U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Conceção e Análise de Algoritmos

EasyPilot

Sistema de Navegação

Algorítmos em Strings

Grupo B, turma 3

Ângelo Miguel Tenreiro Teixeira, up201606516@fe.up.pt
Henrique Melo Lima, up201606525@fe.up.pt
Rui Pedro Moutinho Moreira Alves, up201606746@fe.up.pt

30 de Março, 2018

Índice

Introdução e Descrição do Problema

Nesta segunda parte do trabalho, foi proposto desenvolvermos implementações de um conjunto de algorítmos que operam em strings a fim de melhorar a experiência do utilizador, permitindo ao mesmo especificar as localidades entre as quais pretende viajar de forma textual e, com base no seu *input*, proceder ao seu processamento de forma a sugerir localidades, à semelhança de um sistema de navegação como o *Google Maps*.

Este problema pode ser dividido em três iterações, enumeradas a seguir.

Iteração 1: Verificação de Ocorrência de um Padrão num Texto

Nesta primeira iteração o objetivo é encontrar pelo menos uma ocorrência exata entre o padrão de procura inserido pelo utilizador e entre as várias descrições textuais das localidades presentes no mapa.

Desta forma, é possível detetar *input* do utilizador quando este tem conhecimento **exato** do local para onde / a partir do qual pretende viajar.

Iteração 2: Comparação entre um Padrão e um qualquer Texto

Nesta segunda iteração, o objetivo é encontrar, a partir de padrões não contidos no texto, ocorrências semelhantes a fim de prever o desejo do utilizador nos resultados da pesquisa.

Assim que a distância entre um padrão de procura do utilizador e a descrição textual de uma localidade pode ser quantificada, é possível implementar diversas heurísticas de decisão sobre o resultado dessa distância, a fim de tentar melhorar a experiência do utilizador.

Iteração 3: Implementação de um mecanismo de pesquisa

Nesta última iteração pretende-se, utilizando os algorítmos implementados nas iterações prévias, implementar um bom mecanismo de pesquisa com base nos padrões inseridos pelo utilizador, de forma a retornar resultados que correspondam o melhor possível aos seus desejos de pesquisa.

Estrutura do Programa

A fim de agregar todos os algorítmos que operam em strings, foi definido o namespace **StringSearch**, nos ficheiros *StringSearch.cpp* e *StringSearch.h*.

A restante estrutura do programa encontra-se tal e qual como descrita no relatório da entrega do trabalho anterior.

Solução Implementada

A descrição da solução implementada foi dividida em três partes, cada parte relativa a cada uma das iterações expostas no capítulo de Descrição do Problema.

Iteração 1: Verificação de Ocorrência de um Padrão num Texto

O problema de verificar a ocorrência de um padrão é um problema já densamente estudado, cuja formalização se encontra seguidamente apresentada:

Dados de Entrada

- **P –** Cadeia de caratéres representativa do padrão a pesquisar
- **T** – Cadeia de caratéres na qual é pretendido procurar o padrão **P**

