Universidade do Minho
Escola de Engenharia

Licenciatura em Engenharia Informática

LI3 - Relatório: 2ª Fase

GRUPO 77

António Pedro (a100821) João Magalhães (a100740) Rodrigo Gomes (a100555)

Ano Letivo de 2022/2023

Índice

Lista de Figuras

Lista de Tabelas

1 Introdução

Chegando ao fim do semestre, também o projeto realizado no âmbito da disciplina de Laboratórios de Informática III alcança o seu término. Servirá, portanto, o presente relatório para guiar o leitor pelo alcançado pelo grupo, bem como os métodos que o permitiram e os problemas que se impuseram no nosso percurso.

2 Modos de execução e estruturas de dados

Após a discussão relativa ao estado da primeira fase do projeto com os docentes responsáveis, o nosso *parser* sofreu de imediato uma grande reestruturação, visando melhorar aspetos de encapsulamento de dados, reutilização de código e modularização.

Começamos por reduzir o tamanho da função main em mais de 100 linhas, redistribuindo todo o trabalho feito pela mesma pelos módulos corretos, alcançando uma função main final de acordo com o aconselhado, onde somente se verifica o modo de inicialização do *parser* e se chama a função batch ou interativo.

```
int main (int argc, char* argv[])
{
    if(argc==1)
    {
        if (!strcmp("./programa-testes",argv[0])) testing();
        else interativo();
    }
    else if(argc==3)
    {
        batch(argv);
    }

    return 0;
}
```

Código 1: Função main

2.1 Batch

```
int batch(char* argv[])
{
    struct data* data = load_data(argv[1]);
    mkdir("Resultados", S_IRWXU);
    main_parse (argv[2], data->drivers, data->users, data->rides);
    free_mem(data);
    free(data);
    return 0;
}
```

Código 2: Função batch

De seguida, é feita a leitura, linha a linha e no módulo que trata cada tipo de dados(users.c , drivers.c e rides.c) dos ficheiros .csv dos utilizadores, condutores e viagens, respetivamente, através do caminho fornecido como argumento ao programa-principal. Aquando da leitura de uma linha, esta é verificada por funções do módulo verifica_input.c, de modo a concluir se se trata de uma linha válida, isto é, que deve ser aceite pelo programa. Sendo aceite, a partir da linha em questão é inicializada uma componente do catálogo do tipo de informação obtida. Caso a linha seja considerada inválida, é descartada de imediato e passa-se para a seguinte.

```
struct ride* init_ride(char* line_m)
{
    struct ride* ride = malloc (sizeof(struct ride));
    char* line = strdup(line_m);
    char* linef = line;
    char* ptr;
    int index=0;
    ride->is_valid=1;

    while((ptr=strsep(&line, ";\n")) != NULL && ride->is_valid)
    {
        switch (index)
        {
        //...
        }
    }
    free(linef);
    return ride;
}
```

Código 3: Excerto da função de parsing do ficheiro rides.csv

Na imagem acima mencionada é apresentado um excerto da função rides onde é realizada a leitura do ficheirs rides.csv. Para o ficheiros dos drivers e users o processo é análogo.

Estando terminada a leitura dos ficheiros, estão também os catálogos dos condutores, struct drivers, dos utilizadores, struct users e das viagens, struct rides completos e o programa está pronto a resolver qualquer query que seja imposta pelo ficheiro de texto fornecido como segundo argumento ao programa-principal.

