



UNIVERSIDADE DO MINHO

LICENCIATURA EM ENGENHARIA INFORMÁTICA

Grupo 14 - Relatório Final

Filipe Simões Pereira, A100552

João Pedro Silva Lopes, A100829

Duarte Machado Leitão, A100550

Laboratórios de Informática III

2º Ano, 1º Semestre

Departamento de Informática

5 de fevereiro de 2023

Índice

1	Introdução	1
2	Modelação	1
2.1	Arquitetura da aplicação	1
2.2	Estruturas de dados	2
2.2.1	Estruturas fundamentais	2
2.2.2	Catálogos	3
2.2.3	Relações entre estruturas de dados	5
3	Implementação	5
3.1	Queries	5
3.2	Modo Interativo	7
3.2.1	Paginação	7
3.3	Unidade de Testes	9
3.4	Makefile	9
4	Desempenho	11
5	Conclusão	12

1 Introdução

O presente relatório procura expor o trabalho realizado ao longo da segunda fase do projeto da unidade curricular de Laboratórios de Informática III. Este projeto tem como objetivo criar uma solução eficiente para processar e armazenar grandes quantidades de dados, possibilitando o acesso aos mesmos de forma eficiente. Além disso, o projeto procura desenvolver e consolidar os conceitos de modularidade, encapsulamento e abstração de código para torná-lo mais claro e fácil de manter.

Neste relatório serão descritos os métodos escolhidos para a resolução das queries, as estruturas de dados adotadas, a implementação de um modo interativo e os testes efetuados para avaliação da correção e do desempenho da aplicação.

2 Modelação

2.1 Arquitetura da aplicação

A aplicação que tem quatro componentes principais:

- **Parser:** faz a leitura e o tratamento de todos os ficheiros de dados, de modo a que estes possam ser validados e inseridos no respetivo catálogo; faz também a leitura e interpretação de todos os comandos de input.
- **Catálogos:** reúnem todas as principais funções e estruturas de dados necessárias para a resolução das queries.
- **Módulo das queries:** recebe a interpretação de input feita pelo parser, identifica a query que deve ser executada e solicita aos catálogos o resultado dessa query.
- **Modo interativo:** interface gráfica que permite a utilização da aplicação através do terminal.

A figura 1 ilustra a interação entre estes componentes e fornece uma visão geral da arquitetura da aplicação.

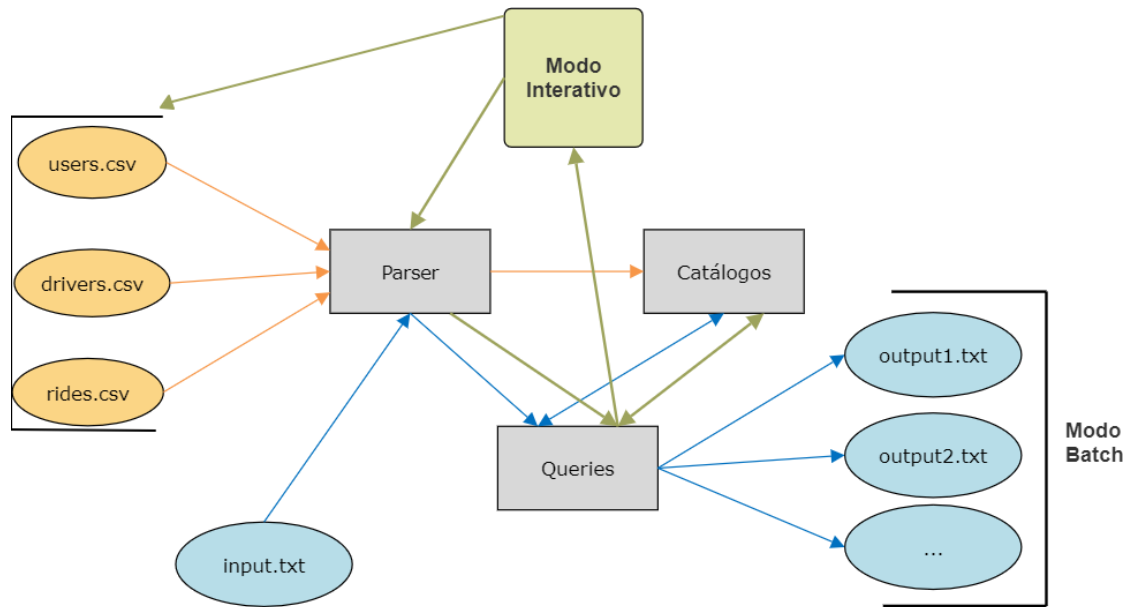


Figura 1: Arquitetura da aplicação

2.2 Estruturas de dados

2.2.1 Estruturas fundamentais

Para facilitar a representação dos dados provenientes de um *dataset*, foram criadas três entidades principais: *User*, *Driver* e *Ride*.

