NOMBRE: Sebastián Díaz Holmgren

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructuras de datos — 2' 2021

Tarea 2 – Respuesta Hashing

- El uso de tablas de hash para esta tarea es una buena opción ya que nos permite codificar la "hebra madre" en función de las hebras de prueba recorriendo únicamente una vez la hebra madre lo cual es más eficiente que con otras técnicas.
  Por otro lado, la población de la tabla de hash implicará la mayor parte de la complejidad del programa, mientras que buscar luego las muestras de la hebra madre en nuestra tabla de hash implicará una complejidad mínima, casi despreciable ya que con solo buscar en la tabla encontraremos una cantidad finita de hebras que comparar con nuestras hebras de ejemplo lo cual ocurre en reducidas iteraciones y en un tiempo casi inmediato.

- Una función de hash que podría servir para el problema planteado es codificar la hebra en base a las hebras de ejemplo. Para facilitar la redacción y comprensión me referiré a la hebra de prueba(Larga) como hebra y a las de ejemplo(Cortas) como samples.
  Para crear la tabla de hash partiremos por utilizar una función de hash para codificar las samples de la siguiente manera:

  – 1) Leemos cada sample asignamos un numero, partiendo del 1, a cada letra que forma parte de la sample. Este número será el mismo para letras iguales y cada vez que se encuentre una nueva letra este valor incrementará. De esta manera obtenemos algo así:

  $$Sample : ZZRXZDTTX = Codificación : 112314553$$

  – 2) Posteriormente se toma la codificación anterior y se suman todos los valores de este código de la forma que sigue:

  $$Codificación : 112314553 \longrightarrow 1+1+2+3+1+4+5+5+3$$

  – 3) Finalmente, usamos esta suma como key en nuestra tabla de hash la cual consistirá de una lista de listas para evitar efectos indeseados de colisiones. En el ejemplo anterior el key para este sample será:

  $$1+1+2+3+1+4+5+5+3 \longrightarrow 25$$

  Cabe destacar que la sample no es almacenada en la tabla de hash.

  Una vez que todas las samples han sido hasheadas empezamos a hashear la hebra. Para esto nos paramos en el primer caracter de la hebra y revisamos fragmentos de los largos de todas las samples. Es decir, iteramos sobre la cantidad de samples y por cada una revisamos su largo y lo utilizamos para revisar un fragmento de la hebra de este tamaño asi entonces, si tenemos 3 samples y la hebra ocurrirá lo siguiente

  $$Hebra : AABCADEECABBDACDBA$$

  $$Sample_1 : ZZRXZDTTX \longrightarrow Fragmento_1 : AABCADEEC$$

$$Sample_2 : ZRTTXRZ \longrightarrow Fragmento_2 : AABCADE$$

$$Sample_3 : XXRTD \longrightarrow Fragmento_3 : AABCA$$

Así, podemos notar que lo único que nos interesa en un inicio es el largo de cada sample. Una vez obtenido este fragmento lo codificamos y revisamos si el key es obtenido del fragmento es igual al de su respectivo sample, en caso de serlo esté fragmento es almacenado en la tabla de hash, en caso contrario no se almacena.
El almacenamiento dentro de la tabla de hash se hace según el key y el orden de llegada. El key indica la lista donde será almacenado el fragmento y el orden en está lista dependerá del orden de llegada del fragmento. Una vez codificados y almacenados todos los fragmentos importantes nos movemos un espacio adelante en la hebra. Así recorremos todos los caracteres de la hebra y codificamos todos los fragmentos significativos.

- La solución expuesta es incremental debido a que cada key puede ser fácilmente actualizado a uno nuevo según la necesidad, sin requerir cambiar la codificación del código anterior. Por ejemplo, si tenemos la codificación anterior:

  $$Sample : ZZRXZDTTX = Codificación : 112314553$$

  Cuyo key sabemos es 25, en caso de ser necesario un reordenamiento de la tabla de hash podemos operar directamente con el valor del key, siempre y cuando consideremos este cambio también para el sample según el cual se tomo la desición de almacenar el fragmento.