2.1.1 Drivers

```
struct drivers
{
    struct dh* dh;
    struct sdarray* sdarray;
    struct sdarray* ctarray;
    char* ordered_by;
};
```

Código 4: Estrutura de dados drivers

A estrutura principal do módulo dos condutores, apresentada em cima, é constituída por um apontador para a estrutura dh, uma *hash table* , dois apontadores para estruturas sdarray, *arrays* dinâmicos da estrutura driver, estrutura imprescindível do catálogo de condutores que contém as informações de cada um, e uma string, ordered_by, responsável por indicar a cidade em que o programa se baseou na última vez que ordenou o ctarray.

```
struct driver
{
    char* id;
    char* name;
    int age;
    char* gender;
    char* car_class;
    char* city;
    char* acc_creation;
    char* acc_status;
```

```
    char* date_ride;
    struct med_score* med_score;
    int n_rides;
    double tot_profit;
    int is_valid;
};
```

Código 5: Estrutura de dados driver

```
struct med_score
{
    double main_med_score;
    struct city_score** city_score;
    int size;
    int max;
};
```

```
struct city_score
{
    char* city;
    double city_med_score;
    int city_n_rides;
    };
```

Código 6: Estruturas de dados med_score e city_score

Entrando em mais profundidade na drivers, comecemos por salientar a estrutura driver, em cima apresentada. Cada condutor tem a sua instância e é através da mesma que o programa obtém as informações que necessita para realizar as *queries* que venham a surgir. O seu único parâmetro não tão intuitivo é o apontador para a struct med_score, que é constituída pela avaliação média de todas as viagens do condutor (main_med_score) e por um array dinâmico das avaliações do mesmo condutor em cada cidade existente. Qualquer elemento deste array é, por sua vez, um *triple* com o nome da cidade, a avaliação média e o número de viagens realizadas na mesma, um city_score.

```
struct sdarray
{
    int size;
    int max;
    struct driver** Darray;
};
```

```
struct dh
{
    GHashTable* hash;
};
```

Código 7: Estruturas de dados sdarray e dh

Sabendo isto, torna-se bastante simples explicar as estruturas dh e sdarray. A primeira, uma hash table (neste caso, sendo proveniente da glib, uma GhashTable, criada com g_hash_table_new_full) de apontadores para struct driver como *value* e o id do mesmo como *key*. A segunda, um array dinâmico de apontadores para struct driver, ordenado de acordo com os critérios pedidos pelas queries.

2.1.2 Users

```
struct users
{
    struct uh* uh;
    struct suarray* suarray;
};
```

Código 8: Estrutura de dados users

A estrutura principal do módulo dos utilizadores, apresentada em cima, é constituída por um apontador para a estrutura uh, uma *hash table* e um apontador para a estrutura suarray, *array* dinâmico da estrutura user, estrutura crucial do catálogo de utilizadores que contém as informações de cada um.

```
struct user
{
    char* username;
    char* name;
```

```
    char* gender;
    int age;
    char* acc_creation;
    char* pay_meth;
    char* acc_status;
    char* date_ride;
    double med_score;
    int n_rides;
    double tot_losses;
    int distance;
    int is_valid;
};
```

Código 9: Estrutura de dados user

Aprofundando nos users, comecemos por salientar a estrutura user, supramencionada. Cada utilizador tem a sua instância desta estrutura e é através da mesma que o programa obtém as informações que necessita para realizar as *queries* que venham a ser objeto de resolução.

Entendendo isto, em semelhança ao módulo dos condutores, torna-se bastante simples explicar as estruturas uh e suarray. A primeira, uma *hash table* (GhashTable) de apontadores para struct user como *value* e o username do mesmo como *key*. A segunda, um array dinâmico de apontadores para struct user, ordenado de acordo com os critérios pedidos pelas queries.

```
struct suarray
{
    int size;
    int max;
    struct user** Uarray;
};
```

```
struct uh
{
    GHashTable* hash;
};
```

Código 10: Estruturas de dados suarray e uh

2.1.3 Rides

```
struct rides
{
    struct rh* rh;
    struct rh* ch;
    struct srarray* sr_male;
    struct srarray* sr_female;
    struct srarray* sr_tip;
};
```