- **User:** contém a informação necessária de um utilizador, resultante do tratamento do dataset efetuado no parser, e tem uma subestrutura *user_stats* que inclui as estatísticas do mesmo, de modo a facilitar e a melhorar o desempenho das *queries*.
- **Driver:** contém a informação necessária de um condutor, resultante do tratamento do dataset efetuado no parser, e tem uma subestrutura *driver_stats* que inclui as estatísticas do mesmo, de modo a facilitar e a melhorar o desempenho das *queries*.
- **Ride:** contém a informação necessária de uma viagem, resultante do tratamento do dataset efetuado no parser.

User	Driver	Ride
char *username char *name unsigned short birth_date unsigned short account_creation char gender bool account_status struct user_stats { unsigned int total_rating double total_spent_money unsigned short total_rides unsigned short total_distance unsigned short latest_ride unsigned short account_age } stats	unsigned int id char *name unsigned short birth_date unsigned short account_creation char *car_class char gender bool account_status struct driver_stats { double total_earned_money unsigned short latest_ride unsigned short account_age GPtArray *ratings } stats	unsigned int id unsigned short date unsigned int driver_id char *user unsigned short city unsigned short distance unsigned short score_user unsigned short score_driver double tip double cost

Figura 2: Implementação das estruturas fundamentais

Na figura 2 é possível ver o tipo **GPtArray** presente na subestrutura do *Driver*. Este tipo de dados está definido e documentado na biblioteca [GLib-2.0](#). O tipo foi utilizado para armazenar a avaliação total de um condutor e o seu número de viagens numa dada cidade. Cada elemento do array está associado a uma cidade diferente.

2.2.2 Catálogos

Os catálogos são as estruturas de dados que armazenam e organizam todas as estruturas fundamentais de modo a providenciar fácil e rápido acesso às mesmas.

- ***Users Catalog***: armazena todos os *Users* criados pelo programa; tem também todas as funções necessárias para resolver as queries referentes a *Users* e aceder a estes de forma eficiente.
- ***Drivers Catalog***: armazena todos os *Drivers* criados pelo programa; tem também todas as funções necessárias para resolver as queries referentes a *Drivers* e aceder a estes de forma eficiente.
- ***Rides Catalog***: armazena todas as *Rides* criadas pelo programa e prepara todas as estatísticas referentes ao *User* e ao *Driver* presentes numa dada *Ride*

de modo a serem passadas ao respetivo catálogo e atualizadas na estrutura principal correspondente; tem também todas as funções necessárias para resolver as queries referentes às *Rides* e aceder a estas de forma eficiente.

Users Catalog	Rides Catalog
GPtArray *users_array GHashTable *users_ht bool is_sorted	GPtArray *rides_array GPtArray *rides_male_gender_array GPtArray *rides_female_gender_array GPtArray *rides_city_arrays; enum sort_status { UNSORTED, DATE, MALE, FEMALE, DATE_AND_MALE, DATE_AND_FEMALE, GENDER, DATE_AND_GENDER } sort_status
Drivers Catalog	Rides_in_City
GPtArray *drivers_array GPtArray *drivers_average_rating_arrays unsigned int last_inserted_id	GPtArray *rides_by_date double average_price bool is_sorted
Drivers_in_City	
GPtArray *drivers_by_city bool is_sorted	

Figura 3: Implementação dos catálogos

Na figura 3 é possível ver tipos como **GPtArray** e **GHashTable** que estão definidos e documentados na biblioteca GLib-2.0 (GPtArray: <https://docs.gtk.org/glib/struct.PtrArray.html>, GHashtable: <https://docs.gtk.org/glib/struct.HashTable.html>).

