Faculdade de Engenharia da Universidade do Porto

Concepção e analise de algoritmos

Relatório - Parte II

Smart Waste

Ana Santos up200700742@fe.up.pt
Bernardo Ramos up201505092@fe.up.pt
Pedro Reis up201506046@fe.up.pt

Professores
Ana Paula Rocha
Rosaldo Rosseti

May 20, 2017

Índice

1 Introdução

No âmbito da unidade curricular *"Conceção e Análise de Algoritmos"* foi-nos proposto para esta segunda parte do projeto a implementação de uma nova funcionalidade: a pesquisa de strings.

No contexto do Smart Waste, o utilizador tem agora a possibilidade de colocar resíduos num contentor à sua escolha do mapa, a partir do nome da rua adjacente, fornecendo assim um nível adicional de controlo sobre a interface. Para além disso, foi introduzido uma opção no menu onde é possível testar e comparar dois dos algoritmos implementados.

1.1 Identificação do problema

Na necessidade de aceder a um determinado contentor, é requerido primeiro saber exatamente onde este se encontra. No nosso projeto, cada contentor está localizado num cruzamento, representado como um vértice do grafo, e entre duas ruas, neste caso duas arestas. Surge assim como possibilidade de solução a pesquisa de contentores na rua que lhe está adjacente.

O problema fundamental na seleção das ruas é a comparação entre a string da rua e a string que queremos encontrar. Para isto analisamos três algoritmos diferentes: dois para caso a expressão que procuramos exista no mapa, e uma que devolve os resultados mais próximos caso esta não se encontre.

1.2 Formalização do Problema

1.2.1 Dados de entrada

Como nos algoritmos queremos avaliar a igualdade, ou na falta desta, a semelhança entre duas expressões, utilizamos como dados de entrada duas strings. Uma delas é a palavra, ou frase que o utilizador escreve, e a outra é a da rua que está a ser analisada.

1.2.2 Limitações de aplicação

Analisando a maneira com que o algoritmo foi implementado, existe uma possibilidade do utilizador pôr uma cadeia muito extensa que não tenha equivalente a uma aresta, a pesquisa pode ter de ser repetida muitas vezes, repetindo um processo já em si muito demorado. Isto pode levar a uma complexidade temporal muito elevada do processo inteiro, que é algo a evitar. No entanto, achamos que foi a melhor alternativa para garantir resultados mais próximos, à custa de eficiência em grafos muito grandes.

1.2.3 Situações de contorno

A possibilidade de se evitar a situação descrita anteriormente passava por apenas correr o ciclo uma vez, ordenar o *multimap* obtido com base nessa operação, e apresentar um número fixo dos primeiros resultados. Garantimos assim a proximidade das ruas, mas podemos descartar possibilidades muito válidas desnecessariamente.

1.2.4 Resultados esperados

Após a função de pesquisa ter sido executada, espera-se um de dois cenários:

• Encontra a string exata, e é(são) apresentada(s) a(s) rua(s) que contêm a expressão procurada na íntegra.

• Não encontra a string exata, por isso experimenta fazer a pesquisa aproximada e apresenta os resultados.

2 Solução

2.1 Técnicas de conceção

Para iniciarmos o processo de comparação é necessário que o utilizador escreva na consola a expressão que quer pesquisar no grafo. Após esta extração, damos início ao processo de comparação. Se a pesquisa exata não devolver nenhum resultado, prosseguimos fazendo uma nova comparação, com a mesma expressão, agora com a pesquisa aproximada. Se também esta não funcionar, estamos perante uma expressão que está longe de qualquer nome de rua do grafo.

A cada tipo de pesquisa está associada um de três algoritmos, a ser explicados em baixo.

2.2 Algoritmos de pesquisa exata

2.2.1 Naive

Este é o algoritmo mais simples que procura, caractere a caractere a sequência a pesquisar. Caso o caractere que estejamos a analisar na cadeia de texto onde procuramos seja igual ao primeiro da expressão introduzida, encontramos uma semelhança. Analisamos agora o caractere seguinte, tanto da expressão como do texto, tentando encontrar semelhanças em todos os caracteres da cadeia que temos. Se chegarmos ao fim da string introduzida sempre com semelhanças, significa que a encontramos no texto. Se num passo surgir uma diferença, repetimos o processo de comparação de novo, agora a partir do caractere seguinte ao inicial no texto a analisar.

Este algoritmo torna-se ineficiente caso o padrão seja comprido, uma vez que leva a muitas comparações redundantes, falha que pode ser minimizado com o algoritmo seguinte.

