

Gestão de contatos

Conceção e Análise de Algoritmos

2 de Junho de 2014

Grupo E:

Henrique Ferrolho - 120509079 - henriqueferrolho@gmail.com

Maria Marques - 120509104 - mariajoao.rmarques@gmail.com

1. Introdução

No âmbito da unidade curricular Conceção e Análise de Algoritmos do curso Mestrado Integrado em Engenharia Informática e Computação foi-nos proposto a implementação de funcionalidades eficientes de pesquisa de contatos.

A pesquisa deve ser dinâmica e a aplicação deve mostrar contatos que interessem ao utilizador, à medida que este escreve mais caracteres. Deve assim possibilitar que o utilizador se engane na escrita e o algoritmo de pesquisa deve sugerir correções que se aproximem da lista de contatos. Para isso recorreremos à utilização do algoritmo de pesquisa aproximada de *strings EditDistance*.

2. Perguntas e respostas

Q: Que algoritmo será usado para a pesquisa de contatos?

A: O algoritmo *EditDistance*: um algoritmo de pesquisa aproximada de *strings*.

Q: Que contentor deverá ser usado para guardar os contatos durante o tempo de execução?

A: A escolha mais apropriada é a de usar o contentor *set*. Os *sets* não só armazenam elementos únicos segundo uma ordem específica, como também são implementados com uma árvore de pesquisa binária, o que será perfeito para operações de pesquisa, adição e remoção que teremos de implementar.

Q: Quanto tempo demorará uma inserção/remoção/pesquisa de contato?

A: Como o contentor dos contatos é um *set*, que novamente é implementado com uma árvore de pesquisa binária, estas operações terão uma complexidade temporal de $O(\log n)$.

3. Especificações da aplicação

3.1. Dados de entrada:

- contacts.txt:

nContacts

firstName	lastName	phoneNumber	email	address
firstName	lastName	phoneNumber	email	address
firstName	lastName	phoneNumber	email	address

...

- settings.txt: número máximo de resultados a apresentar numa pesquisa.

3.2. Restrições:

- Contatos diferentes não podem ter nomes, números ou e-mail iguais.
- O número de telemóvel tem de ter 9 dígitos, o e-mail tem de ter pelo menos um '@' e um '.'.

3.3. Resultados esperados:

A aplicação deverá ser capaz de inserir/remover contatos fácil e intuitivamente, bem como pesquisar fluentemente um contato em específico e editar o mesmo.

4. Principais algoritmos implementados

O objetivo neste projeto era construir um mecanismo robusto e eficaz para a pesquisa numa lista de contatos. Para isso foi implementado o algoritmo *EditDistance*: um algoritmo de pesquisa aproximada de *strings* (o que foi apresentado nas aulas teóricas da unidade curricular).

O algoritmo *EditDistance*, tal como a sua versão melhorada, estão implementados em “StringSearchTools.cpp”.

4.1. Análise do algoritmo *EditDistance*

O algoritmo *EditDistance* recebe como argumentos duas *strings*:

- P – o padrão (palavra) a procurar;
- T – o texto onde procurar P .

O algoritmo retorna um inteiro:

- d – distância entre as strings.

O algoritmo em si consiste em construir uma matriz que é progressivamente preenchida com a distância entre P e T (onde se procura P). Após a conclusão do algoritmo a célula inferior direita da matriz apresenta a “distância de edição”, d , entre as *strings*.

A distância d é um número inteiro que corresponde ao número de operações necessárias para transformar o padrão P no texto T . Podemos concluir, que quanto menor a distância d , maior é a similaridade entre as *strings* P e T ; e ainda que, quando a distância d é zero, P e T são exatamente iguais.

Como o algoritmo analisa todos os caracteres, tanto de P como de T , a complexidade temporal deste é:

$$O(|P|.|T|)$$

O algoritmo faz uso de uma matriz bidimensional com $|P|$ linhas e $|T|$ colunas para guardar progressivamente a distância de edição d entre as strings. Pode-se então concluir que a complexidade espacial é igual à complexidade temporal, ou seja:

$$O(|P|.|T|)$$

4.2. Análise do algoritmo *EditDistance* melhorado

Existe uma versão otimizada do algoritmo *EditDistance* que, em vez de usar uma matriz bidimensional, usa um array de tamanho $|T|$ e duas variáveis auxiliares para guardar os valores relevantes para o preenchimento desse array.

