

UNIVERSITAT_{DE} BARCELONA

Pràctica 3. Anàlisi de codi

Javier González Béjar
Alejandro Guzmán Requena

22 octubre 2022

ÍNDIX

1	Introducció	1
2	Treball realitzat al laboratori	1
1.	Execució del codi	1
2.	Aeroports amb més destinacions	1
3.	Temps d'execució	2
4.	Nombre de vols totals	2
5.	L'eina Callgrind	3
6.	Error amb Valgrind	4
3	Conclusions	5

1 INTRODUCCIÓ

La tercera pràctica de l'assignatura és un preàmbul al treball a realitzar a les darreres dues pràctiques de l'assignatura, dedicades a la concurrència.

L'objectiu és analitzar el codi proporcionat sobre el nombre de vols que hi ha hagut als anys 2007 i 2008 entre diferents aeroports dels Estats Units d'Amèrica, i sobretot entendre-ho per a poder replicar-ho de forma concurrent posteriorment en les següents pràctiques.

2 TREBALL REALITZAT AL LABORATORI

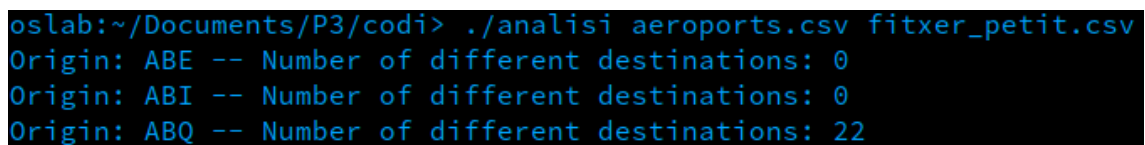
1. Execució del codi

Pregunta: Executar el codi i provar el seu funcionament. Observar que l'aplicació imprimeix per pantalla 303 aeroports d'origen. Identifiqueu aeroports coneguts? Nova York? Atlanta? Comenteu els resultats.

Executem el codi que ens proporcionen utilitzant la següent comanda:

```
./analisi aeroports.csv fitxer_petit.csv
```

Ens apareix un gran llistat d'aeroports identificats amb el seu codi IATA i el nombre de destinacions diferents que prenen amb el fitxer utilitzat passat per paràmetre.



```
oslab:~/Documents/P3/codi> ./analisi aeroports.csv fitxer_petit.csv
Origin: ABE -- Number of different destinations: 0
Origin: ABI -- Number of different destinations: 0
Origin: ABQ -- Number of different destinations: 22
```

Figura 2.1: Part de l'output de l'execució del codi proporcionat

Si observem detingudament el llistat d'aeroports que s'imprimeixen podem identificar alguns de coneguts, com l'Aeroport Internacional del Vall Lehigh (ABE), l'aeroport d'Atlanta o Aeroport Internacional Hartsfield-Jackson (codi IATA ATL) o un dels tres aeroports de New York conegut com Aeroport Internacional John F. Kennedy (codi IATA JFK).

2. Aeroports amb més destinacions

Pregunta: Quins són els aeroports amb més destinacions? Com us ho feu per saber-ho? Hi ha alguna comanda que us ho permeti saber?

Els tres aeroports amb més destinacions són:

- L'aeroport de Las Vegas (**LAS**): 54 destinacions diferents
- Chicago Midway International Airport (**MDW**): 47 destinacions diferents
- Phoenix Sky Harbor Airport (**PHX**): 42 destinacions diferents

Per a saber-ho de forma senzilla, podem redirigir la sortida de l'execució del codi de manera que s'ordenin les fileres de l'arxiu pel valor final de cadascuna (cadena de text 8), que indica el nombre de destinacions diferents. Per això, fem la següent comanda:

```
$ ./analisi aeroports.csv fitxer_petit.csv | sort -k 8n
```

El número 8 indica la posició de la cadena a cada filera, i el 8n s'escriu perquè els números poden ser de més d'1 dígit. La k indica que la ordenació es fa a partir de la key desitjada.

```
Origin: PHX -- Number of different destinations: 42
Origin: MDW -- Number of different destinations: 47
Origin: LAS -- Number of different destinations: 54
```

Figura 2.2: Aeroports amb més destinacions a l'arxiu fitxer_petit.csv

3. Temps d'execució

Pregunta: Quan triga el codi executar-se sobre els fitxers 2007.csv o 2008.csv? Quina comanda es pot fer servir per mesurar el temps (real, usuari i sistema) d'un procés?