Pseudocódigo naive:

```
Function naive(H, I)
        m = |H| - |I| + 1
        matches = 0

        for i = 0 to m
                failed = false
                for j = 0 to |I|
                        if H[i+j] != I[j] then
                                failed = true
                                break
                end for

                if(!failed)
                        matches = matches + 1
         end for

    return matches
```

2.2.2 Knuth–Morris–Pratt

A principal diferença para o algoritmo anterior é que este requer alguma informação sobre a expressão que vai procurar, de modo a evitar comparações desnecessárias.

Faz-se antes da pesquisa um pré-processamento da string para saber a existência de padrões e assim poupar comparações desnecessárias. Armazenando esta informação num vetor, em caso de uma falha na comparação, ou seja, o caractere da string que está a ser analisado é diferente do caractere do texto, consegue-se saber quantas posições podemos avançar, garantindo assim que as comparações anteriores iriam ser semlhanças.

Este algoritmo é especialmente eficiente em sequências de caracteres não aleatórias, em que existe uma maior probabilidade da repetição de padrões e também em padrões mais compridos, ao contrário do algoritmo naive.

Pseudocódigo Knuth–Morris–Pratt - KMP:

```
Function KMP_Matcher(P, T, pi)
        n = |T|
        m = |P|
        q = 0
        s = |H| - |I| + 1
        matches = 0

        for i = 1 to s
                while q > 0 and P[q+1] != T[i]
                do q = pi[q]

                if P[q] = T[i-1] then q = q + 1
                if q == m then
                        matches++
                        q = pi[q]
        end for

return matches

Function computePrefix(P)
        k = 0
        m = |P|
        |pi| = m + 1
        for i = 0 to |pi|
                pi[i] = 0

        for q = 2 to m
                while k > 0 and P[k+1] != P[q]
                        do k = pi[k]
                if P[k+1] = P[q]
                then k = k + 1
                pi[q] = k
        end for
        return pi
```

2.3 Algoritmo de pesquisa aproximada

2.3.1 Pesquisa aproximada

Caso a string introduzida na verdade não esteja no texto, damos seguidamente a hipótese de apresentar resultados semelhantes. No entanto, é relevante que apenas sejam apresentados os resultados mais próximos, necessitando assim da quantificação de proximidade.

É neste contexto que entra o algoritmo de pesquisa aproximada. Funciona à base do número de alterações (substituições, inserções e eliminações) que uma cadeia de caracteres precisa para se tornar noutra. Utilizando a operação mais "barata", que mais rapidamente nos leva ao resultado pretendido, conseguimos determinar a proximidade das duas cadeias. Mais tarde organizamos os resultados por ordem crescente de alterações feitas para apresentar as strings mais semelhantes à introduzida.

Pseudocódigo pesquisa aproximada e do edit distance:

```
Function approximateSearch(H, I)
    while H not empty:
        V = H.line
        num = num + editDistance(I, V)
        numberOfWords = numberOfWords + 1
    end while
    return average num

Funtion editDistance(I, V)
    For j = 0 to |V|
        D[j] = j
    For i = 1 to |I|
        old = D[0]
        D[0] = i
        for j = 1 to |V| do
                if P[i] == V[j] then new = old
            else new = 1 + min(old, D{j}, d[j-1])
            old = D[j]
            D[j] = new
    return D[|T|]
```

2.4 Complexidade Temporal e Espacial

2.4.1 Naive

Assumindo para todos os casos um tamanho da expressão a procurar **S** e um tamanho para o texto **T**, é bastante simples ver a complexidade temporal deste algoritmo. Este percorre, no pior caso possível, toda a cadeia **T**, caractere a caractere de **S**. Em termos de código, isto é manifestado em um ciclo **for** com outro dentro dele, o que leva a uma complexidade final temporal:

$$O\left(|\mathbf{T}| * |\mathbf{S}|\right) \tag{1}$$

A complexidade espacial é igualmente simples, havendo apenas de relevo as duas strings que fazem parte do algoritmo, resultando numa complexidade:

$$O\left(|\mathbf{T}| + |\mathbf{S}|\right) \tag{2}$$

2.4.2 Knuth–Morris–Pratt

Este algoritmo tem uma metodologia mais trabalhosa que a anterior, visto que parte do cálculo reside no pré-processamento da string. Aqui, a tempo de execução da rotina é O (**S**), em que **S** é o tamanho da expressão que o utilizador introduz.

