ESTRUCTURAS DE DATOS Y ALGORITMOS

TRABAJO PRÁCTICO ESPECIAL

ALUMNOS

- 52056 – Juan Marcos Bellini
- 55291 – Julián Rodriguez Nicastro
- 55824 – Juan Li Puma

ITBA 55 años
INSTITUTO TECNOLOGICO DE BUENOS AIRES
•UNIVERSIDAD PRIVADA•

TABLA DE CONTENIDOS

INTRODUCCIÓN

El presente informe trata acerca del trabajo práctico especial de la materia Estructuras de Datos y Algoritmos. El mismo consiste en la implementación de un algoritmo que resuelve tableros de una variante del juego *Scrabble*.

Para poder cumplir con el objetivo, se han desarrollado dos algoritmos. Por un lado, uno encuentra el tablero que más puntos sume, en el tiempo que sea necesario. Por otro lado, el segundo obtiene la mejor solución hasta el momento (indicándole durante cuánto tiempo tiene que ejecutarse).

A continuación se describen ambos algoritmos, las estructuras de datos utilizadas, los problemas encontrados durante el desarrollo, y las decisiones tomadas para abordarlos, y comparaciones entre ambos algoritmos.

ALGORITMOS UTILIZADOS

Para poder encontrar soluciones al problemas, se implementaron dos algoritmos, uno exacto y otro aproximado. El exacto, como lo indica su nombre, encuentra la mejor solución al problema. Esto es, el tablero con mayor puntaje. En cambio, el segundo, encuentra una solución que puede, o no, ser la mejor, pero que es aceptable.

El problema del primer algoritmo es que es demasiado complejo. Al tratar de buscar la mejor solución, se debe realizar numerosas pruebas, hasta llegar al tablero buscado. Si la cantidad de letras y palabras disponibles es muy grande, el algoritmo podría tardar años, o incluso siglos. Por lo tanto, el hecho de poder obtener una solución aceptable – mediante un algoritmo de aproximación – permite resolver problemas que, de otra forma, si bien se resuelve de la manera más óptima, habría que esperar mucho tiempo hasta lograr dar con dicha solución.

ALGORITMO EXACTO

Para obtener la mejor solución posible al problema planteado, se decidió utilizar *backtracking.* Esta técnica consiste en recorrer en profundidad un árbol de soluciones parciales. Al llegar a una hoja de dicho árbol (estado a partir del cual no se puede obtener más soluciones), si dicha solución es mejor que la mejor obtenida hasta el momento, se la remplaza; si no es mejor, se la descarta. El algoritmo termina cuando se recorra todo el árbol o, en el caso a analizar, cuando se obtenga un tablero en el que se hayan ubicado todas las letras.

A continuación se describe el algoritmo en pseudo-código.

```
Sea L el conjunto de Letras que se tiene
Sea D el conjunto de Palabras // El diccionario
Sea mejorSolución un Estado con Tablero vacío // Guardará la mejor solución
Sea máximoPuntaje = Suma(Puntajes de Letras en L)

FUNCIÓN backTrackingSolve(Estado e)

        Sea M el conjunto de Movimientos posibles a partir del Estado e

        SI M es vacío
        ENTONCES:
                SI e.Puntaje > mejorSolucion.Puntaje
                ENTONCES:
                        mejorSolucion = e
                        SI mejorSolución.Puntaje == máximoPuntaje
                        ENTONCES:
                                Terminar la ejecución
                        FIN SI.
                FIN SI.
        SINO:
                PARA TODO movimiento en M:
                        Aplicar movimiento a e
                        backTrackingSolve(e)
                        Quitar movimiento de e
                FIN PARA TODO
        FIN SI.
FIN FUNCIÓN.
```

ALGORITMO APROXIMADO

Debido al problema planteado, fue necesario implementar un algoritmo de aproximación a una solución aceptable. Dentro de todas las técnicas posibles, se eligió utilizar *Hill Climbing* Estocástico ya que ésta es la que mejor se adapta a las necesidades.

Es un algoritmo bastante rápido. Escala rápidamente hacia un máximo (local o absoluto), aunque podría tener el problema de estancarse en un máximo local. Para evitar esto, siempre y cuando quede tiempo de ejecución, el algoritmo se reinicia utilizando otra solución inicial.