Código 11: Estruturas de dados rides

A estrutura principal do módulo das viagens, apresentada em cima é constituída por dois apontadores para estruturas rh, *hash tables* e três apontadores para estruturas srarray , *arrays* dinâmicos da estrutura ride, estrutura fulcral do catálogo de viagens que contém as informações de cada uma.

```
struct ride
{
    char* id;
    char* date;
    char* driver_id;
    char* user_username;
    char* city;
    int distance;
    double score_user;
```

```
    double score_driver;
    double tip;
    double ride_cost;
    int is_valid;
};
```

Código 12: Estruturas de dados rides

```
struct foreach
{
    struct drivers* drivers;
    struct users* users;
    struct rh* ch;
};
```

Código 13: Estruturas de dados foreach

Olhando com mais atenção para a rides, comecemos por salientar a estrutura ride, supramencionada. Cada viagem tem a sua instância desta estrutura e é através da mesma que o programa obtém as informações que necessita para atualizar os dados dos catálogos dos condutores e utilizadores e realizar as *queries* que sejam pedidas. Tal atualização é otimizada com recurso à estrutura foreach, a qual permite que sejam passadas todas as estruturas que sofrem alterações após a leitura das viagens numa única iteração da função g_hash_table_foreach com o seguinte formato:

```
        g_hash_table_foreach(rh->hash,foreach_updt,foreach);
```

```
struct srarray
{
    int size;
    int max;
    struct ride** Rarray;
};
```

```
struct rh
{
    GHashTable* hash;
};
```

Código 14: Estruturas de dados srarray e rh

```
struct mpreco
{
    double price;
    int n_rides;
};
```

Código 15: Estruturas de dados mpreco

Assim sendo, torna-se bastante simples explicar as estruturas uh e srarray, que em nada diferem das suas "familiares" abordadas em cima. A primeira, uma hash table (GHashTable). Neste módulo existem duas exemplares, uma de apontadores para struct rides como *value* e o id da mesma *key* e outra de apontadores para a struct mpreco como *value* e uma string contendo o nome da cidade que corresponde a esse mpreco como *key*. A struct mpreco trata então de guardar o total gasto em viagens numa determinada cidade e o número total de viagens ocorridas na mesma. Por fim, srarray é um array dinâmico de apontadores para struct ride, ordenado de acordo com os critérios pedidos pelas queries. O tipo de ordenação praticada nestes arrays será abordada com mais precisão nos capítulos das queries.

2.2 Interativo

```
int interativo()
{
    system("clear");
```

```
4       struct data* data;
        int opt = 10;
6
        while (opt != 0)
8       {
            switch (opt)
10          {
            // ...
12          }
        }
14      system("clear");
        return 0;
16  }
```

Código 16: Excerto da função interativo

De forma a concluir este projeto, foi-nos proposta a realização de um modo interativo e, para tal, tentamos fazer com que o utilizador tenha uma experiência imersiva no mesmo. Inicialmente, neste modo, como no batch, com exceção à forma como é passada a pasta dos ficheiros csv, é realizado um parsing dos ficheiros de entrada e, depois de apresentadas as instruções para o bom funcionamente do mesmo, o usuário poderá fornecer os comandos de cada query e ter acesso aos outputs diretamente no terminal, com uma ferramenta de paginação para resultados bastante extensos. O utilizador pode ainda alterar o número de linhas a apresentar por página de acordo com a sua preferência.

3 Queries

Uma vez que o tratamento de dados é previamente feito, cabe ao módulo main_parser.c decompor os comandos contidos no ficheiro inputs.txt ou dados pelo utilizador via terminal e identificar as informações a apresentar para cada query. De forma a que se torne mais intuitivo compreender o mecanismo por detrás de cada uma passaremos a explica-las:

