SARSCOV-HIERARCHY PROJECT

Universitat de Lleida

Enginyeria informàtica

Víctor Martínez Montané --- 78100640T

Francisco Manuel Martin Rubio --- 48057095K

Introducció

La major part d'aquest projecte està desenvolupat en Python. Per a la part de l'alineament de seqüencies hem utilitzat el llenguatge Rust implementant també multiprocessament per a fer més ràpida l'execució.

Podeu trobar la documentació tant del codi de Python com de Rust al link que trobareu al fitxer readme del repositori.

Execució del programa

Per a poder executar aquest projecte es necessari importar els fitxers sequences.csv i sequences.fasta. Per a facilitar aquest procés hem inclòs a la carpeta *resources* un zip amb els fitxers amb els quals s'han realitzat les proves.

Seguidament es necessita instal·lar les dependències especificades al fitxer requeriments.txt i ja queda tot preparat per a l'execució.

Es requereix una versió de Python igual o superior a la 3.6.

```
pip install -r requeriments.txt
python src/main/sarscovhierarchy.py <directori_data>
```

Preprocessament

Aquesta primera part del projecte està implementada en python i és aquí on prenem les decisions sobre com es tractaran les dades en tot el programa.

Per a la implementació de la lectura i tractament dels fitxers *.fasta* i *.csv*, vam decidir iniciar la implementació de dos objectes: FastaMap i CsvTable.

L'objecte FastaMap ens ajuda amb el tractament de les dades del fitxer fasta com el propi nom indica, aportant les funcionalitats bàsiques que s'espera d'un diccionari de Python però afegint a més la resta de mètodes requerits per al projecte.

Per altra banda, l'objecte CsvTable ens ajuda a tractar amb les dades del fitxer csv representades com una llista de diccionaris (cada diccionari representa una fila de la taula). De la mateixa manera, aquest objecte es troba dotat de funcionalitats que esperaríem trobar en una taula d'aquest tipus (Poder iterar sobre els elements, indexar per files, agafar les dades de les columnes... etc.)

Desenvolupament de l'algoritme

L'objectiu d'aquesta part del projecte es basa en realitzar el que nosaltres vam interpretar com un filtratge de les dades de la taula csv.

L'algorisme que proposem es prou senzill com per obtenir els resultats iterant un sol cop totes les files. El primer pas es basa en agrupar totes les mostres per països amb un diccionari. La clau d'aquest diccionari es el país a agrupar i el valor es una llista de tuples. Cada tupla conté 2 valors: L'index de la fila a la que correspon la mostra i la mida de la mostra.

Un cop hem agrupat les mostres d'aquesta manera, seleccionem la mostra de mida mediana mitjançant l'algorisme *Quick Select* per cada llista de tuples del diccionari de països creat anteriorment.

D'aquesta manera aconseguim agafar el valor medià de cada llista (El qual recordem que es basa en una tupla *(index_fila, mida)*), i crear una nova llista de diccionaris agafant de la taula original les files corresponents als index de cada tupla.

Com que l'objecte CsvTable permet crear una nova instancia a partir d'una llista de diccionaris, podem retornar sense problema un nou objecte CsvTable!

Complexitat de l'algoritme

Aquest algoritme té un cost O(n^2) en el pitjor dels casos, en el millor té un cost O(n). A que es degut això? Doncs bé, primer de tot recorrem un sol cop totes les mostres per construir el diccionari de països, fins aquí tenim O(n).

Aleshores hem de recórrer cada país del diccionari seleccionant la mostra de mida mitjana amb l'algoritme *Quick Select*, el qual té un cost de O(n^2) en el pitjor dels casos i un cost de O(n) en el millor.

Imaginem que només hi ha un país, per tant el diccionari només té una llista amb totes els mostres, el cost total en el pitjor dels casos quedaria així:

```
O(n + n^2) => O(n^2)
```

I en el millor dels casos quedaria:

```
O(n + n) => O(2n) => O(n)
```

Per molts països que hi hagi, els costos sempre seran iguals, perquè per exemple si tinguéssim 4 països amb les mostres repartides equitativament entre cada país, el cost del pitjor cas seria:

```
O(n + (n/4)^2 * 4) => O(n + (n^2)/16 * 4) => O(n^2)
```

Alineament de seqüencies

Aquesta part del projecte l'hem implementat amb Rust. El motiu d'aquest canvi de paradigma es deu a l'eficiència, control i seguretat que ens aporta aquest llenguatge sobre la gestió de la memòria a l'hora de realitzar l'alineament de seqüencies.