Per a mesurar el temps d'un script en bash utilitzem la comanda *time* procedida de l'script a executar:

```
$ time ./analisi aeroports.csv 2007.csv
```

El temps d'execució ha augmentat molt respecte a l'execució amb el fitxer petit, com era d'esperar degut al gran nombre de dades que contenen. L'execució, a part de mostrar els resultats dels aeroports, mostra el temps d'execució real, d'usuari i de sistema:

```
real    1m1.946s
user    0m58.315s
sys     0m3.558s
```

Figura 2.3: Temps d'execució de ./analisi aeroports.csv 2007.csv

```
real    0m57.952s
user    0m55.788s
sys     0m2.119s
```

Figura 2.4: Temps d'execució de ./analisi aeroports.csv 2008.csv

4. Nombre de vols totals

Pregunta: Quants vols emmagatzemen aquests dos fitxers (2007.csv i 2008.csv)? Amb quina comanda podeu esbrinar aquesta informació?

Hem mencionat que ambdós fitxers proporcionats (2007.csv i 2008.csv) tenen un nombre molt gran de fileres, però ara volem realitzar un recompte de quantes són. Per a realitzar aquesta tasca utilitzem la comanda *wc -l*, la qual realitza un recompte de les línies, ja que recordem que *wc* era un comptador i junt amb el paràmetre *-l* realitzava un recompte de les línies que estan presents al fitxer.

Llavors per executar aquesta comanda escrivim a consola:

```
$ wc -l 2007.csv
$ wc -l 2008.csv
```

I l'output d'ambdós fitxers és:

```
oslab:~/Documents/P3/codi> wc -l 2007.csv
7453216 2007.csv
oslab:~/Documents/P3/codi> wc -l 2008.csv
7009729 2008.csv
```

Figura 2.5: Nombre de línies dels fitxers 2007.csv i 2008.csv

En total n'hi han 14.462.945 línies entre els dos fitxers.

5. L'eina Callgrind

Callgrind és una eina que registra el nombre d'instruccions CPU que fan falta per executar cada funció, així com l'historial de crides entre les funcions d'un programa. Callgrind no mesura el temps real d'execució, sinó que només mesura el temps de CPU.

Ens han demanat realitzar un anàlisi de l'execució del codi proporcionat (analisi.c) mitjançant aquesta eina. Així, executem el següent:

```
$ valgrind --tool=callgrind ./analisi aeroports.csv
fitxer_petit.csv
```

Llavors se'ns crea un fitxer anomenat callgrind.out.<pid>, on <pid> és l'identificador del procés executat. Per a obrir l'arxiu escrivim a la consola *kcachegrind*.

```
$ kcachegrind
```

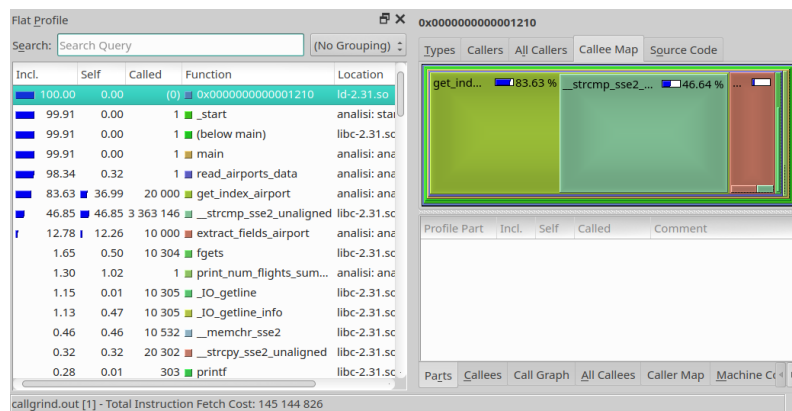


Figura 2.6: KCacheGrind

Se'ns obre l'aplicació de Callgrind amb informació en relació a l'execució del programa. La primera vista que es mostra és un llistat de funcions amb el percentatge i el temps d'execució en relació al total del programa analitzat.

Si cliquem en cada funció, a l'apartat de Call Graph, se'ns mostra un graf molt visual que indica l'historial de crides que inclou la funció seleccionada, per tant es pot veure clicant en una funció qualsevol que la crida el main, i després aquesta crida respectivament a altres funcions durant l'execució.

La primera funció en la que ens centrarem és el *main*, el qual és pràcticament el primer mètode que es crida al iniciar l'execució del programa. Realment la primera funció que es crida

és *_start*, després es crida a (*below main*) per realitzar els imports i l'assignació de memòria i després comença el programa amb el *main*, cridant els mètodes successors en l'execució, tal i com surt en el graf següent:

Dintre del *main*, el mètode que més triga amb diferència és el *read_airports_data*, donat que ha de llegir un fitxer d'una mida de mil·lions de fileres i actualitzar la matriu *num_flights* per saber quants vols n'hi han entre cada ciutat origen i destí. De fet, si executem el callgrind amb el fitxer petit i el 2007.csv podem notar una gran diferència en el nombre d'instruccions d'aquest mateix mètode, ja que estan llegint fitxers diferents i amb el fitxer petit realitza 146.000 instruccions, en canvi amb el 2007.csv realitza més de 3 milions.