Em relação à componente de comparação direta, a maior parte dos cálculos são efetuados no **while** interno, podendo este ter uma complexidade máxima de O (**T**), sendo **T** o tamanho do texto onde vamos procurar a expressão.Somando as duas complexidades obtém-se uma final de:

$$O\left(|\mathbf{T}| + |\mathbf{S}|\right) \tag{3}$$

A gestão de espaço é mais uma vez simples de calcular, utilizando apenas um vetor de tamanho **S**, e duas strings, uma de comprimento **S** e outra **T**. A soma destas fica

$$O\left(2 * |\mathbf{S}| + |\mathbf{T}|\right) \tag{4}$$

2.4.3 Pesquisa Aproximada

Apesar de serem invocadas duas funções no código, a função principal onde existe uma maior complexidade é a que averigua a distância entre duas strings, sendo essa a única que irá ser analisada.

No segmento de código, reparamos que existe tal como no algoritmo Naive, dois ciclos **for**, um dentro do outro. Um deles é repetido **S** vezes, e o seguinte é **T** vezes. Isto leva a uma complexidade temporal de

$$O\left(|\mathbf{S}| * |\mathbf{T}|\right) \tag{5}$$

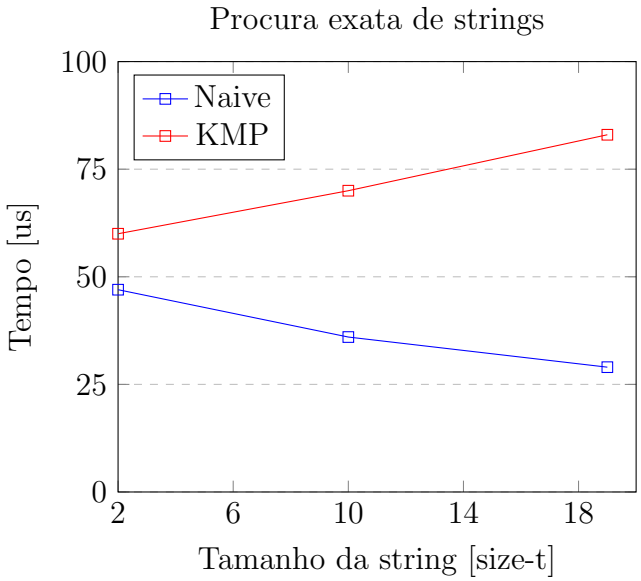
A complexidade espacial podia ser a mesma, mas foi feita uma optimização ao nível do espaço através de umas comparações adicionais que permitiram reduzir esta complexidade para

$$O\left(|\mathbf{T}|\right) \tag{6}$$

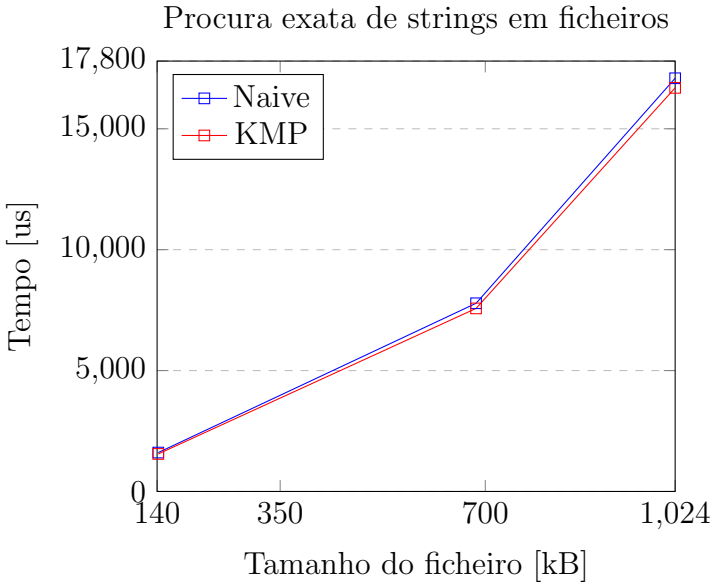
2.4.4 Análise de dados recolhidos

Após implementar os algoritmos, decidimos analisar lado a lado em termos temporais os dois únicos que podiam ser diretamente comparáveis: o Naive e o KMP.

Resolvemos então simular 100 casos de teste para cada um dos algoritmos, onde foram escolhidas três palavras a pesquisar de diferentes tamanhos: 2, 10 e 19 caracteres. Seguida de uma primeira observação, constatamos que as suspeitas inicias sobre qual seria o mais eficiente estavam erradas.