Algoritme Needleman Wunsch

Per a implementar aquesta part del projecte hem escollit l'algoritme de *Needleman Wunsch*. Tot i que no és molt complex en quant a procediment, resulta tenir un cost molt elevat tant en memòria com en temps d'execució degut a com tracta les dades.

L'objectiu final es basa en calcular la distancia entre 2 seqüencies. Per aconseguir-ho, l'algoritme mencionat anteriorment realitza els següents passos:

- Defineix 3 valors per a les següents situacions: GAP (Cal afegir o eliminar un símbol), MISMATCH (2 símbols són diferents) i MATCH (2 símbols són iguals). En el nostre cas el valor per a MATCH es 0, MISMATCH 1 i GAP 2.
- Crea una matriu *(N + 1) * (M + 1)* on N i M representen la llargada de cada seqüencia.
- Recorre la primera fila i columna de la matriu contant cada casella com un GAP.
- Recorre la resta de la matriu (N * M caselles) comparant així cada símbol de la 1a seqüencia amb cadascun de la 2a.

  Introdueix en cada casella el valor òptim a col·locar a partir dels anteriors de la següent manera:

```
// c1 i c2 son els simbols a comparar
// check_match retorna 0 si c1 i c2 són iguals o 1 si son diferents
let fit = matrix[(i, j)] + check_match(c1, c2);
let delete = matrix[(i, j + 1)] + GAP;
let insert = matrix[(i + 1, j)] + GAP;
let min_val = min(min(fit, delete), min(fit, insert));
matrix[(i + 1, j + 1)] = min_val;
```

L'idea per a implementar aquest algoritme es basa en el concepte de *programació dinàmica*. Diem això ja que per a trobar el resultat final, primer necessitem trobar el resultat de tots els possibles casos anteriors travessant tota la matriu. Afrontem el problema que es planteja (trobar el resultat per a $N * M$ caselles), trobant primer el resultat per a tots els grups de 2*2 caselles.

Complexitat de l'algoritme

Un cop coneguda la implementació de l'algoritme de **Needleman-Wunsch**, ja podem parlar de la seva complexitat.

Tal com hem vist, per a realitzar l'alineament es necessita crear una matriu de mida (N + 1) * (M + 1) on N i M representen la llargada de les 2 seqüencies. Seguidament el que fem es recórrer tota la matriu exceptuant la primera fila i la primera columna, que ja s'han recorregut anteriorment.

Per tant podem concloure en que la complexitat d'aquest algorisme es **O(N * M)**.

Anàlisi experimental

Cost en memòria

Com hem dit al principi d'aquest apartat, l'algoritme implementat consumeix molta memòria quan comencem a treballar amb seqüencies grans.

En el nostre cas hem intentat reduir al mínim aquest impacte aprofitant la facilitat que ens aporta el llenguatge ***Rust*** en quant a gestió de la memòria.

I perquè diem que aquest algoritme consumeix molta memòria? Doncs bé, tot recau en el fet d'haver de crear una matriu de mida **(N + 1) * (M + 1)**.

Si pensem en seqüencies petites de fins a 100 caràcters (Per posar un exemple), la matriu arriba a tenir unes 101 * 101 cel·les, es a dir, 10201 cel·les.

Anem a seguir amb l'exemple donat. Si a cada casella introduïm un nombre enter de 64 bits (8 bytes), la matriu sencera ocupara en memòria 10201*8 bytes, es a dir, 0.08 MB.

Sembla poc oi? Doncs anem a fer els càlculs per a seqüencies reals:

Una seqüencia RNA pot arribar a tenir una llargada aproximada de 30000 caràcters. El pitjor dels casos (Si comparéssim 2 seqüencies d'aquesta llargada), suposaria crear una matriu de 30001 * 30001 cel·les, es a dir, 900060001 cel·les.

Si l'emplenem amb nombres enters de 64 bits, la matriu arribarà a ocupar en memòria **7.2 Gygabytes**.

Ara bé, com hem afrontat nosaltres aquest problema fins el punt de poder arribar a fer multiprocessament per a realitzar més d'una comparació a l'hora?

Primer de tot ens vam donar compte de que no necessitàvem enters de 64 bits. De fet canviant els valors MATCH, MISMATCH i GAP de 1, -1 i -2 respectivament a 0, 1 i 2, podíem deixar d'utilitzar nombres amb signe. I no només això, sinó que si les seqüencies poden arribar a tenir fins a 30000 caràcters, la matriu mai arribarà a emmagatzemar nombres que superin a 60000 (En el cas que contéssim tot GAPs). Per tant, hem passat d'utilitzar enters de 64 bits a utilitzar nombres sense signe de 16 bits.