- Query 1
  - Estando os dados dos condutores e dos utilizadores guardados nas hash tables das estruturas dh e uh, respetivamente, utiliza-se o argumento do comando da query como *key* para obter as informações pedidas.
- Query 2
  - Guardados todos os condutores num array (estrutura sdarray) ordenado pelos parâmetros desta query, torna-se trivial imprimir os resultados da mesma.
- Query 3
  - Esta query tem como base um funcionamento análogo à query 2, diferindo no array a que se recorre, que passa a ser o correspondente da estrutura dos utilizadores, suarray.
- Query 4
  - Uma vez que todos os gastos numa cidade bem como o total de viagens da mesma estão previamente guardados numa hash table (da estrutura ch), basta apenas procurar o mpreco usando o nome da cidade como *key* e imprimir a divisão dos dois parâmetros do mesmo.
- Querys 5 e 6

- Recorrendo a uma estrutura auxiliar similar à foreach, de nome foreach_5_6 e com o mesmo propósito, percorrem-se as viagens, retirando-se das relevantes o seu preço e incrementando o número de viagens. Esta estrutura está adaptada para servir ambas as queries e a apresentaçao do resultado passa por uma divisão idêntica à da query 4.

- Query 7

  - De modo a resolver o desafio apresentado por esta query, começamos por verificar por que cidade está ordenado o array secundário das avaliações médias, ctarray. Se não for pela cidade pretendida, é realizada essa ordenação. De seguida, percorre-se o array, imprimindo as viagens relevantes.

- Query 8

  - Através de verificações de género de condutor e utilizador, são criados dois arrays de viagens (srarray), um para cada género (sr_male e sr_female). De seguida,o array do género dado pelo comando é percorrido e são verificadas as idades dos perfis dos intervenientes. Resta apenas imprimir todos os elementos do array que corresponderem ao pedido.

- Query 9

  - Guardadas todas as viagens nas quais se deu gorjeta inicialmente no srarray sr_tip, é apenas necessário ver se as mesmas estavam dentro do intervalo de valores dados como argumento no comando. Se sim, são devolvidas ao utilizador.