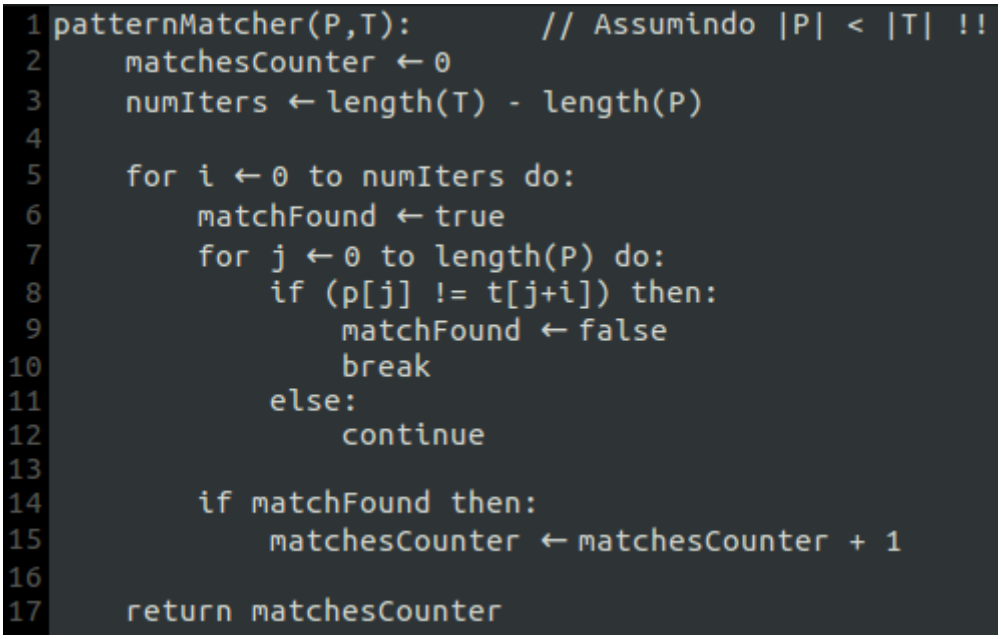
Dados de Saída

- **N** – Número de ocorrência do padrão **P** na cadeia de caratéres **T**

Restrições nos dados

- **|T|** ≥ |**P**| , o tamanho da cadeia de caratéres correspondente ao padrão deverá ter tamanho inferior ao tamanho da cadeia de caratéres correspondente ao texto onde a pesquisa é efetuada.

Uma primeira abordagem para solucionar este problema é, embora um pouco ingénua, procurar a ocorrência do padrão em todas as *sub-strings* de tamanho |P| do texto, sendo esta comparação feita caratér a caratér. O pseudo-código correspondente a este algorítmo encontra-se a seguir apresentado:



```
patternMatcher(P,T):        // Assumindo |P| < |T| !!
    matchesCounter ← 0
    numIters ← length(T) - length(P)

    for i ← 0 to numIters do:
        matchFound ← true
        for j ← 0 to length(P) do:
            if (p[j] != t[j+i]) then:
                matchFound ← false
                break
            else:
                continue

        if matchFound then:
            matchesCounter ← matchesCounter + 1

    return matchesCounter
```

Como é possível observar neste pseudo-código, o número de iterações realizadas é proporcional ao tamanho do padrão, que é procurado tantas vezes quanto o tamanho do texto. Por este motivo, este algorítmo torna-se ineficiente para cadeias de caratéres de comprimento elevado, visto que a sua **complexidade temporal** é **O(|P| * |T|)**.

Quanto à sua complexidade espacial, o espaço ocupado é independente dos dados de entrada, sendo portanto constante para quaisquer dados de entrada. Por este motivo, a **complexidade espacial** é **O(1)**.

É, no entanto, possível obter uma solução melhor, procedendo a um pré-processamento do padrão, de forma a evitar comparações desnecessárias entre o padrão e *sub-strings* do texto.

O algorítmo de **Knuth-Morris-Pratt** faz um pré-processamento do padrão, comparando o padrão com deslocações do mesmo, a fim de calcular a chamada **função prefixo**, quem contém valores correspondentes aos deslocamentos a efetuar no padrão de forma a evitar comparações desnecessárias.

O pseudo-código correspondente ao cálculo dos valores desta função prefixo encontra-se seguidamente apresentado:

```
kmpPrefix(P):
    prefix ← {}
    prefix[1] ← 0
    k ← 0

    for i ← 2 to length(P) do:
        while (k>0 and P[k+1] != P[i] do:
            k ← prefix[k]
        if P[k+1] == P[i] then:
            k ← k + 1
        prefix[i] ← k

    return prefix
```

O número de iterações desta rotina é diretamente proporcional ao tamanho do padrão, pelo que a sua **complexidade temporal** é linear relativamente ao tamanho do padrão, **O(|P|)**. Quanto ao espaço, o tamanho do vetor correspondente à função prefixo é proporcional ao tamanho do padrão, pelo que a **complexidade espacial** é também linear, **O(|P|)**.

