Proyecto 1
Sketches Para Estimación de Similitud Entre Genomas

Tomás Ying-Kit Contreras Kong

José Francisco Cortés Pizarro

26 de septiembre de 2024

Link a repositorio de GitHub.

A través del presente documento, se explica cómo se llevaron a cabo las implementaciones de *hyperloglog* a través de C++, con el fin de aproximar la cardinalidad de distintas secuencias de nucleótidos de ADN pertenecientes a distintos genomas. Adicionalmente, con esta aproximación, calculamos el índice Jaccard de las cardinalidades, que indicará el nivel de semejanza o diversidad en cada par de genomas. Finalmente, resolviendo el error relativo medio y error absoluto medio entre cada Jaccard aplicado al sketch y el Jaccard real, se tendrá el margen de error de la implementación.

1. Implementar Hyperloglog

Archivos Involucrados: *hyperloglog.cpp*, *smhasher_copy*.

hyperloglog es una función cuyos parámetros son el nombre del archivo a leer y un entero sin signo que servirá de semilla para la función hash. Devuelve una estructura *map<unsigned int, unsigned int>* que representa al sketch asociado a la cardinalidad del texto. Para la implementación de *hyperloglog*, se usó la función hash *MurmurHash3* de *smhasher*.

Las variables a utilizar en la función *hyperloglog* son:

- ***map<unsigned int, unsigned int>* sketch**. Como su nombre indica, es el sketch que se desarrollará a través del procedimiento. Su estructura se devolverá al momento de terminar.
- ***unsigned int* hashValue**. Al aplicar el hash *MurmurHash3* en el *k-mer* actualmente seleccionado, se guardará su valor en esta variable.

- ***unsigned int* sketchKey**. La llave a usar en el sketch. Su valor es igual a los primeros p bits de *hashValue*
- ***unsigned int* sketchValue**. El valor a guardar en el sketch. Es igual a la posición del primer bit 1 de izquierda a derecha, luego de los primeros p bits de *hashValue*.
- ***unsigned int* randomSeed**. Valor elegido al azar cada vez que se ejecuta el código *main.cpp* a través de la función *rand()*. Usado en el hash *MurmurHash3*.
- ***int* p**. Valor usado para la asignación de *sketchKey* y *sketchValue* a través de los primeros p bits de *hashValue*. Se asigna previo a la ejecución del código. Cambiar esta variable va a cambiar el error estándar de estimación, la cual se calcula de la siguiente manera:

  $$\frac{1,04}{\sqrt{2^p}}$$

  Se nos pide un error estándar de estimación de aproximadamente 1 %, por lo que se tiene el siguiente valor de p:

  $$0,01 = \frac{1,04}{\sqrt{2^p}} \rightarrow \sqrt{2^p} = 104 \rightarrow 2^p = 10816 \rightarrow p = 13{,}4008794363 \rightarrow p \approx 14$$

- ***bool* firstIteration**. Variable auxiliar que sirve para ordenar la primera iteración de *k-mer* que se haga. Al ser falsa, el iterador agregará k letras al mer. Por otro lado, si es verdadera, se eliminará la letra en el primer índice de *mer* a través de *erase()*, y se agregará la siguiente letra del stream al final.
- ***string* mer**. String que sirve para analizar la cardinalidad del stream. A través de un iterador, revisa un pedazo de stream de tamaño k, o, en nuestro caso, *merSize*. Luego, al String entero se le aplicará el hash *MurmurHash3*, cuyo resultado será posteriormente guardado en *hashValue*.
- ***int* merSize**. Tamaño de los *k-mers*. Asignado previo a la ejecución del código. Se pide que sea de valor 20 y luego 25.
- ***ifstream* inputGenomeFile(fileName)**. Objeto ifstream para leer el archivo fileName. Si el archivo no es abierto con el nombre dado, *hyperloglog* devolvera un *sketch* vacío.
- ***string* firstLine**. Todos los archivos entregados tienen una priemra línea la cual introduce al genoma estudiado. Para no confundirlo con caracteres del genoma, se usa *firstLine* para guardar e imprimir esta información.
- ***istream_iterator<char>* iit(inputGenomeFile)**. Si es que se logró abrir un archivo con el *fileName* dado, se crea un iterador que lee las líneas luego de la primera.
- ***istream_iterator<char>* eos**. End Of Stream. Iterador nulo que sirve para compararlo con el iterador a usar. Si son iguales, significa que se llegó al final del stream.

A continuación, se explicará cada sección del código según su fin.

```
ifstream inputGenomeFile(fileName);
if(!inputGenomeFile.is_open()){
    return sketch;
}
```

Al momento de llamar a la función *hyperloglog*, se requiere el parámetro fileName. Si fileName no abre ningún archivo en el directorio actual, el sketch se retorna vacío.

```
getline(inputGenomeFile, firstLine);
cout << firstLine << '\n';

advance(iit, 1);
```

Cada uno de los archivos de genoma entregados tiene una primera línea la cual tiene una pequeña descripción. Esta sección de código nos permite atrapar esa línea e imprimirla, de esta manera, no se mezcla con el trabajo que tiene que hacer el iterador. Adicionalmente, se hace un avance para ignorar el signo > que contienen los archivos antes del stream.