4 Fugas de memória

A nível de perdas de memória, como é possível observar na figura 1, no resultado da utilização da ferramenta Valgrind durante a execução do programa, não ocorrem *leaks*, isto é, não ocorrem fugas de memória, com exceção de uma perda recuperável resultante da inclusão da biblioteca glib. Tal resultado foi obtido através de uma gestão cuidadosa da memória alocada. De destacar o uso de duas funções para libertar a memória alocada para duas estruturas indispensáveis à execução do projeto, as hash_tables da biblioteca glib, que servem de base a grande parte do armazenamento, cujo espaço utilizado é libertado pela função g_hash_table_destroy(auxiliada por funções de destruição de *key*) e a struct data, cuja memória é libertada pela função free_mem, ambas criadas pelo grupo.

```
==7369== Memcheck, a memory error detector
==7369== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7369== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==7369== Command: ./programa-principal /home/rod/Documents/Testes1/Dataset/ /home/rod/Documents/Testes1/Dataset/input.txt
==7369== Parent PID: 2403
==7369==
==7369==
==7369== HEAP SUMMARY:
==7369==     in use at exit: 19,276 bytes in 10 blocks
==7369==   total heap usage: 135,213,702 allocs, 135,213,692 frees, 1,860,988,331 bytes allocated
==7369==
==7369== LEAK SUMMARY:
==7369==    definitely lost: 0 bytes in 0 blocks
==7369==    indirectly lost: 0 bytes in 0 blocks
==7369==      possibly lost: 0 bytes in 0 blocks
==7369==    still reachable: 19,276 bytes in 10 blocks
==7369==         suppressed: 0 bytes in 0 blocks
==7369== Reachable blocks (those to which a pointer was found) are not shown.
==7369== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==7369==
==7369== For lists of detected and suppressed errors, rerun with: -s
==7369== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 1: Fugas de memória do programa, analisadas via Valgrind

5 Testes de desempenho

Relativamente à execução de testes foram registados os seguintes resultados:

- Para os testes funcionais

  - Dataset regular e respetivos inputs e outputs, o programa executou com sucesso todas as querys;

  - Dataset large, e respetivos inputs e outputs, o programa executou com total sucesso todas as querys. É de mencionar que algumas estão referenciadas como incorretas nos testes realizados online, por falha na apresentação da casa das milésimas e que deriva apenas de características da linguagem C, por isso, devem ser consideradas como bem sucedidas.

- Para os testes de performace
  - Computador 1, com processador Intel Core i3 de 11ª geração, com 2 cores a 3.0GHz e 8GB de RAM, foi registado uma melhoria média de 11,5% no tempo médio de execução, e 9,4% a nível do uso de memória;
  - Computador 2, com processador M1 Max, com 10 cores a 3.0GHz e 32GB de RAM, foi registado uma melhoria em relação à média de 3,8% no tempo médio de execução, e 5,3% a nível do uso de memória;
  - Computador 3, com processador M1, com 8 cores a 3.0GHz e 16GB de RAM, foi registado um decréscimo médio de 13,6% no tempo médio de execução, e 12,1% a nível do uso de memória.