Após esta etapa de pré-processamento, o algorítmo propriamente dito utiliza os valores desta função prefixo para proceder ao cálculo do número de ocorrência do padrão no texto. Devido ao cálculo da função prefixo, são agora evitados diversos deslocamentos do padrão desnecessários, como é possível observar no pseudo-código:

```
kmpMatcher(T, P):      // Assumindo |P| < |T| !!
    prefix ← kmpPrefix(P)
    matchesCounter ← 0
    k ← 0

    for i ← 1 to length(T) do:
        while (k>0 and P[k+1] != T[i] do:
            k ← prefix[k]
        if P[k+1] == T[i] then:
            k ← k + 1
        if (k == m) then:
            matchesCounter ← matchesCounter + 1

    return matchesCounter
```

O número de iterações desta rotina é diretamente proporcional ao tamanho do texto, pelo que a sua complexidade temporal é linear relativamente ao tamanho do texto, **O(|T|)**. Quanto ao espaço, para além do espaço ocupado pela função prefixo, o espaço ocupado é independente do tamanho dos dados de entrada, pelo que a **complexidade espacial** é constante, **O(1)**.

Por este motivo, a **complexidade temporal** do algorítmo é linear relativamente à soma dos tamanhos do padrão e do texto, **O(|P|+|T|)** e a **complexidade espacial** é proporcional ao tamanho do padrão (devido ao cálculo da função prefixo), sendo linear relativamente ao tamanho do padrão, **O(|P|)**, oferecendo um desempenho temporal bastante melhoror do que o algorítmo "ingénuo" previamente apresentado, à custa da utilização de um pouco mais de espaço.

É também possível uma implementação com base num autómato finito, melhor que a primeira abordagem. No entanto, optamos por não implementar esse algorítmo, visto que o algorítmo de Knutt-Moris-Pratt apresenta melhores resultados.

Iteração 2: Comparação entre um Padrão e um qualquer Texto

Após resolvido o problema de encontrar a ocorrência de um padrão num texto, a próxima iteração consiste em conseguir quantificar a semelhança entre um padrão e um texto.

A **distância de edição** entre duas cadeias de caratéres (em inglês, *edit distance*) corresponde ao menor número possível de alterações que são necessárias para transformar um texto T num padrão P. Estas alterações podem ser a **inserção** de um caratér, a **remoção** de um caratér ou a **substituição** de um caratér.

É possível calcular esta distância recursivamente da seguinte forma:

Sejam **i** um contador pertencente ao intervalo [0 .. |P|] e **j** um contador pertencente ao intervalo [0 .. |T|] e distance[i,j] a distância de edição entre P[1..i] e T[1..j], então:

- As condições de fronteira são distance[0,j]=j e distance[i,0]=i
- Os casos resursivos são, para i>0 e j>0:
  - P[i] = T[j] → distance[i,j] = distance[i-1,j-1]
  - Caso contrário, escolher a alteração menos custosa, que é o mínimo entre
    - 1 + distance[i-1,j], que corresponde a inserir P[i] imediatamente após T[j]
    - 1 + distance[i,j-1], que corresponde a eliminar T[j]
    - 1 + distance[i-1,j-1], que corresponde a substituir T[j] por P[i]

Através desta formulação recursiva do problema, é possível chegar a uma solução de programação dinâmica, como especificado no seguinte pseudo-código:

```
editDistanceMatrix(P,T):
    distance <- {{}}

    for i ← 0 to length(P) do:
        distance[i,0] ← i
    for j ← 0 to length(T) do:
        distance[0,j] ← j

    for i ← 1 to length(P) do:
        for j ← 1 to length(T) do:
            if (P[i] == T[j]) then:
                distance[i][j] ← distance[i-1][j-1]
            else:
                distance[i][j] ← 1 + min(distance[i-1][j-1],
                                         distance[i-1][j],
                                         distance[i,j-1])

    return distance[length(P)][length(T)]
```