A continuación el algoritmo en pseudo-código.

```
Sea e un Estado con Tablero vacío
Sea I el conjunto de Movimientos posibles a partir de e // Estados iniciales
Sea mejorSolución un Estado con Tablero vacío // Guardará la mejor solución
Sea máximoPuntaje = Suma(Puntajes de Letras en L)

FUNCIÓN stochasticHillClimbingSolve(Tiempo t_max)

        Sea t_inicial = 0

        PARA TODO movimiento en I
                Aplicar movimiento a e
                stochasticHillClimbingRecursive(e, t_inicial, t_max)
                SI mejorSolución.Puntaje == máximoPuntaje
                ENTONCES:
                        Terminar la ejecución
                FIN SI.
                Quitar movimiento a e
        FIN PARA TODO
FIN FUNCIÓN
```

```
FUNCIÓN stochasticHillClimbingRecursive(Estado e, Tiempo t_actual, Tiempo t_max)

    SI t_actual >= t_max
        Terminar la ejecución
    FIN SI

    SI e.Puntaje > mejorSolución
    ENTONCES:
        mejorSolución = e
        SI mejorSolución.Puntaje == máximoPuntaje
        ENTONCES:
            Terminar la ejecución
        FIN SI.
    SINO:
        Sea M el conjunto de Movimientos posibles a partir del Estado e
        PARA TODO movimiento en M
            Sea r un número real al azar entre 0 y 1
            SI r < movimiento.probabilidadDeSerElegido
            ENTONCES:
                Aplicar movimiento a e
                t_actual = obtenerTiempoDeEjecución
                stochasticHillClimbingRecursive(e, t_actual, t_max)
            FIN SI.
        FIN PARA TODO
    FIN SI.
FIN FUNCIÓN.
```

ESTRUCTURAS UTILIZADAS

Debido a la gran complejidad del problema, es necesario hacer uso de estructuras que faciliten la ejecución. Estas estructuras deben guardar la información de forma eficiente (utilizando la menor cantidad de memoria posible), y deben poder devolverla rápidamente cuando se les pide algún dato.

Las dos estructuras de datos más importantes que se decidieron utilizar son una variante de *trie* para el diccionario de palabras, y un arreglo de enteros que representa las letras disponibles.

DICCIONARIO

En cuanto al diccionario, se realizo una implementación parcial de un *trie*, el cual tiene únicamente métodos para agregar palabras, para consultar palabras, y para pedir palabras que cumplan una cierta condición.

Básicamente, el *trie* se compone de nodos que almacenan como clave un dato de tipo *char*, y un mapa de nodos siguientes (cuya clave es el *char* siguiente y, el valor, el nodo existente para dicho *char*). A medida que se avanza en el *trie*, se va formando una palabra, hasta llegar a un nodo con la marca '*#*', la cual indica fin de palabra. La raíz del *trie* no es un nodo. El árbol empieza desde un mapa cuyas claves también son de tipo *char*, y los valores, los nodos iniciales de cada palabra.

Para consultar si el diccionario posee una palabra, existe un método que recorre el árbol iterando sobre la palabra a buscar. Si durante la búsqueda no encuentra una de las letras de la palabra, entonces el algoritmo termina.

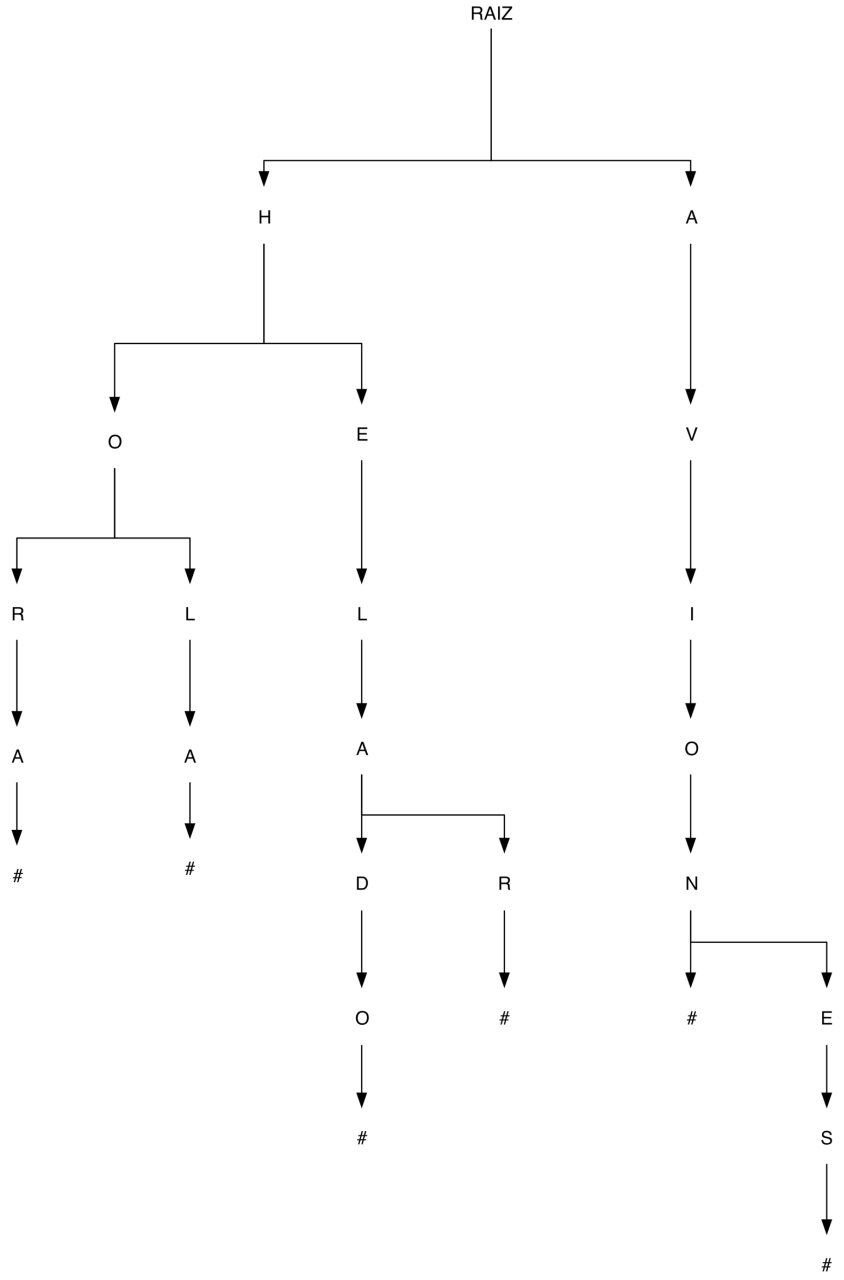
Además, se cuenta con un método que recibe condiciones de palabra. Las condiciones de palabra es un objeto que posee una letra y una posición. Las palabras que cumplen con dicha condición son aquellas que tengan esa letra en esa posición. Es una forma fácil de buscar palabras que puedan entrar en el tablero cuando hay palabras cruzadas.