Esta versão apresenta a mesma complexidade temporal, porque é igualmente necessário percorrer todos os caracteres de P e T . Contudo, como não é usada uma matriz bidimensional mas sim um array, a complexidade espacial é reduzida para:

$$O(|T|)$$

4.3. Análise do algoritmo de pesquisa da aplicação

De seguida é apresentado um resumo do mecanismo de pesquisa da aplicação.

Quando o utilizador, no menu principal, opta por fazer uma pesquisa, a interface de pesquisa dinâmica é inicializada. O padrão a pesquisar começa, como seria de esperar, vazio (*string* vazia). Portanto os resultados da pesquisa exibidos, quando esta é inicializada, serão a própria lista de contactos.

Cada contacto possui um atributo correspondente à distância d , distância esta que se relaciona com o padrão P a ser procurado. A partir do momento em que o utilizador começa a escrever no campo de pesquisa, o algoritmo é invocado sempre que um carácter é inserido/removido:

- para cada contacto presente na lista (de contactos), a distância à pesquisa P é atualizada, recorrendo ao método *updateDistanceToSearch*;
- a lista de resultados é ordenada segundo a distância de cada contacto, isto é, de forma a que os contactos com menor distância estejam no início da lista e os que apresentam maior distância fiquem no fim da lista.

Pseudocódigo para atualizar os resultados de pesquisa, tal como descrito em cima:

```
updateSearchResults(string search) {
    searchResults.clear

    if (search.isEmpty())
        foreach contact in contacts
            searchResults.add(contact);
    else
        foreach contact in contacts
            contact.updateDistanceToSearch(search);
            searchResults.add(contact);

    sort(ALL(searchResults), shortestDistanceContactFirst);
}
```

O método *updateDistanceToSearch*, surgiu de forma a resolver um dos problemas que encontrámos neste ponto do desenvolvimento da aplicação:

Como determinar a distância d , a que um determinado contato se encontra de uma *string*, fazendo uso do algoritmo *EditDistance*?

Na nossa aplicação, decidimos que a distância d de um contato seria o menor valor das distâncias que cada atributo desse contato apresenta em relação ao padrão P .

Para facilitar a explicação do algoritmo, consideram-se as seguintes abreviaturas e as suas respetivas designações:

- P - padrão a procurar;
- $|P|$ - retorna o tamanho de P ;
- $ED(x, y)$ - invocação da função *EditDistance*, que retorna a distância de edição entre x e y ;
- $TL(x)$ - invocação da função *toLower*, que retorna a *string* x em minúsculas;
- $str[j]$ - retorna a sub *string* de str , desde o primeiro carácter até j .
- $addressTokens[i]$ - retorna o token i , da morada de um contato.
- Dn - distância a que o nome de um contato se encontra do padrão P ;
- Dp - distância a que o telefone de um contato se encontra do padrão P ;
- De - distância a que o e-mail de um contato se encontra do padrão P ;
- Da - distância a que a morada de um contato se encontra do padrão P .

A distância d é portanto dada por:

$$d = \min(Dn, Dp, De, Da)$$

Onde Dn é calculado por:

$$Dn = \min(ED(P, firstName[|P|]), ED(P, TL(firstName[|P|])))$$

Ou, se o contato possuir ultimo nome:

$$Dn = \min(ED(P, firstName[|P|]), ED(P, TL(firstName[|P|])), ED(P, lastName[|P|]), ED(P, TL(lastName[|P|])), ED(P, name[|P|]), ED(P, TL(name[|P|])))$$

Finalmente, Dp , De e Da são dadas por:

$$Dp = ED(P, phoneNumber[|P|])$$

$$De = ED(P, email[|P|])$$

$$Da = \min(ED(P, addressTokens[0][|P|]), \dots, ED(P, addressTokens[n][|P|]))$$

Pseudocódigo para atualizar a distância d de um contato ao padrão P , tal como descrito em cima:

```

updateDistanceToSearch(string search) {
    // name comparison
    distanceToSearch = min(
        EditDistance(search, firstName.substr(search.length)),
        EditDistance(search, toLower(firstName.substr(search.length))));

    if (lastName)
        temp1 = min(
            EditDistance(search, lastName.substr(search.length)),
            EditDistance(search, toLower(lastName.substr(search.length))));
        temp2 = min(
            EditDistance(search, name.substr(search.length)),
            EditDistance(search, toLower(name.substr(search.length))));
        distanceToSearch = min(distanceToSearch, temp1, temp2);

    // phone comparison
    if (phoneNumber)
        distanceToSearch = min(
            distanceToSearch,
            EditDistance(search, phoneNumber.substr(search.length)));

    // email comparison
    if (email)
        distanceToSearch = min(
            distanceToSearch,
            EditDistance(search, email.substr(search.length)));

    // address comparison
    if (address)
        tokens = getTokens(address, " ");
        foreach contact in tokens
            temp = min(
                EditDistance(search, tokens[i].substr(search.length)),
                EditDistance(search,
                    toLower(tokens[i].substr(search.length)));
            distanceToSearch = min(distanceToSearch, temp);
}