Na figura 3 estão presentes também duas outras estruturas **Drivers_in_City** e **Rides_in_City** que são usadas como elementos do *Drivers Catalog* (GPtArray *drivers_average_rating_arrays) e do *Rides Catalog* (GPtArray *rides_city_arrays) respetivamente. De maneira semelhante à referida na estrutura principal *Driver* estas estruturas são utilizadas para aceder facilmente aos *Drivers* ou *Rides* numa certa cidade.

2.2.3 Relações entre estruturas de dados

Na figura 4 é exemplificada a interação entre os catálogos e as estruturas principais. Os catálogos conseguem interagir uns com os outros mas o acesso às estruturas principais só pode ser feito pelo catálogo que as armazena com a utilização exclusiva de *getters* e *setters*, encapsulando e tornando opacas ao programa as estruturas principais. As estruturas principais não têm nem necessitam de qualquer acesso aos catálogos.

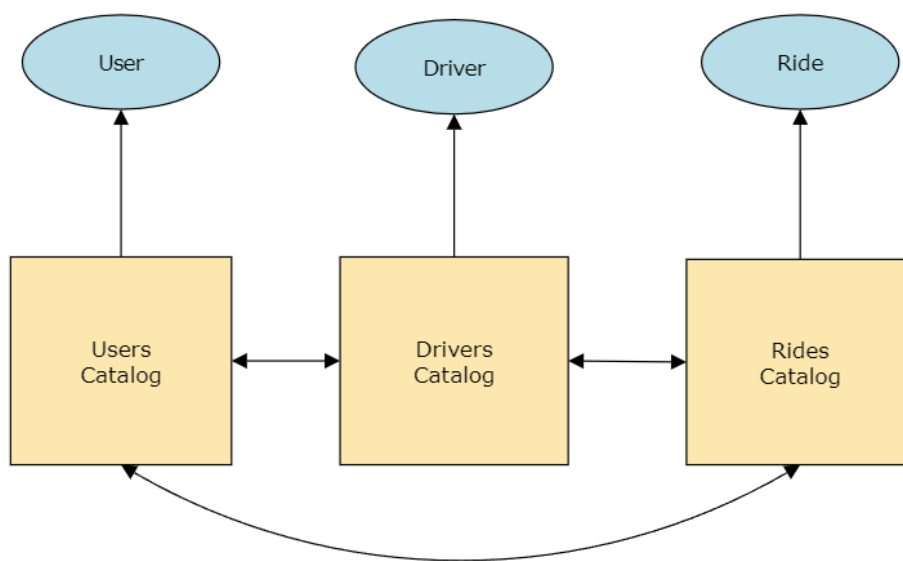


Figura 4: Relação entre estruturas de dados

3 Implementação

3.1 Queries

Para resolver a **query 1**, é necessário determinar se o perfil desejado corresponde a um driver ou a um user. De seguida acede-se ao respetivo catálogo e procura-se o perfil através do *input*, onde tem todos os dados para o *output*.

A **query 2** recorre à struct `drivers_in_city` (índice 7), onde se encontram

todos os drivers com a sua avaliação média global. Estes drivers são ordenados pelos parâmetros do enunciado e em seguida é construído o output com os primeiros N (argumento) drivers.

A resolução da **query 3** é bastante semelhante à da query 2. Numa fase inicial, o array de users é ordenado de acordo com os parâmetros do enunciado. Depois é apenas necessário construir o output com a informação dos primeiros N users.

A **query 4** recorre ao `rides_city_arrays` do catálogo das rides. É necessário encontrar o índice onde se encontra a struct `Rides_in_City` relativa à cidade recebida como argumento. O preço médios das viagens (sem grojeta) já se encontra calculado na struct. É apenas necessário incluí-lo no output.

A resolução da **query 5** começa com a ordenação do array de rides por data. Com o array ordenado, procura-se a primeira ocorrência da data de início e a última ocorrência da data final. A partir daí são lidos os custos das rides e é calculada a média.

A **query 6** também recorre ao `rides_city_arrays` para aceder apenas às rides na cidade recebida como argumento. Estas rides são depois ordenadas por data e procura-se a primeira ocorrência da data de início e a última ocorrência da data final. Finalmente são lidos os índices relevantes do array e é calculada a distância média percorrida.

Na **query 7** é usada a struct `drivers_in_city` correspondente à cidade pedida. É necessário, no início da resolução, encontrar o índice do array correspondente à cidade recebida como argumento. De seguida ordena-se o array obtido de acordo com os critérios dados. Basta agora construir o output com os dados pedidos dos N primeiros drivers do array ordenado anteriormente.

