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*

En este modo se presentaron más dificultades, pues principalmente la creación de un árbol a lo sumo 10-ario con 4 niveles de profundidad es un cálculo computacionalmente intenso y el planteamiento en programación funcional de la construcción del agente Minimax resulta complejo al no estar acostumbrados a pensar en este paradigma de resolución de problemas, sin embargo, al lograr conseguir la solución se puede notar por qué es este un problema adecuado para resolverse en un lenguaje de programación declarativo con más facilidad que en uno imperativo.

Para el modelado del problema se usaron las siguientes extructuras de Datos:

```
data Guess = Guess String Int
data Score = Score String Int
data Rose a = Leaf a | Node a [Rose a]
data MinimaxNode = MinimaxNode {
    value    :: String,
    wordSet  :: Set Guess,
    scoreSet :: Set Score,
    score    :: Int
}
```

Al tener que generar un árbol con cantidad no acotada y probablemente variable de nodos hijo, se decidió implementar la estructura de datos `Rose`. Por otro lado, se modelan las Strings de adivinación y calificación del usuario con los registros `Guess` y `Score`, respectivamente, cada uno de los cuales contiene una String y su calificación, de acuerdo a la dada en el enunciado.

De tal manera, se diseñaron funciones que dada una String crean el `Score` o la `Guess` con su puntuación correspondiente. Como la creación del árbol es costosa, se decidió usar un `Int` para representar las puntuaciones con más precisión y hacer cálculos con mayor rapidez, siendo estas el resultado de multiplicar por 10 la puntuación del enunciado. Además, en el caso del cálculo de los `Score`, para evitar que cada String valga 10 y se reste 2 o 1 por cada Toro o Vaca, se cambió el esquema de puntuación a uno análogo que no usa resta, sino que suma 0 por cada Toro, 1 por cada Vaca y 2 por cada guión, siendo las puntuaciones resultantes equivalentes.

Luego, se guarda el estado del juego en cada ronda en dos conjuntos, un `Set Guess` y un `Set Score`, que contienen estos las palabras que sirven de posible adivinación y posible calificación del usuario en la ronda actual, respectivamente. Así, se aprovecha la implementación de Haskell de `Set` como un árbol binario de búsqueda para implementar la typeclass `Ord` en `Guess` y `Score`, de forma que al crear cada conjunto se tengan las adivinaciones posibles en orden ascendente de puntuación, y las puntuaciones posibles del usuario en orden descendente de puntuación, que servirá posteriormente para simplificar la construcción del árbol Minimax.

Estos conjuntos se crean y calculan una sola vez durante todo el programa. En el caso del `Set Guess` se pasan todas las palabras del `Set String`, cargadas en `Main` a `Guess`, y se usa la comprensión de listas de Haskell para obtener todas las combinaciones posibles de 5 letras de `'-'`, `'V'` y `'T'`.

Por otro lado, se diseñaron distintos algoritmos para aprovechar al máximo las pistas que da el jugador y filtrar lo más posible el conjunto de adivinaciones y puntuaciones. Estos algoritmos generan filtros que se dividen en filtros de posición, que filtran palabras con respecto a si tienen o no un caracter en una posición específica, y filtros frecuenciales, que filtran palabras por la cantidad de veces que contienen cierto caracter. A continuación se explica cómo se generan los filtros en cada caso:

- Para el conjunto de `Guess`:
  - **Filtros posicionales**:
    - Se filtran las palabras con el mismo caracter en las posiciones que el jugador indicó como Toro.

    - Se filtran las palabras que no contengan el caracter en la misma posición que el usuario indicó como Vaca.
  - **Filtros frecuenciales**:
    - Se filtran las palabras que contengan al menos n apariciones de cierto caracter en la misma posición que el usuario indicó Vaca, donde n es el número de veces que este caracter aparece asociado con una Vaca, aprovechando así las pistas que puedan indicar repetición de caracteres en la palabra.
    - Se filtran las palabras que contengan 0 apariciones de cierto caracter si este solamente aparece asociado a guiones y no vacas.

- Para el conjunto de `Score`:
  - **Filtros posicionales**:
    - Se filtran las puntuaciones que tengan Toros exactamente en las mismas posiciones, pues no es posible que al detectarse un Toro la próxima adivinación sea una palabra que no contenga el mismo Toro.
  - **Filtros frecuenciales**:
    - Se filtran las palabras que contengan al menos la misma cantidad de Toros y Vacas, pues una vez obtenido un Toro este no desaparecerá en las próximas adivinaciones, y las Vacas seguirán siendo Vacas o se convertirán en Toros.

Con estos filtros, desde la segunda adivinación los cómputos se aligeran considerablemte, ya que estos reducen en gran manera el conjunto de posibilidades y, por ende, los cálculos necesarios al construir el árbol.

En prosecución, el registro `MinimaxNode` se encarga de guardar el estado de cada adivinación/nodo del árbol, y al momento del diseño se deliberó que estos debían contener solamente información estrictamente necesaria para generar un nuevo nivel del árbol de la forma más eficiente posible. Por ello, se guardan en estos un valor, que bien puede ser una String de calificación del usuario en nodos de profundidad par, o una palabra de adivinación, en nodos de profundidad impar; los conjuntos de palabras y puntuaciones posibles dado el valor y el estado del juego, de forma que estos no se vuelvan a calcular mientras se construye el árbol; y la puntuación del nodo, que inicialmente contiene la misma puntuación de `value`.

