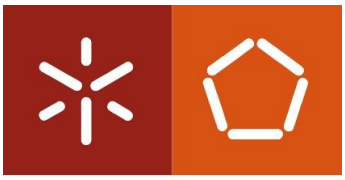


Universidade do Minho

Departamento de Informática

Projeto de Laboratórios de Informática III
Grupo nº 31

Ricardo Gomes (a93785) Rui Fernandes (a89138)

Abril 2020

Resumo

O presente relatório descreve o projeto realizado no âmbito da disciplina de *Laboratórios de Informática III* (LI3), ao longo do segundo semestre do segundo ano do Mestrado Integrado em Engenharia Informática da Universidade do Minho.

Este projeto teve como principal objetivo o processamento e tratramento de grandes volumes de dados provenientes de três ficheiros de texto, e posterior desenvolvimento de um sistema de gestão de vendas modular capaz de armazenar informação de vendas e relacionar, de forma eficiente, produtos, clientes e vendas.

Este projeto considera-se, de certa forma, um desafio pelo facto de estarmos a falar de programação em larga escala, uma vez que se trata de uma aplicação com grandes quantidades de dados e com uma elevada complexidade algorítmica e estrutural. Nesse sentido, o desenvolvimento deste programa foi realizado seguindo príncipios como modularidade, encapsulamento de dados, conceção de código reutilizável e gestão eficiente de memória.

Conteúdo

1 Problema **3**
1.1 Descrição do problema 3

2 Módulos de dados e API **4**
2.1 Catálogo de Clientes 4
2.2 Catálogo de Produtos 4
2.3 Venda 5
2.4 Faturação Global 5
2.5 Gestão de Filial 6
2.6 SGV 7

3 Interpretação e reflexão sobre os resultados **8**

4 Conclusão **9**

5 Anexos **10**
5.1 Testes de *performance* 10
5.2 Gestão de memória 10
5.3 Grafo de dependências e Makefile 11

1 Problema

1.1 Descrição do problema

São fornecidos três ficheiros de texto que contêm informação sobre as transações de uma cadeia de distribuição:

- O ficheiro *Produtos.txt* contém cerca de 200.000 códigos de produto, sendo que cada linha deste ficheiro representa o cóodigo de um produto da cadeia de distribuição. Cada código é formado por duas letras maiúsculas seguidas de quatro digitos (ex.: *HG1389*).
- O ficheiro *Clientes.txt* contém cerca de 20.000 códigos de clientes, sendo que cada linha deste ficheiro representa o código de um cliente da distruibuidora. Cada código é formado por uma letra maiúscula seguida de quatro dígitos (ex.: *A2657*).
- O ficheiro *Vendas_1M.txt* contém 1.000.000 de registos de vendas, sendo que cada linha deste ficheiro representa uma venda efetuada numa das três filiais da cadeia de distribuição. Cada um destes registos é formado por um código de produto, preço unitário do prduto comprado, número de unidades compradas, um caractere (*'N'* ou *'P'*) conforme tenha sido uma compra normal ou uma compra em promoção, o código do cliente que efetuou a compra, assim como o mês e a filial em que esta ocorreu (ex.: *VC1429 768.94 153 N G3562 11 2*).

Com este trabalho prático prende-se extrair a informação necessária dos vários ficheiros, de forma a responder, de forma eficiente, a um conjunto de 13 *queries*, que relacionam com o conteúdo dos mesmos. tendo especial atenção aos tempos de execução, assim como ao encapsulamento dos dados e gestão eficiente de memória.

2 Módulos de dados e API

A arquitetura do programa traduz-se em quatro módulos principais: Catálogo de clientes, Catálogo de produtos, Faturação global e Gestão de filial, um módulo auxiliar que trata uma venda individual, e um módulo SGV, que relaciona todos os anteriores de modo a responder, de forma eficiente, a todas as *queries*..

Tendo cada módulo um objetivo específico, a divisão física dos ficheiros tem como finalidade distribuir as tarefas do programa de modo a garantir a abstração dos dados, bem como facilitar a reutilização do código.