Para resolver a **query 8**, existem no catálogo das rides dois arrays compostos por rides com drivers e users do mesmo género. O array escolhido será ordenado de acordo com os parâmetros do enunciado. Em seguida, procura-se a última ocorrência do array em que o driver tem idade maior ou igual à pedida no input. Por fim, o array é percorrido e o output preenchido até atingir o resultado da

procura.

Na **query 9** começa-se por ordenar o array de rides por data e encontrar a primeira ocorrência da data de início e a última ocorrência da data final. Todas as rides no intervalo de datas são transferidas para um novo array para que estas sejam reordenadas de acordo com os parâmetros do enunciado. O array criado tem agora todas as rides relevantes na ordem correta. Apenas resta mostrar o output.

Nota: a ordenação dos arrays é feita recorrendo sempre ao mesmo método. Existem várias funções para comparar os dados de acordo com os parâmetros do enunciado. Estas funções serão depois usadas pela função *g_ptr_array_sort* da *GLib*, que irá realizar a ordenação. Vale a pena mencionar que os arrays serão ordenados apenas se for necessário. Existem flags que indicam se os arrays foram ou não ordenados previamente.

3.2 Modo Interativo

O modo interativo foi implementado com a biblioteca *ncurses*, consiste num programa que permite a introdução do *path* para o dataset que pretende-se utilizar para a execução de *queries* ou observação dos *datasets*, tudo devidamente paginado se tal for necessário.

O programa funciona melhor com resoluções de proporção *16:9* mas devido ao uso das macros da *ncurses* permite um redimensionamento limitado, sendo que resoluções muito baixas ou com proporções muito diferentes poderão ter problemas a apresentar o programa na sua totalidade.

3.2.1 Paginação

A paginação de resultados longos faz-se ao alterar a ordenada onde o *output* é impresso. Mais precisamente, obtém-se o número de linhas da janela onde as strings são impressas e multiplica-se esse número (em negativo) pelo índice da página atual menos um, ou seja:

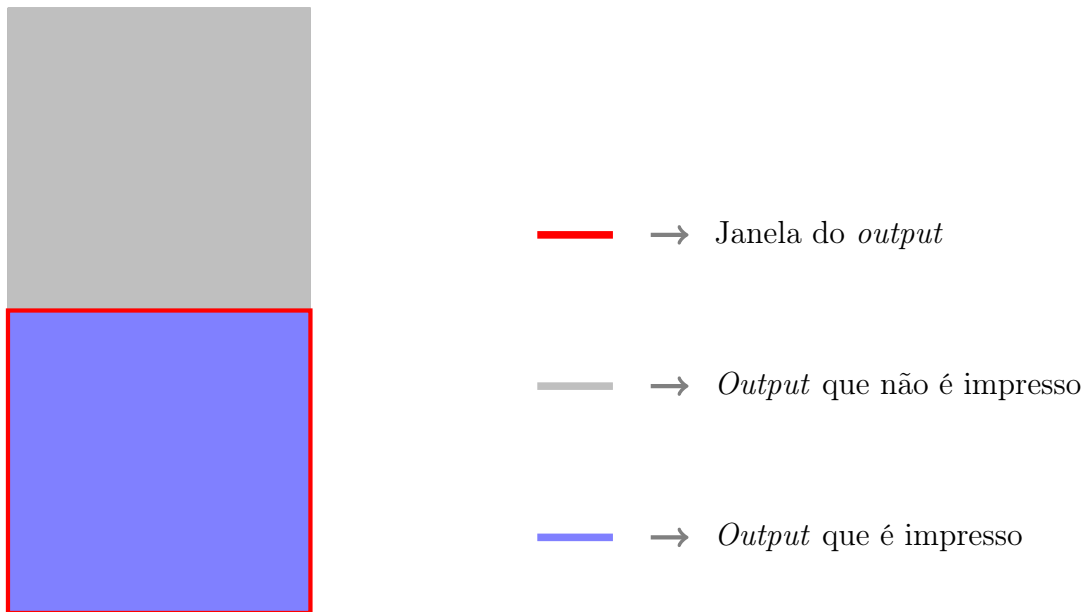
$$y = (-linhas) * (pagina - 1).$$

Este método provou-se mais eficiente que dividir uma string num array de

strings, especialmente para resultados maiores como os da *query 9* para grandes intervalos de tempo, no entanto ainda existiam preocupações quanto ao desempenho devido à string (ou ficheiro) estar a ser impressa na sua totalidade independentemente de qual pedaço é que realmente deve ser apresentado ao utilizador, isto resultava em grande parte da string/ficheiro estar invisível, o que é claramente um desperdício visto que apenas interessa a parte da string/ficheiro que vai ser impressa com ordenada maior ou igual a 0 e menor ou igual ao número de linhas da janela, pois apenas essa parte é visível e apenas essa parte representa a página atual:

$$0 \leq y \leq \textit{linhas}.$$

Este problema resolveu-se ao alterar as funções de paginação, simplesmente percorre-se a string (ou ficheiro) até a sua ordenada ser 0 e a partir daí imprimem-se linhas da string/ficheiro até a sua ordenada ser igual ao número de linhas (tal como na relação acima). A função termina depois da página estar impressa, evitando a necessidade de percorrer a string/ficheiro na sua totalidade em todos menos o pior caso.

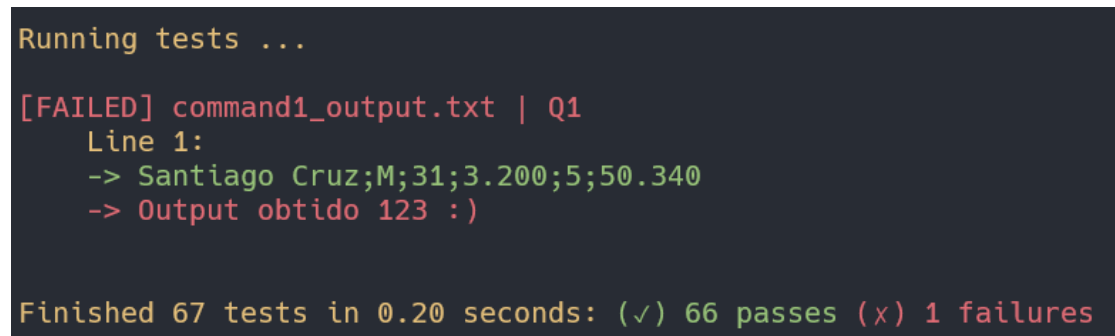


3.3 Unidade de Testes

A unidade de testes compara os resultados obtidos na execução do programa em modo batch com os resultados esperados para o input e dataset usados.

A primeira comparação a fazer é a do número de ficheiros em ambas as pastas de resultados, para serem iguais é necessário que tenham o mesmo número de ficheiros, de seguida compara-se o tamanho de todos os ficheiros e se os ficheiros comparados nunca tiverem tamanhos diferentes passa-se à ultima comparação, sendo essa comparar os ficheiros por *chunks*, cada chunk com no máximo *512KB*, se esta comparação nunca falhar significa que os outputs são iguais ao esperado.

Esta unidade também tem a capacidade de encontrar as linhas onde ocorrem erros caso tais aconteçam, para isso sempre que alguma comparação a partir da comparação de tamanhos falha é executada uma função que procura linhas diferentes nos dois ficheiros (resultado obtido e resultado esperado) e imprime essas linhas como se pode ver na seguinte imagem:



```
Running tests ...  
[FAILED] command1_output.txt | Q1  
  Line 1:  
    -> Santiago Cruz;M;31;3.200;5;50.340  
    -> Output obtido 123 :)  
  
Finished 67 tests in 0.20 seconds: (✓) 66 passes (✗) 1 failures
```

Figura 5: Unidade de testes com um teste falhado

Foi usado este método para a unidade de testes devido ao seu desempenho, tanto comparar tamanhos de ficheiros como comparar chunks são processos bastante rápidos.