  A diferença entre os computadores 2 e 3 deve-se a um hardware superior, já a performance superior do 1 para o 2, apesar do hardware notoriamente inferior, poderá ter sido do SO, sistema operativo, Linux, que é mais eficiente do que o SO, macOS. Por fim, as diferenças entre o 1 e o 3 podem ser justificadas de forma análoga, dado que se tratam do mesmo modelo que o computador 2. O computador 3, devido às condições em que iniciou o teste, nomeadamente a bateria disponível, e outras tarefas a executar em segundo plano, os seus resultados podem ter sido afetados.

  A nível de diferença entre Datasets na execução de querys, o large é em média 8,6% mais lento que o regular, consequência da maior distância entre posições de memória das instruções que iniciam o programa e dos dados a que é preciso aceder.

Através da tabela, 1 poderá visualizar os dados médios a nível dos tempos por nós obtidos, para a realização de cada querie.

Queries	**Invocações**	**Média**	**Desvio**	**Min**	**Max**
Load (Regular)	2	6.8327735	0.1565023	6.6321478	7.1145840
Load (Large)	2	105.8147095	1.7106212	103.3020706	108.4890747
1	352	0.716	0.162	0.0000510	0.0905610
2	35	0.0000896	0.0000117	0.0000730	0.0001070
3	35	0.0000855	0.0000092	0.0000680	0.0000980
4	31	0.0000619	0.0000147	0.0000490	0.0001060
5	75	0.1325789	0.0043879	0.1276760	0.1461550
6	105	0.1344201	0.0037874	0.1274070	0.1468860
7	89	0.0080816	0.0001849	0.0078700	0.0084490
8	37	0.1170396	0.0027803	0.1135930	0.1215700
9	55	0.0887494	0.0012317	0.0871260	0.0905610

Tabela 1: Dados médios de load e de cada query

A nível de utilização da memória, foram obtidos os seguintes valores (em Megabytes) para a execução completa do programa:

Média	Desvio	Min	Max
278,259353	2,959760658	273,840815	282,042241

Tabela 2: Dados do Dataset Regular

Média	Desvio	Min	Max
2777,1595	6,707834919	2763,35351	2787,94229

Tabela 3: Dados do Dataset Large

Analisando as tabelas 2 e 3,a execução do programa com o Dataset large, que é 100 vezes maior que o Regular, necessita de apenas 10 vezes mais de memória do que este último.

Calculando o tempo de execução das queries, cujo range de valores a analisar é variável, obitvemos os seguines resultados:

Range (elementos impressos)	Média	Desvio	Min	Max
10	0.0000589	0.0000061	0.0000520	0.0000700
50	0.0000761	0.0000037	0.0000720	0.0000850
500	0.0002799	0.0000094	0.0002660	0.0002930
1000	0.0004454	0.0000143	0.0004280	0.0004650

Tabela 4: Tempos da Query 2 para 2 invocações, 10 repetições nos 3 computadores

Range (elementos impressos)	Média	Desvio	Min	Max
10	0.0000596	0.0000060	0.0000520	0.0000690
50	0.0000745	0.0000054	0.0000670	0.0000840
500	0.0002576	0.0000165	0.0002370	0.0002900
1000	0.0004090	0.0000130	0.0003900	0.0004240

Tabela 5: Tempos da Query 3 para 2 invocações, 10 repetições nos 3 computadores

Range (dias de intervalo)	Média	Desvio	Min	Max
1	0.1416902	0.0068574	0.1349670	0.1532950
90	0.1428839	0.0057238	0.1378860	0.1544040
365 (1 ano)	0.1452072	0.0087756	0.1372250	0.1569100
1096 (3anos)	0.1442687	0.0070761	0.1369500	0.1553500

Tabela 6: Tempos da Query 5 para 2 invocações, 10 repetições nos 3 computadores

Range (dias de intervalo)	Média	Desvio	Min	Max
1	0.1344963	0.0064005	0.1276110	0.1456080
90	0.1361547	0.0093229	0.1284540	0.1538120
365 (1 ano)	0.1359145	0.0078711	0.1283370	0.1527110