Uma vez que este projeto está estruturado seguindo a arquitetura *MVC - Model. View, Controller*, além destes módulos dados, existem também outros módulos: *View*, componente visual responsável por mostrar ao utilizador as páginas, tabelas e menus, que apenas mostra e não recebe *input* do utilizador, *Controller*, componente que interage com o sistema (SGV) e com o utilizador, funcionando como intermediário entre estes, que, além de tratar e validar o input do utilizador, efetua a comunicação necessária com o sistema e passar à *View* a informação que deve ser mostrada ao utilizador, e Navegador, componente que mostra ao utilizador listas de *Strings* de grande dimensão sob a forma de páginas. Importa referir que a camada de dados não passa os próprios dados ao *Controller*, mas uma cópia destes. Desta forma, garante-se o encapsulamento dos dados, garantindo que que estes não são alterados inadvertidamente por agentes externos.

Tendo em mente o encapsulamento dos dados, estratégias como *forward declaration* de todas as estruturas definidas, implementadas com recurso à biblioteca *Glib* do projeto *Gnome*, impossibilitam que os seus atributos sejam acedidos diretamente: fora do próprio mõdulo, estes apenas são acessíveis através de *getters*.

2.1 Catálogo de Clientes

Os dados do ficheiro *Clientes.txt* são armazenados num array de AVL's com 26 posições referentes às 26 letras do alfabeto. Cada índice contém um apontador para uma AVL com os dados relativos aos clientes cujo código se inicia pela letra associada a esse índice.

A escolha da estrutura de dados recaíu sobre AVL's uma vez que, além de permitem manter os clientes organizados pelo seu código, tnato a procura como a inserção de dados nesta estrutura **são/é** relativamente eficientes (*O(logN)*). Por outro lado, a estratégia de dividir o catálogo num array potencia a eficiência destas operações, o que é especialmente importanto tendo em consideração que a procura é a operação mais efetuada na validação de vendas.

Este módulo contém também a estrutura que representa um cliente, que contém uma string que o identifica, isto é, o seu código.

O módulo que implementa o catálogo de clientes tem uma API capaz de, não só lidar com a informação de um cliente individual, mas também com o respetivo armazenamento e pesquisa.

```
struct cliente {
    char *codCliente;
};

struct catclientes {
    GTree *avl[N_LETRAS];
};
```

2.2 Catálogo de Produtos

À semelhança do Catálogo de Clientes, o Catálogo de Produtos consiste num array de AVL's com 26 posições referentes às 26 letras do alfabeto, onde serão armazenados os dados do ficheiros *Produtos.txt*. Cada índice contém um apontador para uma AVL com os dados relativos aos clientes cujo código se inicia pela letra associada a esse índice.

Note-se que, tsl como no catálogo de clientes, o uso de AVL's garante que os produtos são organizados por índice alfabético, o que permite, de forma eficaz, saber quais e quantos são os produtos, cujos códigos começam por uma determinada letra do alfabeto.

Este módulo contém também a estrutura que representa um produto, que contém uma string que o identifica, isto é, o seu código.

À semelhança do módulo Catálogo de clientes, o módulo de que implementa o catálogo de produtos tem uma API construida de maneira a que seja capaz de tratar a informação referente a um produto e lidar, de forma eficiente, com o armazenamento e pesquisa de produtos.

```
struct produto {
    char *codProd;
};

struct catprod {
    GTree *avl[N_LETRAS];
};
```

2.3 Venda

Este módulo auxiliar tem uma API capaz de fazer o *parse* de uma *String* com um formato previamente definido, **inserir NAO INSERE, CRIA** numa estrutura com os campos necessários de maneira a preservar a informação, para posteriormente ser tratada pelos módulos de Faturação e Filial.

```
struct venda {
    char *codProd;
    char *codCliente;
    double precoUnit;
    int quantidade;
    char tipo;
    int mes;
    int filial;
};
```

2.4 Faturação Global

Módulo de dados que contém as estruturas de dados responsáveis pela resposta eficiente a questões quantitativas que relacionam os produtos às suas vendas mensais, em modo Normal (N) ou em Promoção (P), para cada um dos casos guardando o número de vendas e o valor total de faturação de cada um destes tipos. Este módulo referencia todos os produtos, mesmo os que nunca foram vendidos e não contém qualquer referência a clientes, sendo, no entanto, capaz de distinguir os valores obtidos em cada filial.