Quanto à complexidade temporal, o número de iterações é proporcional à tamanho do padrão e do texto, pelo que a **complexidade temporal** é **O(|P| * |T|)**. Quanto ao espaço, esta implementação utiliza uma matriz para memorizar os valores, sendo a matriz de dimensão |P| * |T|. Por este motivo, a **complexidade temporal** é **O(|P| * |T|)**.

No entanto, é possível obter uma solução mais eficiente a nível de espaço, utilizando apenas um vetor ao invés de uma matriz, como evidenciado no seguinte pseudo-código:

```
editDistanceOptimized(P,T):
    distance <- {}

    for j ← 0 to length(T) do:
        distance[j] ← j

    for i ← 1 to length(P) do:
        oldValue ← distance[0]
        distance[0] ← i
        for j ← 1 to length(T) do:
            if (P[i] == T[j]) then:
                newValue ← oldValue
            else:
                newValue ← 1 + min(old,
                                   distance[j],
                                   distance[j-1])
            old ← distance[j]
            distance[j] ← newValue

    return distance[length(T)]
```

Nesta implementação, a **complexidade temporal** continua a ser **O(|P| * |T|)**, mas a **complexidade temporal** é otimizada para **O(|T|)**.

Iteração 3: Implementação de um Mecanismo de Pesquisa

Nesta última iteração procedeu-se a implementar um bom mecanismo de pesquisa com base nos algorítmos implementados nas iterações prévias.

Pesquisa Exata

Após seleção de “Pesquisa Exata” por parte do utilizador, é utilizado o algorítmo Knutt-Morris-Pratt para encontrar ocorrências do padrão de pesquisa nas localidades existentes na aplicação.

É de salientar que, ao contrário do algorítmo canónico, foi realizada uma otimização no algorítmo, sendo que este termina assim que é encontrada uma ocorrência em vez de calcular todas as ocorrências, o que poupa tempo de execução.

Pesquisa Aproximada

Foram realizadas 4 diferentes iterações a fim de apresentar bons resultados com base no input do utilizador

Iteração 1

Numa primeira tentativa foi avaliada a distância de edição do padrão de pesquisa a todas as localidades existentes na aplicação. Seguidamente, apresentava-se ao utilizador as 15 localidades mais idênticas ao padrão de pesquisa. Porém, esta solução revelou-se ineficaz, visto que devido a haver um elevado número de localidades com identificar textual “Unkwon”, quando era inserido um padrão de pesquisa pequeno a distância a estas localidades era muito reduzida, sendo a grande maior parte das localidades apresentadas ao utilizador aquelas que são “Unknown”.

Iteração 2

Seguidamente, tentou-se, após calcular a distância entre o padrão e cada identificador textual de localidade, subtrair a essa distância a diferença entre os seus tamanhos, de forma a detetar pesquisas como, por exemplo, em que utilizador pesquisa “vista” e que obter “Boavista”. Neste exemplo, a distância de edição seria 3. Ao subtrair a diferença do tamanho das cadeias de carateres (8 – 5 = 3), a distância total ficaria 0, sendo um bom *match*. No entanto, desta forma a pesquisa “Baisa” também obteria como bom *match* a palavra “Boavista”, sendo que não seria uma boa escolha.

Iteração 3

Numa terceira iteração tentou-se calcular a distância entre o padrão e todas as *substrings* de tamnho |P| pertencentes ao texto, utilizando como distância final a menor das distâncias de edição encontrada. Seguidamente, são apresentados ao utilizador os 15 melhores resultados de pesquisa.

No entanto, quando o padrão era demasiado distante de todos os nomes das localidades, eram ainda assim apresentados alguns resultados muito distantes do padrão de procura.