Dadas estas estructuras de datos, las herramientas que nos ofrece Haskell y las librerías permiten implementar la construcción el árbol Minimax con un algoritmo simple de recursión mutua que aprovecha el estilo declarativo y funcional del lenguaje, descrito a continuación en alto nivel:

- Para generar un nivel de minimización (dado un `MinimaxNode` padre con una puntuación):
  - Se toman 10 palabras del conjunto que guarda el nodo (`take 10`) - Nótese que no se ordenan porque el conjunto ya las contiene ordenadas en orden ascendente de puntuación.
  - Se construye un nodo para cada una de estas palabras (`map`).
  - A partir de cada nodo como padre, se genera un nivel de maximización (`map`).
  - Si el nuevo nivel es generado no tiene elementos, el nodo es una hoja, de otra forma, es una rama.

- Para generar un nivel de maximización (dado un `MinimaxNode` padre con una adivinación):
  - Se toman 10 puntuaciones del conjunto que guarda el nodo (`take 10`) - Análogamente, no se ordenan porque el conjunto ya las contiene ordenadas en orden descendente de puntuación. Para tomar 10 `Score` que sean válidos, se toman solamente si al filtrar el conjunto del propio nodo con el `Score` es no vacío. Gracias al modo de evaluación de Haskell esto resulta sumamente eficiente.
  - Se construye un nodo para cada una de estas puntuaciones (`map`), los conjuntos que guarda cada nodo son los filtrados dada la palabra del nodo padre y el `Score` de la lista (`filter`).
  - A partir de cada nodo, se genera un nivel de minimización (`map`).
  - Si el nuevo nivel es generado no tiene elementos, el nodo es una hoja, de otra forma, es una rama. A su vez, no genera un siguiente nivel si la profundidad ya es 4, deteniendo la recursión mutua.

Una vez generado el árbol y puntuado se recorre este de nuevo para incrementar a cada nodo la suma de las puntuaciones de sus hijos y tomar el nodo de primer nivel con menor valor como próxima adivinación, terminando el algoritmo.

La detección de tramposos en el marco de todas las funciones que se tienen consiste simplemente en ver si el conjunto resultante al filtrarse este de acuerdo a la pista del jugador es vacío, en cuyo caso se descarta cualquier posibilidad incluso antes de comenzar a generar el árbol.

3. Consideraciones de la implementación y ejecución

Al ejecutar el programa final, el modo *mentemaestra* no sufre de ninguna ineficiencia, y tras numerosas pruebas muestra total precisión en las pistas mostradas al usuario, así como en la matriz de Vacas y Toros y el manejo de intentos restantes.

Por otra parte, para el modo *decodificador*, el algoritmo resulta más eficiente de lo esperado, aunque en ciertas ejecuciones la construcción del árbol para hallar la segunda adivinación puede tardar entre 2 y 8 segundos[1], pero esto se debe a que es en el primer intento donde se construye el árbol más grande. Quizás la optimización más importante es tener el conjunto de adivinaciones precalculado y las palabras puntuadas una sola vez al comienzo del programa, y que las sucesivas adivinaciones obtienen estos conjuntos cada vez más pequeños, por lo que por lo general, para la tercera adivinación el tiempo de respuesta es por lo general menor a 2 segundos, y las posteriores adivinaciones se generan virtualmente de forma instantánea.

Luego, se decidió que la primera adivinación del modo *decodificador* sería una palabra escogida al azar, para que siempre hayan posibilidades de adivinar la palabra del jugador desde otra palabra inicial; si se comenzara siempre con la misma palabra, por la naturaleza del algoritmo (y la implementación funcionalmente de Haskell[2]), existirán palabras imposibles de adivinar, y la forma de cambiar esto es comenzar siempre desde una palabra distinta escogida al azar.

Para muchas palabras el solucionador implementado resulta infalible y adivina correctamente la palabra en alrededor de 4 intentos en promedio, sin embargo, tiene dificultad para adivinar conjuntos

[1] Pruebas hechas en Debian 11 Bullseye 64-bit, en un procesador Intel(R) Core i3-2120 @ 3.30GHz, con el programa compilado en GHC version 8.8.4

[2] La misma entrada produce siempre la misma salida

de palabras iguales excepto por una letra, en cuyo caso el solucionador llega a los 6 intentos sin hallar una solución si el usuario pensaba en una de las palabras menor puntuada de ellas. Un ejemplo de este caso es `RUPIA`, que coincide con `RUBIA`, `RUCIA`, `RUBIA`, `RUGIA`, `RUJIA`, `RUSIA` y `RUMIA`, donde la mayoría de los intentos el algoritmo explorará cada palabra restante como opción en orden sin llegar a la del usuario. Una solución a esto, que sin embargo no se podría implementar en el estilo funcionalmente puro de Haskell, es escoger una palabra al azar de las restantes para aumentar la probabilidad de ganar, en vez de recorrer la secuencia siempre en el mismo orden y no llegar a la palabra del usuario.

4. Conclusiones

(No se ha hecho)

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.