1096 (3anos)	0.1339474	0.0095430	0.1282610	0.1564330

Tabela 7: Tempos da Query 6 para 4 invocações, 10 repetições nos 3 computadores

Range (elementos impressos)	Média	Desvio	Min	Max
10	0.0000633	0.0000064	0.0000580	0.0000800
50	0.0001013	0.0000068	0.0000960	0.0001180
500	0.0003521	0.0000093	0.0003390	0.0003690
1000	0.0076517	0.0001653	0.0074200	0.0080320

Tabela 8: Tempos da Query 7 para 4 invocações, 10 repetições nos 3 computadores

Range (anos da conta)	Média	Desvio	Min	Max
5	0.1576354	0.0068229	0.1474940	0.1674300
10	0.1253210	0.0068419	0.1169980	0.1339090
11	0.1212817	0.0072041	0.1137360	0.1316690
12	0.1175278	0.0057733	0.1109790	0.1254740

Tabela 9: Tempos da Query 8 para 4 invocações, 10 repetições nos 3 computadores

Range (dias de intervalo)	Média	Desvio	Min	Max
1	0.0996422	0.0009760	0.0985360	0.1019990
90	0.1209697	0.0008835	0.1200630	0.1227090
365 (1 ano)	0.1461668	0.0020901	0.1440660	0.1510320
1096 (3anos)	0.1981735	0.0014046	0.1964140	0.2013020

Tabela 10: Tempos da Query 9 para 2 invocações, 10 repetições nos 3 computadores

Como é possível observar nas tabelas 4 e 5, as queries 2 e 3 têm um comportamento temporal semelhante, derivado da sua estrutura aproximada a nível de código. O tempo de execução aumenta com o número de elementos a apresentar, comportando-se como uma função linear em ambas.

As queries 5 e 6, supramencionadas nas tabelas 6 e 7, também exibem um elevado grau de similaridade entre si e é possível observar que o tempo de execução de ambas se mantém praticamente constante, independentemente da variação do range.

A query 7, apresentada na tabela 8, assemelha-se à query 2. No entanto, como necessita de executar mais operações para apresentar apenas os da cidade requerida, o tempo de execução é maior.

A query 8, exposta na tabela 9, apresenta tempos de execução bastante próximos uns dos outros, quando o range de idade da conta é alto, mas um pouco piores quando este é curto. Um dos fatores que pode ter levado a essa diferença de tempo poderá ser o facto dos dados avaliados terem mais contas recentes, provocando um incremento temporal de execução para um range menor.

A query 9, mencionada na tabela 10, exibe um comportamento linear quanto ao número de dias do intervalo de datas usadas. o que pode ser facilmente justificado pelo facto de haver mais viagens a analisar, elevando o número de comparações, de memoria usada, entre outros, do programa.

De uma forma geral e de acordo com os parâmetros indicados pelo corpo docente, concluímos que a nível de performance e desempenho, tanto no uso de memória como no tempo de execução, o nosso programa cumpre com um elevado nível de rigor.

6 Conclusões

Com a finalização deste projeto, podemos com certeza afirmar que a nossa capacidade de utilização da linguagem C, se encontra agora num nível que nos permite o desenvolvimento de programas muito mais complexos, que envolvam técnicas cuja evolução na utilização das mesmas já havia sido mencionada no final da 1ªa fase, como a criação de estruturas de dados e a utilização de funções de bibliotecas já existentes. Para além disso, estamos agora habilitados a criar mecanismos de testes aos nossos programas que, em adição à capacidade de testar os tempos de execução e a memória utilizada, são preparados para executar diversos e complexos cálculos estatísticos sobre esses dados, o que permite aumentar a escala dos testes e, por isso, o grau de fiabilidade dos mesmos.