```

4.3.1. Análise das complexidades temporal e espacial

Seja X o número de chamadas à função *EditDistance* existentes em *updateDistanceToSearch*:

$$X = (|nameTokens| * 4 - 2) + 2 + 2 * |addressTokens|$$

A complexidade temporal de *updateDistanceToSearch* será portanto:

$$O(X.|T|.|P|)$$

A complexidade espacial, por sua vez, será:

$$O(X.|T|)$$

Considere também que:

- C - conjunto dos contatos existentes na lista;
- $|C|$ - número de contatos existentes na lista de contatos.

A complexidade temporal e espacial do *sort* usado no método *updateSearchResults* é:

$$O(|C|.log(|C|))$$

Assim sendo, quando o padrão P não é equivalente a uma *string* vazia, as complexidades temporal e espacial serão, respetivamente:

$$O(|C|.X.|T|.|P| + |C|.log(|C|))$$

$$O(|C|.X.|T| + |C|.log(|C|))$$

Ou, especificando X :

$$O(|C|.((|nameTokens| * 4 - 2) + 2 + 2 * |addressTokens|).|T|.|P| + |C|.log(|C|))$$

$$O(|C|.((|nameTokens| * 4 - 2) + 2 + 2 * |addressTokens|).|T| + |C|.log(|C|))$$

6. Lista de casos de utilização

É esperado que o utilizador use a aplicação como uma ferramenta simples e eficaz de gestão de contactos.

Em baixo encontra-se um diagrama das funcionalidades disponibilizadas pela aplicação desenvolvida.

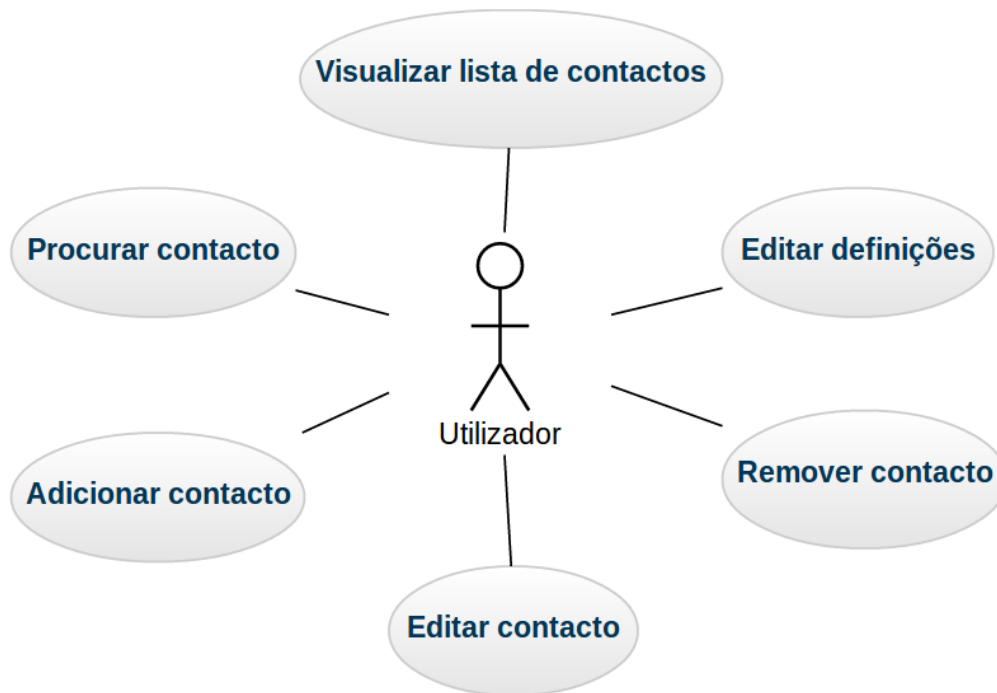


Fig. 1. Diagrama de casos de utilização.

A interface da aplicação foi desenvolvida para a linha de comandos. Para facilitar a navegação pela aplicação, todos os menus, sub-menus e respetivas solicitações de input para navegar entre estes estão implementados e encapsulados numa classe que gere toda a interface da aplicação em "Interface.cpp".