Si analitzem el mètode *read_airports_data*, ens adonem que dintre torna a cridar a altres funcions, llavors podem veure que la funció amb la qual triga més temps és *gets_index_airports*, aquesta funció es crida aproximadament mig milió de cops dins del bucle de *read_airports_data*, per la qual cosa implica que el programa romandrà un temps considerable dins d'aquesta funció.

6. Errors amb Valgrind

Pregunta: El codi entregat no és net, sinó que té errors de programació. Els podeu identificar?

En primer lloc executem el codi amb la comanda que se'ns dona, redreçant la sortida cap a un fitxer:

```
$ valgrind ./analisi aeroports.csv fitxer_petit.csv
```

L'eina Valgrind ens permet analitzar els problemes de programació del codi. En aquest cas l'output ens dona el següent diagnòstic:

```
==2526== Memcheck, a memory error detector
==2526== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2526== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==2526== Command: ./analisi aeroports.csv fitxer_petit.csv
==2526==
==2526== Conditional jump or move depends on uninitialised value(s)
==2526==    at 0x400A3F: print_num_flights_summary (in /home/oslab/Desktop/P3-Analisi de codi (invalid encoding)/codi/analisi)
==2526==    by 0x400E30: main (in /home/oslab/Desktop/P3-Analisi de codi (invalid encoding)/codi/analisi)
==2526==
```

Figura 2.7: Error 1, Valors no inicialitzats

```
void **malloc_matrix(int nrow, int ncol, size_t size)
{
    int i;

    void **ptr;

    ptr = calloc(nrow, sizeof(void *));
    for(i = 0; i < nrow; i++)
        ptr[i] = calloc(ncol, size);

    return ptr;
}
```

Figura 2.8: Solució per l'error 1

Se'ns comenta en primer lloc un error que ve produït per valors no inicialitzats. Això és degut a l'ús de *malloc*. *Malloc* només en reserva espai per a les variables de la matriu, però no les inicialitza. Per a solucionar-ho, és suficient amb canviar el *malloc* per *calloc()*, que sí inicialitza els valors a 0.

Aquesta funció rep dos paràmetres: El nombre de blocs per a assignar memòria, i la mida de cada bloc (en bytes). A la imatge anterior podem veure com queda el codi després d'arreglar-ho.

```
==2526==
==2526== HEAP SUMMARY:
==2526==   in use at exit: 4,848 bytes in 2 blocks
==2526==   total heap usage: 613 allocs, 611 frees, 383,456 bytes allocated
==2526==
==2526== LEAK SUMMARY:
==2526==   definitely lost: 4,848 bytes in 2 blocks
==2526==   indirectly lost: 0 bytes in 0 blocks
==2526==   possibly lost: 0 bytes in 0 blocks
==2526==   still reachable: 0 bytes in 0 blocks
==2526==   suppressed: 0 bytes in 0 blocks
==2526== Rerun with --leak-check=full to see details of leaked memory
==2526==
==2526== Use --track-origins=yes to see where uninitialised values come from
==2526== For lists of detected and suppressed errors, rerun with: -s
==2526== ERROR SUMMARY: 91809 errors from 1 contexts (suppressed: 0 from 0)
```

Figura 2.9: Error 2, Memory leak

```
void free_matrix(void **matrix, int nrow)
{
    int i;

    for(i = 0; i < nrow; i++)
        free(matrix[i]);
    free(matrix);
}
```

Figura 2.10: Solució per l'error 2

El diagnòstic també esmenta una pèrdua de memòria del heap. Si revisem el codi ens adonem que a l'hora d'alliberar la memòria, només s'estan alliberant les adreces de cada filera de la matriu, de manera que falta alliberar la memòria reservada per la pròpia matriu. És possible solucionar-ho afegint un *free(matrix)* després del bucle.

3 CONCLUSIONS

Per a concloure ens agradaria destacar el repàs que s'ha realitzat a comandes útils de bash per a manipular i tractar-ne outputs de fitxers, a més de la utilització de les eines Valgrind i Callgrind per a diagnosticar errors de programació de codi i obtenir-ne informació de les instruccions CPU executades.

A nivell personal considerem que ha estat una pràctica principalment útil per a reforçar-ne conceptes ja estudiats, d'una dificultat no gaire elevada.

Estem, per això, satisfets amb la nostra dedicació i rendiment a la pràctica.