Los métodos para agregar y consultar si una palabra existe en el diccionario, tienen complejidad *O(L)*, siendo *L* la cantidad de letras de la palabra. En cambio, el método para obtener palabras que cumplan una cierta condición, tiene complejidad *O(N)*, siendo *N* la cantidad de palabras almacenadas.

Suponer el siguiente ejemplo. La palabra "HOLA" se almacenaría de la siguiente manera. En el mapa inicial se tiene la clave 'H', cuyo valor es un nodo que tiene almacenado la letra 'H'. Este nodo tiene también un mapa con la clave 'O', cuyo valor es un nodo con la 'O' almacenada. Repitiendo lo anterior tantas veces como el largo de la palabra, eventualmente se llega a un nodo con el carácter '*#*' almacenado, y sin mapa.

Si se pregunta si posee la palabra "HOLA", responderá que sí. Además, se le puede pedir palabras con una 'H' en la primera letra, y una '0' en la tercera, y dentro de ese conjunto de palabras, aparecerá "HOLA".

A continuación se muestra un diagrama de ejemplo, que almacena las palabras "HOLA", "HORA", "HELADO", "HELAR", "AVION" y "AVIONES".



LETRAS DISPONIBLES

El almacenamiento de letras es mas simple. Para lograr un almacenamiento eficiente, se decidió utilizar un arreglo entero de 26 posiciones. Cada posición representa una letra del diccionario, y el valor en dicha posición, la cantidad disponible para dicha letra.

De esta manera, se podría perder espacio en caso de que una letra no esté, pero el acceso a dicha estructura es muy simple. Al ser un arreglo, se accede con complejidad *O(1).*

Por ejemplo, si se tienen las letras A,B,C,D,A,B,F,E,A,B, el arreglo será de la siguiente manera: [3, 3, 1, 1, 1, 1, 0, 0, …, 0].

PROBLEMAS ENCONTRADOS

El principal problema que se encontró durante el desarrollo de la aplicación fue la enorme cantidad de memoria utilizada durante la ejecución de los algoritmos. Debido a las características del problema a resolver, en cada paso (en este caso, en cada recursión), se crea una nueva rama, la cual a su vez podría tener muchas más ramas, generando una inmensa utilización de recursos.

Teniendo esto en cuenta, se pensó la siguiente solución. La primera fue almacenar cada estado por el que se pasaba (tablero con letras dispuestas de una cierta forma). Cada vez que se intentaba realizar algún movimiento, se verificaría si dicho movimiento no fue analizado previamente. En caso de que sí haya sido, no se procedería en la recursión a partir de este estado.

El problema de esto es que almacenar cada estado es muy costoso. La memoria podría agotarse rápidamente. Además, se perdería tiempo de ejecución en el almacenamiento y consulta de estados. Por ende, finalmente, se descartó. Aplicar la solución descripta generaba más problemas que beneficios.

COMPARACIONES

El

CONCLUSIÓN

El p