Houve a preocupação de desenvolver uma aplicação versátil, que funcionasse tanto em Windows como em Unix. Para isso foi implementada uma função dependente do sistema em que a aplicação for compilada, necessária para a pesquisa dinâmica. Esta implementação encontra-se em "ConsoleUtilities.cpp".

Relativamente à pesquisa dinâmica: é aconselhável não usar a consola do Eclipse, pois através desta a pesquisa dinâmica não funcionará; é aconselhável usar a linha de comandos se estiver a usar Windows, ou o terminal se estiver a usar um ambiente Unix.

De seguida é apresentado um diagrama relativamente aos diferentes menus e sub-menus da aplicação:

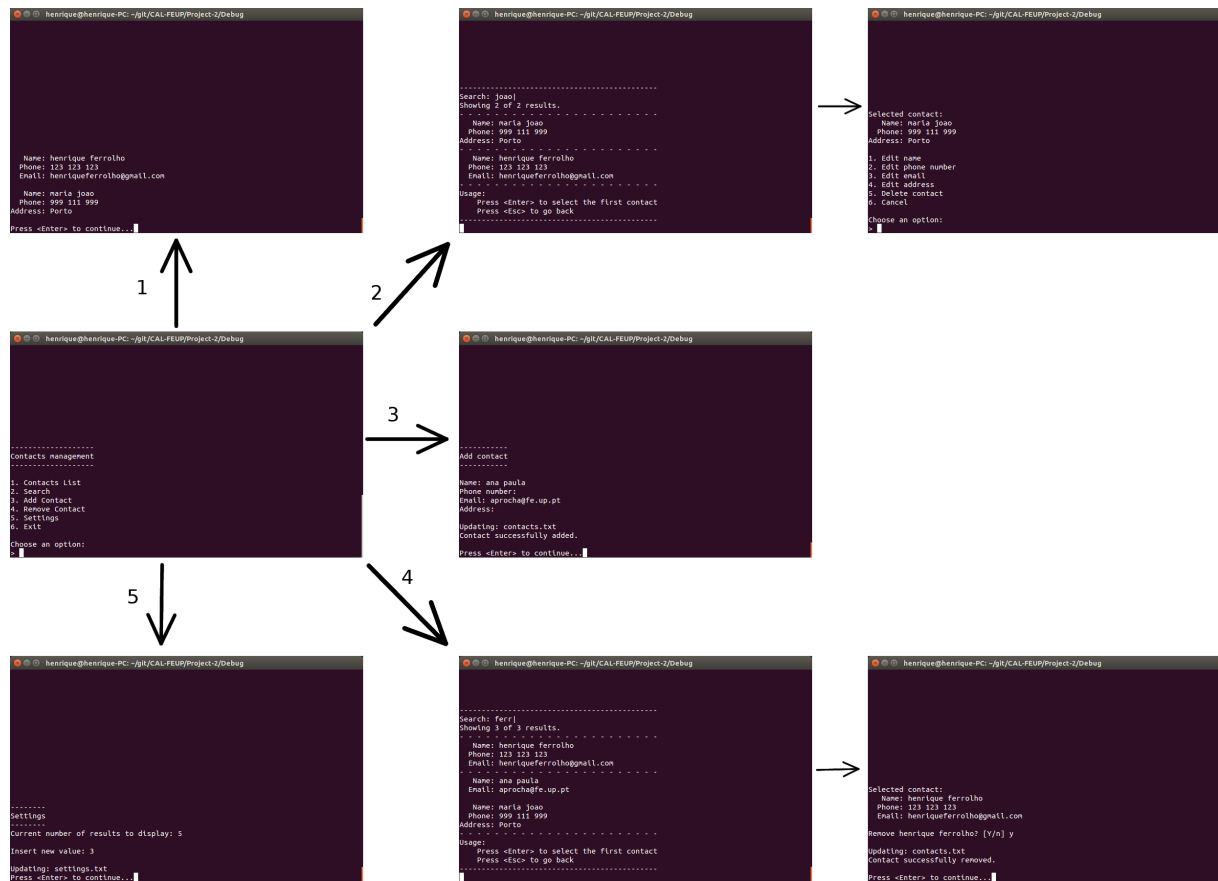


Fig. 2. Diagrama da interface da aplicação.

7. Conclusão

7.1. Dificuldades

A única dificuldade sentida foi na implementação dos algoritmos devido a um bug no pseudocódigo dos slides das aulas teóricas, mas foi facilmente ultrapassado após testar a aplicação em tempo de execução.

7.2. Contribuição no projeto