De forma a facilitar a respota às *queries*, consideramos que faturação da cadeia de distribuição pode é representa por três campos: duas matrizes, uma contendo informação sobre o número de vendas, e outra sobre totais faturado, fazendo ambas distinção entre meses e tipos de venda. Assim, consultas como, por exemplo, dado o código de um produto, determinar o número de vendas em promoção num determinado mêss tornaram-se relativamente eficientes

Além destes campos, dada a inexistência de uma função de comparação especialmente útil neste contexto, e tirando partido da complexidade constante das inserções (*O(1)*) nesta estrutura, consideramos que a representação da faturação contém, também uma *Hashtable*. Nesta, a cada código de produto (*key*), está associada uma estrutura com informação sobre uma a faturação desse produto (*value*), isto é, o número de vendas, quantidade ventidade e total faturado.

```
struct infoproduto {
    char *codProd;
    int nVendas[N_MESES][N_FILIAIS][TIPOS_VENDA];
    double totalFaturado[N_MESES][N_FILIAIS][TIPOS_VENDA];
    int quantidadeVendida[N_MESES][N_FILIAIS][TIPOS_VENDA];
    int nVendasGlobal;
    double totalFaturadoGlobal;
    int quantidadeVendidaGlobal;
};

struct faturacao {
    double totalFaturado[N_MESES][TIPOS_VENDA];
    int nVendas[N_MESES][TIPOS_VENDA];
    GHashTable *produtos;
};
```

2.5 Gestão de Filial

O módulo Gestão de Filial é a *blueprint* de como uma filial é representada, daí a importância de seguirmos os princípios de modularidade, considerando que a cadeia de distribuição é composta por três filiais.

Uma vez que nenhuma ordenação se apresenta particularmente vantajosa, optamos por implementar este módulo utilizando *Hashtables*, tirando partido do facto de as operações de inserção e procura serem particularmente eficientes (*O(1)*). Isto é especialmente importante dado o elevado número de registos de venda a inserir na estrutura.

Desta forma, uma filial é representada por duas *Hashtables*. Na primeira, a cada código de cliente, está associada uma está associada uma estrutura *nodocliente*. Esta, por sua vez, possui três atributos: o código do cliente, um array de 12 posições que traduz a quantidade de produtos que este comprou em cada um dos meses, e uma *Hashtable* que relaciona o próprio cliente com os produtos que este comprou: nesta *Hashtable*, a cada código de um produto que o cliente comprou, está associada uma estrutura *produtocliente*, que relaciona um cliente e um produto em função da quantidade comprada e total faturado.

Na *Hashtable* que relaciona produtos, a cada um dos códigos de cliente está associoada uma estrutura *nodoproduto*. Esta estrutura, por sua vez, é composta por dois campos: o código do cliente em questão e uma *Hashtable* na qual, a cada código de um cliente (*key*), está associado um um apontador para o tipo *Tipo*, que permite saber se o cliente comprou o produto em causa em modo Normal, Promoção, ou Ambos.

```
struct filial {
    GHashTable *clientes;
    GHashTable *produtos;
};

struct compradores {
    GPtrArray *arr[TIPOS_VENDA];
    int quantidade[TIPOS_VENDA];
};

typedef enum {
    N, P, AMBOS
} Tipo;
```

```
typedef struct nodocliente {
    char *codCliente;
    GHashTable *produtos;
    int quantidade[N_MESES];
} *NodoCliente;

typedef struct nodoproduto {
    char *codProd;
    GHashTable *compradores;
} *NodoProduto;

typedef struct produtocliente {
    char *codProd;
    char *codCliente;
    int quantidade[N_MESES];
    double totalFaturado;
} *ProdCliente;
```

2.6 SGV

O módulo SGV é o módulo que incorpora todos os módulos anteriormente descritos, contendo este uma estrutura que contém um catálogo de clientes, um catálogo de produtos, uma estrutura Faturação, um array de três filiais e uma estrutura que contém informação relativa à leitura dos ficheiros, nomeadamentos quantos clientes, produtos e vendas foram lidos e quantos desses são, de facto, válidos, assim como o nome dos ficheiros utilizados. Este módulo faz a ponte entre todos os módulos internos e o exterior, sendo este aquele a que ao qual o *Controller* faz pedidos. Na verdade, o *Controller* é a unica entidade a interagir com o sistema.

```
struct sgv {
    CatClientes cat_c;
    CatProd cat_p;
    Faturacao fat;
    Filial filiais[N_FILIAIS];
    Info info;
};

struct info {
    int clientes_lidos, clientes_validos;
    int produtos_lidos, produtos_validos;
    int vendas_lidas, vendas_validas;
    char *clientes;
    char *produtos;
    char *vendas;
    bool err; /* flag que permite saber se os
                 ficheiros foram lidos com sucesso */
};
```

3 Interpretação e reflexão sobre os resultados

Em anexo, apresenta-se um gráficos que traduz o tempo de carregamento dos diferentes ficheiros de texto. Importa salientar que os valores apresentados não são resultado de uma só medição, mas da média de um conjunto de 15 medições.