Las siguientes secciones de código están dentro de un *while(true)*, y, por tanto, se compilarán hasta que encuentren un *break*.

```
    if(firstIteration){
        firstIteration = false;

        mer.push_back(*iit);
        for(int i = 1; i < merSize; i++){
            advance(iit, 1);
            if(iit != eos){
                mer.push_back(*iit);
            } else {
                break;
            }
        }

    } else {
        mer.erase(mer.begin());

        advance(iit, 1);
        if(iit != eos){
            mer.push_back(*iit);
        } else {
            break;
        }
    }
```

Si es que es la primera iteración del *while*, se insertan *merSize* elementos del archivo al string *mer*. Caso contrario, se elimina el primer elemento de *mer* y se le inserta al final el siguiente caracter del archivo. Cada vez que se hace una inserción, se avanza un espacio

en el archivo. Si en algún momento, en ambos casos, se llega a *eos* (esto es, el iterador *iit* es igual a un iterador nulo), se saldrá del *while*.

```
MurmurHash3_x64_128 (&mer, sizeof(mer), seed, &hashValue);
```

A través de la función hash *MurmurHash3* en el *mer* actual, se obtendrá un valor de hash que se guardará en *hashValue*.

```
sketchKey = (hashValue >> (32 - p)) + 1;
```

La llave a usar en el sketch para el *mer* actual será igual al valor representado por los primeros *p* bits de *hashValue*.

```
sketchValue = clz(hashValue << p) + 1;
```

El valor a ingresar en el sketch será igual a la posición de izquierda a derecha del primer 1 luego de los primeros *p* bits de *hashValue*. Destacar que *clz* es la función *__builtin_clz()*.

```
if(sketch.count(sketchKey) > 0){
    if(sketch[sketchKey] < sketchValue){
        sketch[sketchKey] = sketchValue;
    }
} else {
    sketch[sketchKey] = sketchValue;
}
```

Finalmente, usando el valor de *sketchKey*, se revisa si es que en el sketch ya hay un elemento usando la misma llave. De ser así, se compara el valor guardado con el valor de *sketchValue*, y se guarda aquel que sea el mayor. Por otro lado, si *sketchKey* no tiene colisión, se guarda el valor de *sketchValue* con su llave.

Al instante de romperse el bucle *while*, la función devolverá el valor de sketch que guardó hasta ese momento.

2. Implementar similitud de Jaccard

Archivos Involucrados: *jaccard.cpp*, *realJaccard.cpp*.

jaccard es una función cuyos argumentos constan de dos mapas de enteros que representan el 'genoma A' y el 'genoma B'.

```
float jaccard_function(map<int unsigned, unsigned int> GenA, map<unsigned int, unsigned int> GenB)
```

La función luego lee el mapa de ambos genomas y obtiene la cardinalidad de cada uno sumando sus valores.

```
        map<unsigned int, unsigned int> GenAoB;
        int cardA = 0;
        int cardB = 0;
        int cardAoB = 0;
        for(auto p: GenA)
            cardA += p.second;

        for(auto p: GenB)
            cardB += p.second;
```

Una vez obtenidas las cardinalidades se crean dos iteradores, uno para cada sketch de genoma. Se escoge el genoma con la mayor cardinalidad para usar como punto de término a la iteración, y luego se construye la conjunción lógica entre ambos genomas. Se inicía un *while* el cual terminará al llegar al final del map del genoma con mayor cardinalidad. En cada iteración, se comparará la llave del mapa del iterador actual y todas las llaves del otro mapa. De haber coincidencia, se inserta al mapa de unión la llave coincidente y el valor mayor entre el valor del genoma A o el genoma B. De no haber coincidencia, se inserta la llave y valor del iterador actual al mapa de unión. Se avanza el iterador y se sigue el bucle.

El siguiente bloque de código muestra lo que sucede cuando la cardinalidad del genoma A es mayor al genoma B. Cabe destacar que si lo contrario es cierto, se hará exactamente el mismo procedimiento, pero cambiando los valores de *GenA* a *GenB* e *itA* a *itB* y viceversa.

```
if(cardA>cardB){
    while(itA != GenA.end()){
        breakLoopFlag = false;
        map<unsigned int, unsigned int>::iterator itB = GenB.begin();

        while(itB != GenB.end()){
            if(itA->first == itB->first){

                if(itA->second > itB->second){
                    GenAoB.insert(pair<unsigned int, unsigned int>(itA->first, itA->second));
                } else {
                    GenAoB.insert(pair<unsigned int, unsigned int>(itA->first, itB->second));
                }

                breakLoopFlag = true;
                break;
            }

            advance(itB, 1);
        }
        if(!breakLoopFlag)
            GenAoB.insert(pair<unsigned int, unsigned int>(itA->first, itA->second));
        advance(itA, 1);
    }
} else {
    ...
}
```

