbol en tiempo asintótico $O(n)$, y no $(n \lg n)$, por lo cual no hay pérdidas de eficiencia notables respecto al tiempo que tardaría en cargarse en una lista.

Sin embargo, la mayor de las ventajas de este acercamiento con `Data.Set` es que la búsqueda de un elemento en este ocurre en tiempo logarítmico con respecto al tamaño del conjunto, por lo que para validar la respuesta del usuario se recorren menos de 16 nodos (a diferencia de las listas, cuyo tiempo de búsqueda es lineal), resultando en un ahorro de cómputo considerable.

2.1. Modo *mentemaestra*

Para validar las entradas del usuario se decidió usar el tipo de datos `Either`, el cual es una característica útil de Haskell para el manejo de errores sin excepciones, a la vez que se mantiene el tipado fuerte y estático y la pureza funcional. Se retorna `Left` con un mensaje de error en caso de una validación fallida, y `Right` en caso contrario, con la palabra convertida a mayúsculas y sin acentos.

El algoritmo usado para producir la cadena de Toros y Vacas para el usuario consiste en verificar primero los Toros y luego las Vacas. Para verificar los Toros se comparan uno a uno la respuesta de la computadora con la adivinación del usuario, ya que los Toros indican una coincidencia exacta en la posición; luego de verificar los Toros, la función retorna la cadena constuída considerando solo Toros y los caracteres restantes por adivinar de la palabra de la computadora, retorno que es directamente pasado a la función que verifica las Vacas, que solo chequea que si letras del usuario forman parte del restante por adivinar, independientemente de su posición exacta. Cuando se encuentra membresía de una letra en la palabra de la computadora, se marca la Vaca y se sigue revisando, removiéndose la letra del conjunto de caracteres restantes por adivinar, y esto garantiza que se maneja bien cualquier caso borde de letras repetidas, en los cuales solo se marcan dos o más veces, o solo la primera según sea el caso.

Con respecto a la validación de la palabra dada por el jugador, las verificaciones, aunque son todas computacionalmente livianas, se hacen desde la más liviana hasta la más pesada: i) que la palabra tenga 5 letras, b) que todos sus caracteres sean alfabéticos y iii) que pertenezca al conjunto de palabras, habiéndose llevado a cabo antes de ello una uniformización de la entrada del usuario, convirtiéndola a mayúsculas removiendo los acentos agudos.

Por ser sencillo de implementar, se decidió mantener el historial de intentos (la matriz de Toros y Vacas) para compartir el resultado, consistiendo en concatenar cada intento, de ser válido, a la cabeza de una lista de `String` en cada llamada recursiva, para ser impresa al final en orden inverso.

2.2. Modo *descifrador*

3. Conclusiones

El algoritmo presentado para resolver el *problema del cartero rural* (RPP), su implementación en `grafoLib` y los resultados experimentales obtenidos permiten concluir que:

- Sin tomar el elevado tiempo exponencial característico de los algoritmos de fuerza bruta que solucionan los problemas *NP-complejo*, un algoritmo heurístico puede ayudar a obtener soluciones

no tan alejadas de las óptimas, las cuales puede ser útiles en ciertas aplicaciones, y requerir un tiempo muchísimo menor.

- Los algoritmos sobre grafos vistos en el curso sirven para resolver variedades de problemas, tanto computacionales como en otras áreas. En este caso, fueron de utilidad algoritmos para obtener árboles mínimos cobertores, apareamientos perfectos, componentes conexas y caminos de costo mínimo (juntos con sus costos).
- Aunque para resolver esta implementación se necesitó de un isomorfismo entre dos grafos con vértices representados con números naturales, se observó que los grafos isomorfos son de gran utilidad para mapear grafos de cualquier tipo a grafos de números naturales en un intervalo, de forma que se mantiene una implementación sencilla a la vez que se pueden resolver problemas que requieran grafos con vértices de otro tipo (como cadenas de caracteres e incluso objetos). Además, es útil y eficiente para la implementación de la función f del isomorfismo el uso de arreglos y diccionarios como tablas de hash, que toman tiempo lineal en su creación pero luego tiempo amortizado constante en su acceso.
- Si bien se sabe que una estrategia ávida no necesariamente obtiene la mejor solución a los problemas, son en su mayoría sencillas de implementar y pueden usarse para resolver partes de problemas más grandes proporcionando soluciones razonablemente cercanas a la óptima.
- Las ideas y algoritmos sobre grafos dirigidos pueden, en muchos casos, extenderse a grafos no dirigidos, teniendo cuenta de la "orientación" de las aristas y considerando que siempre se tiene que los lados (u, v) y (v, u) son iguales.
- Cuando se dispone de suficiente tiempo y recursos, puede resultar beneficioso implementar y ejecutar varias veces un algoritmo que explora el espacio de soluciones para obtener soluciones mejores que la de los algoritmos deterministas, especialmente si el tiempo de ejecución del algoritmo no determinista es similar o mejor que el determinista.
- Modelar problemas con grafos proporciona en la mayoría de los casos poderosas herramientas para resolver desde problemas cotidianos a complejos, de forma argumentablemente más sencilla a si se buscase resolver los mismos problemas con el uso de otras estructuras matemáticas o de datos.

4. Referencias

1. David Avis. A survey of heuristics for the weighted matching problem. Networks, 13(4):475–493, 1983.
2. Wen Lea Pearn and TC Wu. Algorithms for the rural postman problem. Computers & Operations Research, 22(8):819–828, 1995.