Embora não tenhamos medido os tempos de carregamento de cada ficheiro individualmente, facilmente se conclui que a leitura mais demorada ocorre no ficheiro que contém os registos de vendas. Isto deve-se não só à validação de uma venda ser mais complexa do que a validação de um cliente ou produto, mas também ao facto da maior complexidade das estruturas de dados que armazenam informação relativa a vendas, quando comparadas com as estruturas onde são guardados os dados de clientes e produtos.

Uma vez que o número de registos de vendas vai aumentando, é aceitável que os tempos de carregamento dos ficheiros aumentem. De facto, através da análise do gráfico, podemos concluir que o tempo de carregamento dos ficheiros de dados tende a aumentar linearmente em função do número de registos de vendas (compilando o programa com diferentes *flags* de otimização, nomeadamente *-O2, -O3 e -Ofast*, notamos que estas não alteram de forma significativa o tempo de carregamento dos ficheiros) . Uma forma de atenuar este rápido crescimento seria implementar estruturas de dados especialmente eficazes no que diz respeito à inserção de dados, em particular *Hashtables* (inserção *O(1)* em contraste com *O(logN)* das AVL's). No entanto, com tal implementação, as estruturas de dados perderiam a noção de ordem. Ora, ainda que esta estratégia tenha como consequência uma redução dos tempos de carregamento dos ficheiros, também provocaria um aumento no tempo de execução de cada *query*. Dado que algumas das estruturas de dados por nós implementadas tiram partido desta noção de ordem, para responder à *query 2*, apenas são percorridos os elementos cujo código se inicia pela letra em causa, o que não aconteceria se o catálogo de produtos fosse implementado utilizando uma *Hashtable*: não havendo uma noção de ordem, seria necessário testar, de entre todos os produtos, aqueles que satisfazem a condição em causa e, posteriormente, ordenar esse conjunto pelo índice alfabético, estratégia essa que se revela particularmente problemático à medida que o volume de dados aumenta. Posto isto, consideramos que é desejável abdicar de tempos de carregamentos mais rápidos se isso significar tempos de resposta às queries particularmente otimizados.

4 Conclusão

Através do *parsing* dos ficheiros de texto, foi possível extrair toda a informação considerada relevante para a resolução das queries propostas, de forma otimizada, guardando a mesma em estruturas de dados apropriadas, motivo pelo qual consideramos que atingimos o oobjetivo proposto, ainda que haja espaço para melhorias.

No entanto, estamos conscientes de que a experiência de utilização do programa não é a mais apelativa. Apesar do esforço para tornar a sua utilização mais fácil e menos cansativa possível, características que consideramos essenciais, estes objetivos não foram atingidos na sua totalidade. Uma melhoria que poderia ser implementada no que diz respeito à utilização do programa seria a criação de um comando *help*, que elucidaria o utilizador sobre a forma como deve interagir com o programa. No entanto, de salientar que estas melhorias são de fácil implementação uma vez que o desenvolvimento do projeto seguindo princípios como modularidade funcional teve como principal intuito permitir a fácil alteração e melhoria de funcionalidades existentes, assim como a implementação de novas funcionalidades que possámos vir a considerar relevantes.

Consideramos, também que o módulo Gestão de Filial está consideravelmente mais complexo quando comparado com os restantes. Estamos convictos de que existem possíveis implementações alternativas relativamente mais simples sem que comprometam a sua eficácia e *performance*. Assim, temos noção de que as melhorias mais significativas passariam, por uma reestruturação deste módulo juntamente com o aperfeiçoamentos no que diz respeito à experiência de utilizaçao.