- Un Hash uniforme serviría para evitar la sobrepoblación de las listas de algunos keys ya que con un hash uniforme se cuenta con una distribución uniforme de los fragmentos dentro de la tabla, en caso de no ser uniforme pueden existir muchos fragmentos de distintas hebras dentro de una misma lista con un mismo key lo que aumenta el tiempo de ejecución en la búsqueda de un fragmento para algunas hebras.

- La complejidad en tiempo depende de la cantidad de samples, largo del sample y el largo de la hebra, ya que iteramos sobre estas cantidades en cada ejecución de lo cual se obtiene una complejidad en tiempo igual a $\mathcal{O}(N * l_i * S)$ donde N es el largo de la hebra, l_i el largo del sample i, donde i va de 1 a S,y S la cantidad de samples. En cuanto a la complejidad en memoria esta depende de la creación de la tabla de hash, la cual tiene 27*N espacios con listas de 1000 espacios cada una, por lo tanto su complejidad en memoria es $\mathcal{O}(27 * N * 1000)$. El 27 que acompaña al len se debe por a las 27 letras del abecedario común, lo que permite que existan 27 combinaciones de valores posibles para cada sample y fragmento.

NOMBRE: Sebastián Díaz Holmgren

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructuras de datos — 2' 2021

Tarea 2 – Respuesta Backtracking

- Backtracking es una buena opción para afrontar este ejercicio ya que nos permite identificar un output erróneo antes de finalizar todas las iteraciones lo que reduce inmensamente la cantidad de iteraciones totales y con esto la complejidad tanto de memoria como de tiempo.
  En este caso particular nos ayuda a no iniciar nuevamente con el tablero inicial tras cada error sino manteniéndonos desde el último paso correcto y seguir revisando.

- La complejidad sin backtracking es igual a $\mathcal{O}(64!)$ y con backtracking es igual a $\mathcal{O}(8^{64})$, ya que en caso sin backtracking deben probarse todas las posibilidades desde el inicio, en cambio con backtracking deben probarse una por una corrigiendo los errores desde la ultima falla.

- Para mi programa utilicé 3 podas:

  – Por imposibilidad de movimiento: En vez de probar todos los movimientos posibles del caballo, este probará solos los que están dentro de su rango de movimiento, evitando así segfaults y saltos erróneos.

  – Por casilla ocupada: Si el movimiento que realizará el caballo aterrizará en una casilla numerada con un número que no corresponde a su salto entonces este movimiento no es incluido en sus movimientos posibles para de esta manera evitar todos los movimientos que estarían relacionados a este, los cuales claramente serían erróneos.

  – Por número de salto: Si el número del movimiento se encuentra dentro del tablero inicial entonces los movimientos posibles se restringen únicamente al movimiento donde está la casilla con esa numeración, en caso de no ser posible este movimiento no hay movimientos posibles y por tanto esa rama queda cortada. De esta manera se evita revisar una rama claramente errónea.
    Además, para revisar que la suma final sea correcta, se revisa en el último paso que se cumpla esta condición. Se consideró incluir esta revisión en la búsqueda de pasos, sin embargo, esta acción aumentó enormemente el tiempo de ejecución por lo que se decidió dejar al final.

- La complejidad en tiempo en el caso sin podas es igual a $\mathcal{O}(8^{64})$ debido a que existen 8 pasos posibles para 64 pasos totales. Considerando las podas, principalmente la por número de salto tendremos $\mathcal{O}(8^{64-N})$ siendo N el número de casillas inicialmente numeradas. Si luego consideramos la poda por casilla ocupada y por imposibilidad de movimiento tendremos $\mathcal{O}((M_i)^{64-N})$ donde M_i es la cantidad de movimientos posibles del paso i, con i $\in$

  $$0, 64$$

  .

  En cuanto a la complejidad en memoria esta es fija en $\mathcal{O}(64)$ ya que solo se necesitan 64 casillas para guardar toda la información del programa. En el caso de mi programa son $\mathcal{O}(2 * 64)$ debido a que mantengo una copia del tablero inicial para poder corregir los movimientos erróneos.