3.4 Makefile

Para a criação de um ambiente de desenvolvimento mais produtivo foi utilizada uma Makefile com três modos de *BUILD* do executável principal sendo estes

release, debug e sanitizer, cada um com as flags mais adequadas. A seguir estão enumeradas todas as regras que podem ser utilizadas com esta Makefile:

- **make**: linka todos os ficheiros *.c* e *.h* presentes no projeto.
- **make run**: depende da criação do executável e corre o programa no modo batch com um dataset e o respetivo input; utiliza o comando `time` com o objetivo de mostrar o total de memória utilizado e o tempo de execução do programa.
- **make run i**: depende da criação do executável e corre o programa em modo interativo.
- **make set**: seleciona o dataset, input e resultados de output que deverão ser usados em conjunto com os restantes comandos; tem 4 opções de seleção (`sv`, `si`, `lv`, `li`), qualquer outro modo será rejeitado por esta regra; `sv` -> seleciona dataset regular e sem entradas inválidas, bem como respetivo input e resultados de output; `si` -> seleciona dataset regular e com entradas inválidas, bem como respetivo input e resultados de output; `lv` -> seleciona dataset large e sem entradas inválidas, bem como respetivo input e resultados de output; `li` -> seleciona dataset large e com entradas inválidas, bem como respetivo input e resultados de output; o único problema deste comando é o facto de todos os datasets, inputs e resultados necessitarem de um nome específico (datasets: `sv` -> `small-valid-dataset`, `si` -> `small-invalid-dataset`, `lv` -> `large-valid-dataset`, `li` -> `large-invalid-dataset`; inputs: `sv` -> `sv-input.txt`, `si` -> `si-input.txt`, `lv` e `li` -> `lv-li-input.txt`, resultados de output: `sv` -> `sv-outputs`, `si` -> `si-outputs`, `lv` -> `lv-outputs`, `li` -> `li-outputs`)
- **make test**: cria e corre o executável que executa os testes de correção referidos na secção anterior.
- **make gdb**: compila o projeto em modo *BUILD* e abre o `gdb` com o executável já com o dataset e input inseridos.
- **make valgrind**: compila o projeto em modo *BUILD* e corre o `valgrind` com um dataset e input próprios que rapidamente gera os resultados com todos os possíveis memory leaks que possam ter ocorrido durante a execução do

programa.

- ***make clean***: elimina todos os executáveis que possam ter sido gerados e as respectivas dependências; limpa também todos os paths criados pelo *make set*.

4 Desempenho

Foram testados o tempo de execução de cada um dos datasets e o pico do uso de memória de cada um deles em dois computadores com as seguintes especificações:

	PC 1	PC 2
CPU	Intel i5-10300H	AMD Ryzen 5 5600H
RAM	8GB DDR4	16 GB DDR4
Disco	512GB SSD	512 GB SSD
Sistema Operativo	EndeavourOS	WSL 2
Compilador	GCC 12.2.1	GCC 9.4.0

Tabela 1: Especificações dos computadores usados para testes

Cada computador executou cada dataset dez vezes, os valores apresentados são as médias dessas execuções.

Obtiveram-se os seguintes resultados quanto aos picos de memória:

Datasets	PC 1	PC 2
Regular Dataset (without invalid entries)	149	149
Regular Dataset (with invalid entries)	144	145
Large Dataset (without invalid entries)	1488	1493
Large Dataset (with invalid entries)	1444	1447

Tabela 2: Pico de uso de memória nos computadores (em MB)

Obtiveram-se os seguintes resultados quanto ao tempo de execução:

Datasets	PC 1	PC 2
Regular Dataset (without invalid entries)	5,09	4,88
Regular Dataset (with invalid entries)	4,54	4,49
Large Dataset (without invalid entries)	67,79	68,01
Large Dataset (with invalid entries)	63,29	63,36

Tabela 3: Tempo de execução nos computadores (em segundos)

Não existem diferenças significativas entre computadores em nenhum dos valores.

5 Conclusão

A realização deste trabalho prático permitiu consolidar vários conceitos sobre o desenvolvimento de programas. Desde a primeira fase do projeto foram realizadas melhorias que se refletiram na qualidade e funcionamento geral da aplicação. A separação de um único catálogo em três permitiu ter um código mais modular e fácil de manter. A reestruturação das estruturas de dados presentes nos catálogos revelou o efeito que a organização dos dados pode ter no desempenho do programa.

Fomos ainda capazes de tirar mais proveito de ferramentas de desenvolvimento. O uso do debugger *GDB* facilitou a detecção e correção de erros. As bibliotecas externas *GLib* e *ncurses* revelaram-se úteis na criação de estruturas de dados eficientes e na implementação do modo interativo. A adoção de testes unitários permitiu garantir que as nossas soluções produziam sempre o resultado esperado. A criação de uma *makefile* permitiu a incorporação destas ferramentas no nosso projeto, tornando assim o nosso ambiente de desenvolvimento mais produtivo.