Em todos os casos o método Naive era consistentemente melhor que o KMP, pressupostamente mais eficiente. Reverificamos o procedimento, mas os dados não se alteravam. Só ao utilizar ficheiros de texto maiores com blocos de texto mais compridos é que havia uma clara vantagem para este último algoritmo. Para este analise foram realizados 100 testes com ficheiros de texto de diferentes tamanhos, 142 kB, 684.3 kB e 1024 kB, sempre pesquisando a mesma palavra de tamanho médio (9 caracteres).



Este facto deriva da necessidade do KMP de pré-processamento de cada string antes de ser comparada. No caso de comparações entre cadeias de caracteres pequenas, este cálculo anterior pode revelar-se sem sentido quando não há padrões que possam ser aproveitados. Só quando as strings e os textos a pesquisar aumentam de tamanho é que é mais nitidamente observável uma vitória computacional.

No âmbito do *Smart Waste*, como é possivel haver um elevado número de strings de grande tamanho, utilizamos este algoritmo invés do Naive, apesar de ambos serem boas opções.

2.5 Lista de casos de utilização

Estes algoritmos são utilizados em praticamente todos os sistemas em que o utilizador necessite de aceder a informação armazenada numa base de dados. Um exemplo disto é a plataforma digital de distribuição de música, SoundCloud. Na barra de pesquisas desta empresa, podemos ir escrevendo e, em que cada caractere introduzido ou removido, apresenta uma lista dos resultados mais próximos de maior interesse (neste caso, ou visualizações de cada música ou número de pesquisas).

Num contexto mais semelhante a este trabalho, podemos mencionar os mecanismos de pesquisa de ruas de todos os aparelhos de GPS (Global Positioning System). Para seleccionar um destino, o utilizador precisa de introduzir uma cadeia de caracteres. O algoritmo varia de empresa para empresa, mas um comum é utilizar a string e comparar com os destinos já anteriormente visitados, ou caso isso não se suceda, devolver os resultados mais aproximados, normalmente ordenados por ordem alfabética. São apenas dois exemplos de um mecanismo fortemente empregue.

Por fim, podemos também referir os teclados dos telemóveis, cuja grande maior parte tem um sistema de *auto-complete*, onde são evidentes algoritmos semelhantes aos usados nesta iteração.

2.6 Dificuldades encontradas

A componente de maior dificuldade foi a própria implementação dos algoritmos fornecidos no âmbito do nosso trabalho. Visto que queremos manipular o grafo de alguma maneira com a pesquisa, decidimos possibilitar o enchimento de um contentor em específico. Com isto veio a dificuldade de aceder aos vértices específicos com base em apenas o nome da rua, ou uma aresta a especificar do grafo.

Para resolver o problema referido, foi necessária uma alteração à estrutura dos grafos utilizado no primeiro projeto. Então foi acrescentada informação a cada aresta e assim, em cada uma delas é também guardado o seu nó de origem e o de destino, uma vez que, deste modo, é muito mais simples e eficiente o acesso aos nós adjacentes a cada aresta a partir unicamente do seu Id.

2.7 Trabalho desenvolvido por cada elemento

Semelhantemente à entrega anterior, subdividimos o projeto em três componentes. Relativamente à estruturação do código, a implementação dos algoritmos foi repartida pelo Ana Santos e Pedro Reis. A Ana foi também responsável pelo tratamento do input, geração de testes para a análise dos algoritmos e apresentação dos resultados na consola.

O relatório foi também tratado pelos mesmos dois membros, a sua elaboração foi maioritariamente pelo Pedro, com revisão e a elaboração do pseudocódigo pela Ana. O Bernardo Ramos foi o responsável pela conversão para Latex.

Ana Santos - 40 %
Pedro Reis - 40 %
Bernardo Ramos - 20%

2.8 Conclusão

Com o fim desta iteração do projeto introduzimos mecanismos de pesquisa de strings. Ao contrário da parte anterior do nosso projeto, a utilização destes algoritmo é algo tão mundano, que nem se dá um segundo pensamento sobre a sua existência.

Basta pesquisarmos algo num motor de pesquisa, plataforma de música, lista de aplicações para estarmos a utilizar na nossa vida diária estes sistemas. No entanto, reconhecemos que os algoritmos que implementamos são básicos, na medida que apenas têm em consideração ou o conteúdo ou a proximidade a uma string, enquanto que existem em outras interfaces mais variáveis que elevam a complexidade de pesquisa. Um excelente exemplo é a frequência com que a expressão é pesquisada.

Tudo isto contribui para que, em qualquer sistema, cada procura retorne, consistentemente, os resultados que mais possam interessar ao utilizador. Isto resulta numa melhoria para não só quem utiliza a plataforma, mas para a empresa por detrás, que fornece assim um melhor serviço.