Tendo em conta aquilo que foi proposto para esta fase, consideramos mais uma vez não só ter cumprido com os objetivos, mas também indo além dos mesmos, incluindo várias características extra, algumas das quais sugeridas pelos docentes, como a escolha de linhas na paginação e outras que achámos da nossa relevância para a melhor utilização do programa nas suas várias vertentes, modo batch, interativo e de testes. Tal como era esperado, o método de trabalho que foi empregue na primeira fase facilitou imenso a conclusão do modo batch e o desenvolvimento dos restantes modos. De mencionar que, como resultado da nossa interação com o corpo docente durante a apresentação da primeira fase, reconhecemos necessário a alteração de algumas estruturas e a criação de novas, levando a melhor modularização, encapsulamento e reaproveitamento do código, permitindo simplificá-lo e tornar o programa mais eficiente. Após estas alterações, o modo batch foi rapidamente finalizado, com o desenvolvimento das restantes queries nas estruturas já existentes e utilizando mecanismos similares aos que já havíamos utilizado. Seguidamente, o modo interativo foi efetuado com grande aproveitamento das funções criadas para o modo batch. Por fim, o modo de testes foi mais trabalhoso, pois apesar de ser inteiramente baseado no modo batch, envolveu desafios por nós criados, que permitissem não só a medição do tempo de load e execução das querys para uma execução do programa, mas para várias execuções, armazenando todos esses dados e executando cálculos estatísticos relevantes sobre os mesmos.

A nível da escrita do código, não só mantivemos o padrão estabelecido durante a primeira fase, como também acreditamos que demonstramos grande evolução, resultado das críticas tecidas durante a apresentação da primeira fase, como já foi referido. Assim, adicionalmente ao programa ser de fácil utilização para qualquer pessoa, também o código pode ser facilmente alterado conforme a necessidade de inserção de novas características que se deseje que o programa possua.

Com um tempo máximo para a execução da maioria das querys em qualquer range a rondar a décima do segundo, e nunca superior a duas décimas, após o load, que para o dataset normal é efetuado com um tempo médio bastante inferior a 10 segundos(6,8 segundos), tempo que era considerado razoável para a execução de uma query, e para o dataset large é de menos de 2 minutos (105,8 segundos), orgulhamo-nos de ter um programa que, de acordo com os padrões estabelecidos pelo corpo docente, pode ser considerado bastante eficiente em termos temporais, sem descuido no uso da memória, com valores máximos inferiores aos valores de referência fornecidos e sem a existência de *leaks* .

Tendo tudo em conta, o balanço final que fazemos do projeto mantém-se bastante positivo e reafirmamos que foi um prazer para os elementos do grupo o seu desenvolvimento em conjunto.

7 Grafo de dependências

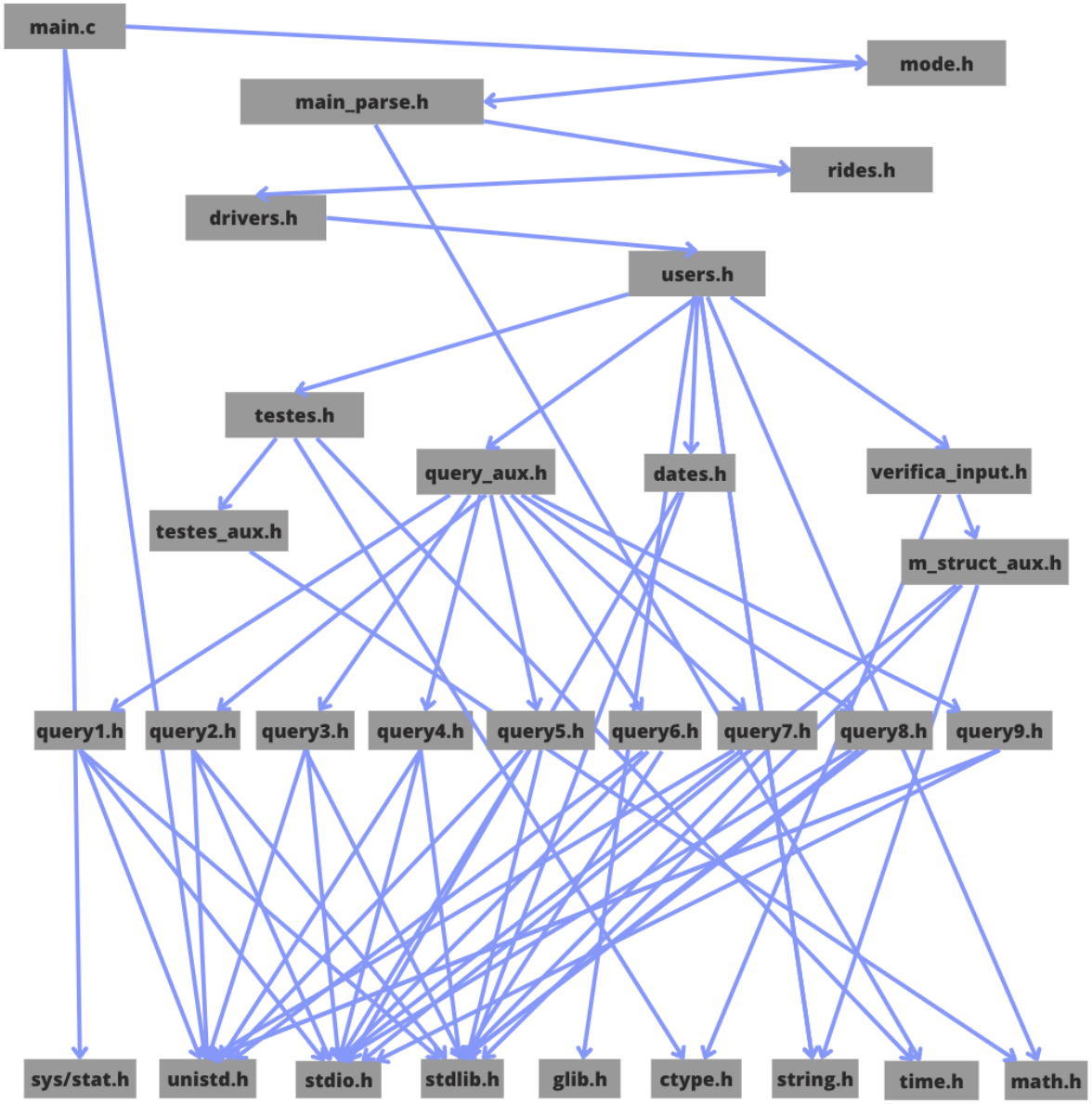

Figura 2: Grafo de dependências do projeto