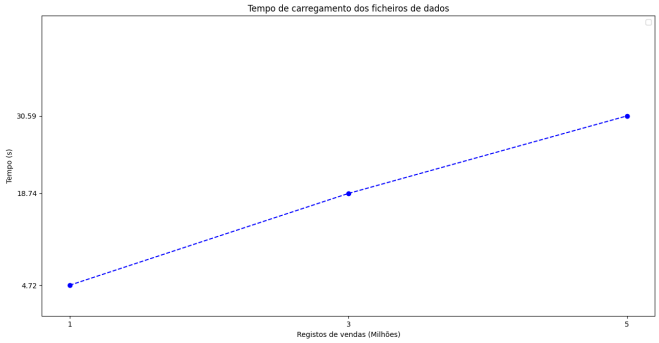
Por outro lado, analisando o programa com a ferramenta *Valgrind*, notámos que toda a memória alocada é libertada no final da execução do programa, evitando, assim, *memory leaks*.

Posto isto, ao longo do desenvolvimento deste projeto, acabamos por perceber as dificuldades associadas à gestão e tratamento de grandes quantidades de dados. Além disso, consideramos que um dos maiores desafios foi a escolha das estruturas de dados para armazenar informação.

Em suma, não obstante as potenciais melhorias que poderiam ser implementadas no programa, os testes por nós realizados revelaram tempos de execução que consideramos aceitáveis.

5 Anexos

5.1 Testes de *performance*

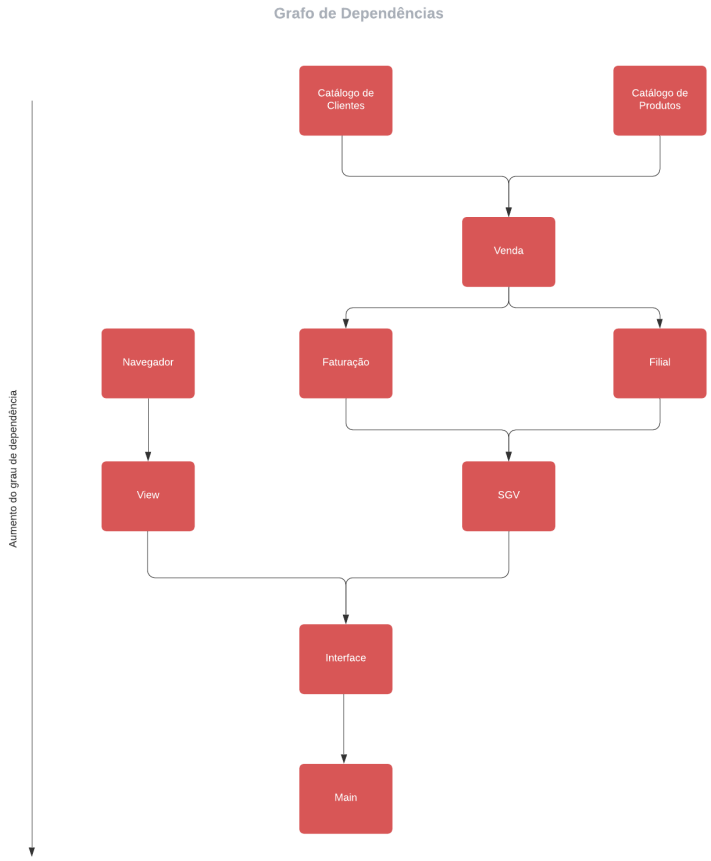


5.2 Gestão de memória

```
==3727==
==3727== HEAP SUMMARY:
==3727==     in use at exit: 18,604 bytes in 6 blocks
==3727==   total heap usage: 32,008,220 allocs, 32,008,214 frees, 836,433,010 bytes allocated
==3727==
==3727== LEAK SUMMARY:
==3727==    definitely lost: 0 bytes in 0 blocks
==3727==    indirectly lost: 0 bytes in 0 blocks
==3727==      possibly lost: 0 bytes in 0 blocks
==3727==    still reachable: 18,604 bytes in 6 blocks
==3727==         suppressed: 0 bytes in 0 blocks
==3727== Reachable blocks (those to which a pointer was found) are not shown.
==3727== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==3727==
==3727== For counts of detected and suppressed errors, rerun with: -v
==3727== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5.3 Grafo de dependências e Makefile

Apresenta-se, em seguida, o grafo de dependências.



A Makefile disponibiliza os seguintes comandos:

- **make** ou **make program**: Compila o programa.
- **make run**: Executa o programa.
- **make debug**: Permite fazer o *debug* do programa utilizando o *GDB (GNU Project Debugger)*.
- **make doc**: Gera a documentação do programa.
- **make memcheck**: Executa o programa utilizando a ferramenta *Valgrind*.
- **make clean**: Elimina o executável, a documentação e os ficheiros *.o*.