Procedeu-se, então, a descartar todos os resultados em que a distância era superior a 30% do tamanho do padrão (valor obtido experimentalmente, após tentativas com diversos valores). Desta forma, são garantidamente apresentados ao utilizador resultados idênticos ao padrão de pesquisa.

Iteração 4

Numa quarta e última iteração, adicionaram-se distritos a cada localização. Desta forma, é também possível procurar uma localidade por distrito.

No entanto, a ordem das palavras no padrão de procura introduzido pelo utilizador condicionava os resultados de pesquisa de forma indesejada.

Procedeu-se à separação do padrão nas suas palavras constituintes, a fim de remover esta limitação. De seguida, calcula-se a distância para cada uma das palavras (ao invés de o fazer para todo o padrão), sendo a distância total a soma das mesmas. Desta forma, os resultados de pesquisa melhoram substancialmente.

Casos de Utilização

Ao contrário da versão anterior do programa (primeira parte do projeto) é agora possível pesquisar as localidades pelo seu nome.

O utilizador pode escolher fazer esta pesquisa de forma exata ou aproximada, através de palavras chave (fazendo estas referência ao distrito e/ou ao nome da rua).

```
Please select the type of search you want to use:
1 - Exact search
2 - Approximate search
```

Desta forma, a utilização do programa torna-se mais simples e intuitiva.

Dificuldades encontradas no desenvolvimento do Trabalho

No decorrer do trabalho encontramos algumas dificuldades na seleção dos resultados a mostrar ao utilizador.

Esta seleção foi iterativamente refinada com base na análise dos resultados obtidos, obtendo bons resultados na versão final.

Conclusões

Todos os objetivos do trabalho foram cumpridos na totalidade:

- Foram implementados **todos os algoritmos** abordados nas aulas teóricas (quer de pesquisa exata, quer de pesquisa aproximada).
- Foram implementados **algorítmos para melhorar os resultados de pesquisa** apresentados ao utilizador (para pesquisa aproximada).
- Foram também utilizadas **estratégias de conceção de algoritmos** abordadas nas aulas teóricas, como a recursividade e a programação dinâmica (algorítmo de cálculo da distância de edição).
- Foram realizadas **análises de complexidade** temporal e espacial de todos os algoritmos implementados e foi apresentado o **pseudo-código** de todos os algoritmos, com base nas apresentações das aulas teóricas da disciplina.
- Foi também implementada uma **Interface de Utilizador** bastante intuitiva e fácil de utilizar, que permite facilmente recorrer a todas as funcionalidades implementadas, utilizando um sistema de menus.
- Os algorítmos relativos às operações em strings foram agregados num ***namespace*** (num só módulo).

O trabalho foi realizado de forma equalitária por todos os membros do grupo, sendo a maior parte do trabalho desenvolvido em reuniões presenciais nas quais foram discutidas várias ideias, foram planificadas as várias etapas do trabalho e onde se discutiu eficiência e implementação dos vários algoritmos. Por este motivo, todos os membros tiveram aproximadamente igual contribuição em todas as etapas do trabalho, não havendo nenhuma distinção evidente entre nenhum dos membros.

Bibliografia e outras Fontes de Referência

- Apresentações das Aulas Teóricas de Conceção e Análise de Algoritmo 2018, da autoria da Professora Doutora Liliana Ferreira, Professor Doutor João Pascoal Faria e Professor Doutor Rosaldo Rossetti.
- Knuth-Morris-Pratt algorithm, https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm
- Knuth-Morris-Pratt string matching, https://www.ics.uci.edu/~eppstein/161/960227.html
- Levenshtein Distance, https://en.wikipedia.org/wiki/Levenshtein_distance
- Edit Distance, https://en.wikipedia.org/wiki/Edit_distance
- Dynamic Programming Algorithm (DPA) for Edit-Distance, http://www.allisons.org/ll/AlgDS/Dynamic/Edit/