

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](mailto:up201606516@fe.up.pt)

Henrique Melo Lima, [up201606525@fe.up.pt](mailto:up201606525@fe.up.pt)

Rui Pedro Moutinho Moreira Alves, [up201606746@fe.up.pt](mailto:up201606746@fe.up.pt)

30 de Março, 2018

Índice

[Bibliografia e outras Fontes de Referência 3](#__RefHeading___Toc204_1963847657)

# 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 com 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

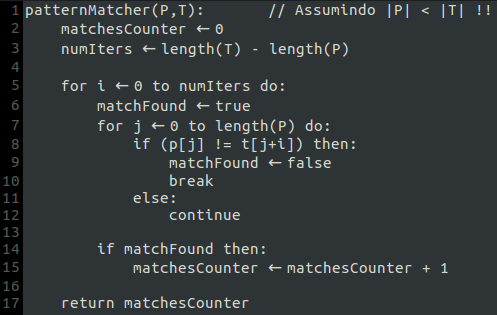
O problema de verificar a ocorrência de um padrão é um problema já densamente estudado, cuja formalização se encontra seguidamente apresentada:

* 1. **Dados de Entrada**
* **P –** Cadeia de caratéres representativa do padrão a pesquisar
* **T** – Cadeia de caratéres na qual é pretendido procurar o padrão **P**
  1. **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:

****

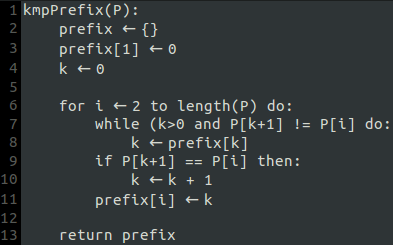
Como é possível observar neste pseudo-código, o número de iterações realizadas é proporcional ao tamanho do padrão, que é procurado tantas vezes quanto o tamanho do texto. Por este motivo, este algorítmo torna-se ineficiente para cadeias de caratéres de comprimento elevado, visto que a sua **complexidade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

Quanto à sua complexidadeespacial, 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.

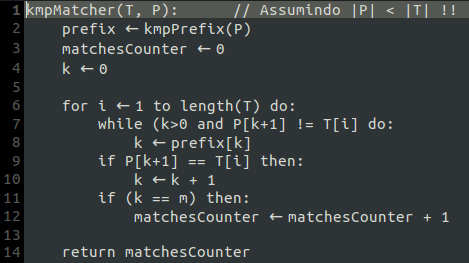
O algorítmo de **Knuth-Morris-Pratt** faz um pré-processamento do padrão, comparando o padrão com deslocações do mesmo, a fim de calcular a chamada **função prefixo**, quem contém valores correspondentes aos deslocamentos a efetuar no padrão de forma a evitar comparações desnecessárias.

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

****

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 evitadas diversos deslocamentos do padrão desnecessários, como é possível observar no pseudo-código:

****

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.

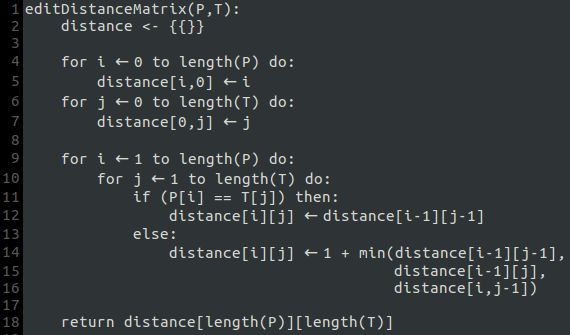
A **distância de edição** entre duas cadeias de caratéres (em inglês, *edit distance*) corresponde ao menor número possível de alterações que são necessárias para transformar um texto T num padrão P. Estas alterações podem ser a **inserção** de um caratér, a **remoção** de um caratér ou a **substituição**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

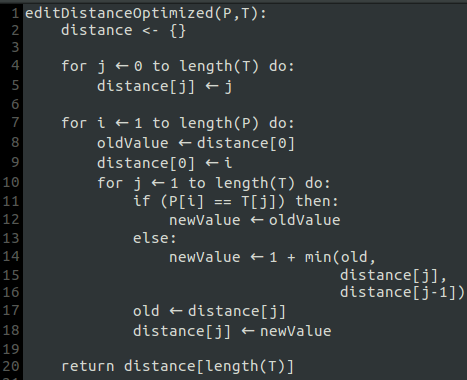
* As condições de fronteira são distance[0,j]=j e distance[i,0]=i
* Os casos resursivos são, para i>0 e j>0:
  + P[i] = T[j] → distance[i,j] = distance[i-1,j-1]
  + Caso contrário, escolher a alteração menos custosa, que é o mínimo entre
    - 1 + distance[i-1,j], que corresponde a inserir P[i] imediatamente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



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 um matriz, como evidenciado no seguinte pseudo-código:



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

# 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](https://en.wikipedia.org/wiki/Knuth–Morris–Pratt_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/>