Si tornem a realitzar els càlculs, en el pitjor dels casos una matriu ocuparà 900060001 * 2 bytes, es a dir, **1,8 Gygabytes**.

Temps d'execució

Gràcies a l'optimització de la gestió de la memòria explicada anteriorment, el temps d'execució s'ha vist reduït en picat.

Això és deu a la reducció del nombre de dades amb les quals s'ha de treballar al *Heap* (Zona de memòria a la qual suposa un cost elevat accedir en comparació al *Stack*), ja que és on s'emmagatzema cada matriu que es crea.

Actualment treballant amb valors del tipus **u16**, una sola comparació entre 2 mostres mitjançant alineament de seqüencies tarda de mitjana 1.5 segon.

Anteriorment, treballant amb valors del tipus **i16**, una comparació podia arribar a tardar uns 3.5 segons de mitjana, i treballant amb valors del tipus **i64**, en un *PC* amb mínim 8 GB de RAM, podia tardar uns 7 segons.

Classificació

Per acabar hem decidit classificar les mostres mitjançant una tècnica de **Clustering** anomenada **Hierarchical Clustering**.

El procediment es senzill:

- Comencem tractant cada mostra a avaluar com un únic *Cluster*.
- Cerquem les 2 mostres que més s'assemblen (Es troben més a prop).
- Ajuntem les 2 mostres seleccionades en un únic *Cluster*.
- Repetim el procés fins acabar tenint un sol *Cluster*.

Podem veure el procediment de manera gràfica seguint aquest link.

Pel que fa a l'obtenció de les distancies entre mostres, ens basem en l'algoritme d'alineament de seqüencies analitzat en l'apartat anterior i, simplement, comparem les seqüencies totes entre totes (Si acabem tenint 44 mostres després de realitzar el 'filtrat' del primer apartat, acabem realitzant 946 comparacions).

Aquest últim pas també l'hem implementat amb *Rust* i aquesta vegada el motiu va més enllà de la gestió de la memòria.

Complexitat de l'algoritme

Aquest algoritme presenta una complexitat **O(n^2)**, on **n** representa la quantitat de mostres a tractar.

Primer de tot necessitem comparar les mostres totes entre totes, fet que ja presenta una complexitat **O(n^2)**.

Un cop hem realitzat totes les comparacions busquem repetidament les 2 mostres més properes, reduint en 1 la quantitat d'elements a tractar a cada volta del *loop*. Per tant, la complexitat d'aquest pas és:

```
O((n*(n - 1))/2) => O(n^2)
```

Per tant, la complexitat final d'aquest algoritme és:

```
O(n^2 + n^2) => O(n^2)
```

Anàlisi experimental

Degut a les optimitzacions que comentem en l'apartat anterior pel que fa a la memòria que ocupa una comparació, hem volgut experimentar realitzant multi-processament i el resultat es simplement espectacular.

Si una sola comparació tarda de mitjana 1.5 segons, processar 946 comparacions tarda al voltant de 1400 segons. Ara bé, i si realitzem més d'una comparació a l'hora?

Mitjançant el poder de la llibreria **Rayon** de *Rust* hem implementat d'una manera molt senzilla un sistema concurrent per a realitzar més d'una comparació a l'hora. *Rayon* ens ofereix mètodes com *par_iter* el qual ens retorna un iterable que dividirà les tasques a executar en diferents *threads*, i sobre aquest podem utilitzar qualsevol funció que utilitzaríem sobre un iterable comú.

El codi implementat s'assembla al següent:

```
v.par_iter().map(|s1, s2| {
        needleman_wunsch(s1, s2),
})
```

Som conscients que no podem utilitzar aquest mètode per a qualsevol *PC* ja que es pot produir un error en quant a espai disponible en memòria.

Per a solucionar aquesta problemàtica, comprovem la memòria disponible del sistema gràcies a la llibreria *psutils* i, havent calculat prèviament l'espai màxim que pot ocupar realitzar una comparació, calculem el nombre de *threads* màxims que s'haurien d'utilitzar. Per seguretat hem assignat el valor màxim de *threads* a 4 degut a que ja ens dona uns resultats extraordinàriament bons.

Nosaltres hem pogut testejar el rendiment d'aquesta millora amb un *PC* de fins a 6 *threads* i els resultats han sigut els següents:

```
+ 1 THREADS: 1400 segons aprox.
+ 2 THREADS: 693 segons aprox.
+ 3 THREADS: 500 segons aprox.
+ 4 THREADS: 390 segons aprox.
+ 5 THREADS: 340 segons aprox.
+ 6 THREADS: 300 segons aprox.
```