Una vez construido el mapa de unión, se obtiene su cardinalidad sumando sus valores.

```
for(auto p: GenAoB)
    cardAoB += p.second;
```

Cuando todas las cardinalidades son obtenidas, se realiza la operación Jaccard y se retorna su valor.

```
float Jaccard = (cardA + cardB - cardAoB) / cardAoB;

return Jaccard;
```

Luego, para encontrar el jaccard real entre genomas, se hace el mismo procedimiento, pero en vez de entregar sketches a la función se entregan los nombres de los archivos. De esta manera, se hacen dos mapas con atributos <*char, int*> que siguen las mismas características que GenA y GenB en *jaccard.cpp*. Luego, la unión se construye de manera similar, usando un mapa <*char, int*> y comparando las llaves de los mapas anteriormente construidos para luego sumar sus valores y calcular Jaccard.

3. Evaluación experimental

Archivos involucrados: *main.cpp*, *hyperloglog*, *jaccard realJaccard*, archivos tipo *FNA*, *smhasher_copy*.

Se tiene que tener en cuenta que la evaluación experimental puede variar cada vez que se compila el código, ya que los valores de hash pueden causar menos o más colisiones según el valor de *randomSeed*. Se consideraron las siguientes secuencias de genomas, y se les llamará de la siguiente forma a través de la documentación:

1. GenA = GCF_000331305.1_ASM33130v1_genomic.fna
2. GenB = GCF_000959725.1_ASM95972v1_genomic.fna
3. GenC = GCF_000373685.1_ASM37368v1_genomic.fna
4. GenD = GCF_000583735.1_ASM58373v1_genomic.fna
5. GenE = GCF_001189295.1_ASM118929v1_genomic.fna

Para la primera compilación del código, se considera un *k-mer* de tamaño $k = 20$. Recopilando los resultados, se declara las cardinalidades de cada genoma y su unión usando *hyperloglog* en el Cuadro 1

Luego, usando los datos mencionados anteriormente, se hace un calculo del Jaccard aproximado usando el principio de inclusión/exclusión, y se recopilan los resultados en el Cuadro 2

	GenE	GenD	GenC	GenB	GenA
GenA	58456	58352	66927	81675	53348
GenB	75566	75366	80507	72922	
GenC	55275	55210	49807		
GenD	30420	24760			
GenE	25081				

Cuadro 1: Cardinalidad en cada genoma individual y su unión con otros genomas con $k = 20$

	GenE	GenD	GenC	GenB
GenA	0.341676	0.338566	0.541306	0.546005
GenB	0.296919	0.296102	0.524451	
GenC	0.354826	0.350607		
GenD	0.638429			

Cuadro 2: Jaccard aproximado usando *hyperloglog* con $k = 20$

Estos datos se pueden comparar con el valor Jaccard real de cada par de genomas en el Cuadro 3

Con ambos, podemos calcular el Error Relativo Medio (ERM) y Error Absoluto Medio (EAM) de esta iteración, quedando así con los valores de 0,397729 y 0,307917 respectivamente.

Luego, se hace el mismo proceso, pero cambiando el valor de k en *hyperloglog* a $k = 25$. El resultado de cardinalidad en los genomas es expuesto en el Cuadro 4

Al igual que la iteración anterior, comparamos dos genomas para encontrar su Jaccard aproximado, resultados los cuales están descritos en el Cuadro 5

Para calcular los valores de ERM y EAM, se usarán los mismo valores del Cuadro 3. De tal menera, el ERM para $k = 25$ sería de 0,278462 y el EAM sería de 0,215085.

Finalmente, comparando con los valores de error anteriores, se puede concluir que con un *k-mer* con $k = 25$ se tendrán sketches más similares a la cardinalidad comparado a usar $k = 20$, siendo la diferencia de errores, con Jaccard aproximado y Jaccard real, de $0,119267$ para ERM y de $0,092932$ para EAM.

	GenE	GenD	GenC	GenB
GenA	0.537104	0.562687	0.505484	0.559035
GenB	0.781494	0.854478	0.901978	
GenC	0.773922	0.845646		
GenD	0.914586			

Cuadro 3: Jaccard real de cada par de genomas

	GenE	GenD	GenC	GenB	GenA
GenA	79105	67822	62008	59915	37106
GenB	82059	72198	67200	50750	
GenC	82882	73402	53158		
GenD	85153	60657			
GenE	74860				

Cuadro 4: Cardinalidad en cada genoma individual y su unión con otros genomas con $k = 25$

	GenE	GenD	GenC	GenB
GenA	0.415410	0.441464	0.455683	0.466344
GenB	0.530728	0.543076	0.546250	
GenC	0.544581	0.550571		
GenD	0.591453			

Cuadro 5: Jaccard aproximado usando *hyperloglog* con $k = 25$