CONCEPÇÃO E ANÁLISE DE ALGORITMOS

TripPlanner : Itinerários para transportes públicos

TEMA 11 – PARTE 2

Professor Regente: João Carlos Pascoal Faria

Professor das aulas Teórico-Práticas: Rosaldo José Fernandes Rossetti

Turma 3 Grupo C

Fábio Daniel Reis Gaspar | up201503823@fe.up.pt

Filipa Manita Santos Durão | up201606640@fe.up.pt

Pedro Miguel Sousa da Costa | up201605339@fe.up.pt

Índice

Casos de Utilização

Complementando as funcionalidades já previamente implementadas na primeira parte do projeto, foram implementadas novas funcionalidades relacionadas com o reconhecimento e aproximação de *Strings*.

Primeiramente, na funcionalidade de planeamento de viagens, foi adaptada a aplicação já existente, para aceitar no campo de pesquisa das paragens um nome em vez de um índice duma lista de paragens que era mostrada ao utilizador. O processo de pesquisa tem várias etapas. Numa primeira tentativa faz-se pesquisa com o nome exato da paragem. Seguidamente foi implementada a pesquisa aproximada. Assim, se o utilizador inserir um nome que não existe exatamente na lista de paragens, é apresentada uma lista das paragens com o nome mais semelhante ao *input* do utilizador. No caso de existir apenas uma paragem com um nome semelhante é assumida que é essa a paragem a que o utilizador se refere. Se não estiver presente na lista de paragens sugeridas a paragem que o utilizador pretende, é ainda possível realizar uma pesquisa avançada, que irá realizar uma pesquisa mais minuciosa sobre as paragens existentes. Esta pesquisa minuciosa não é executada automaticamente porque é muitas vezes desnecessária e é mais pesada que os outros algoritmos implementados.

Seguidamente foi implementada a pesquisa de paragens que passam numa certa linha. A introdução do nome da paragem tem o reconhecimento de strings aproximadas, no entanto, tal não foi aplicado ao nome das linhas, pois como são nomes muito pequenos (como 'B' ou '204') e não faria sentido pesquisar semelhantes. Se a linha introduzida não existir, o utilizador verá uma mensagem com tal informação.

Dados de Entrada

Tal como na Parte 1 do projeto, a informação relativa às paragens e linhas é lida dos respetivos ficheiros, **arestas.txt** e **nos.txt**.

A maior mudança dos dados de entrada face à primeira parte é o *input* do utilizador. Enquanto que na *Parte 1* era mostrado ao utilizador uma lista com todas as estações tendo um índice associado, e o utilizador apenas teria de inserir um número correspondente a estação desejada, nesta segunda parte passa a ser possível pesquisar pelas paragens a partir do seu nome, ou seja, o programa recebe Strings representando o nome das paragens e trabalha sobre essa informação.

Solução

Algoritmos implementados

Para a interpretação e validação do user Input foram implementados dois algoritmos, o *Knuth-Morris-Pratt Matcher*, pesquisa exata, e *Edit Distance*, pesquisa aproximada, também conhecido por *Levenshtein Distance*. Ambos recebem duas strings como input, a *string* P, padrão, a procurar e a *string* T, texto, onde é efetuada a pesquisa.

Existe ainda um terceiro algoritmo, implementado por nós, usado para uma pesquisa mais avançada. Contudo, o que esse algoritmo faz é apenas processar os dados, nome da paragem, T, e user input, P, antes de invocar o algoritmo *Edit Distance*. O pré-processamento consiste em "partir" as *strings* nas palavras que a compõe, eliminando artigos definidos e pronomes como "a, os, de, do", e fazendo comparação entre as palavras resultantes, tentando verificar se um *token* do *input* é suficientemente próximo de outro *token* do nome de uma estação.

Algoritmo de Knuth-Morris-PrattKnuth-Morris-Pratt

Comparativamente ao algoritmo *naive* de pesquisa que, basicamente, pesquisa P em cada deslocamento possível em T, este algoritmo efetua um pré-processamento do padrão a procurar para evitar comparações inúteis. Assim, a sua eficiência vai tanto depender da execução do algoritmo em si como do pré-processamento, denominado por *prefix-function*.

Pseudo-Código:

Kmp-Matcher

```
1   KMP-Matcher(P, T)
2       n = length(T)
3       m = length(P)
4       pi = Prefix-Function(P)
5       q = 0
6       result = 0;
7       for i = 0 to n do
8           while q > 0 and P[q] != T[i] do
9               q = pi[q -1]
10          end while
11
12          if(P[q] == T[i]) then
13              q++
14          end if
15
16          if(q == m=) then
17              result++
18              q = pi[q -1]
19          end if
20      end for
21  return result
```

Figura 1 - Kmp Matcher PseudoCode

Prefix-Function

```
1   Prefix-Function(P):
2       m = length(P)
3       pi[0] = 0
4       k = 0
5       for q = 1 to m do
6           while k > 0 and P[k] != P[q] do
7               k = pi[k -1]
8               if P[k] == P[q] then
9                   k++
10              end if
11              pi[q] = k
12          end while
13      end for
14  return pi
```

Figura 2 - Prefix-Function PseudoCode

Análise Temporal e Espacial

Relativamente à análise temporal, em ambas a partes, a eficiência vai depender do número de iterações do ciclo *while* interior. Dado que inicialmente o valor inicial de q é 0 e só pode ser incrementado no máximo N vezes (tal que N é o comprimento da string), o número de vezes que pode ser decrementado na instrução dentro do ciclo *while* é também no máximo n. Assim, o número de iterações do ciclo *while* no conjunto de todas as instruções do ciclo exterior será n. A complexidade total do algoritmo será, então $O(|P| + |T|)$, onde $|P|$ e $|T|$ representam o tamanho da string P e T, respetivamente. Note-se que quando é dito que o valor de q é decrementado dentro do ciclo while, isto deve-se ao facto dos valores presentes no array pi estarem entre os valores $[0, q-1]$.

Em termos espaciais o algoritmo apenas mantém um array de comprimento igual ao comprimento da string P, ou seja $O(|P|)$.

Algoritmo de Edit Distance (Levenshtein Distance)

Efetua uma pesquisa aproximada, calculando o "*grau de semelhança*" entre P e T, retornando a distância de edição entre eles. O algoritmo original usa uma matriz bidimensional de tamanho $|P| * |T|$, no entanto é possível otimizar para usar apenas uma matriz de tamanho $|T|$. Dito isto, foi apenas implementada e usada a versão otimizada.

Pseudo-Código:

```
1   editDistance(P, T):
2       n = length(P)
3       m = length(T)
4       cur[m+1]
5       cur[0] = 0
6       for i = 1 to m do
7           cur[i] = i
8       end for
9       for j = 1 to n do
10          pre = cur[0]
11          cur[0] = j
12          for i = 1 to m
13              temp = cur[i]
14              if P[i-1] == T[j-1] then
15                   cur[i] = pre
16              else
17                  cur[i] = min(pre + 1, cur[i] + 1, cur[i - 1] + 1)
18              end if
19
20              pre = temp;
21          end for
22      end for
23  return cur[m]
```

Figura 3 - EditDistance PseudoCode

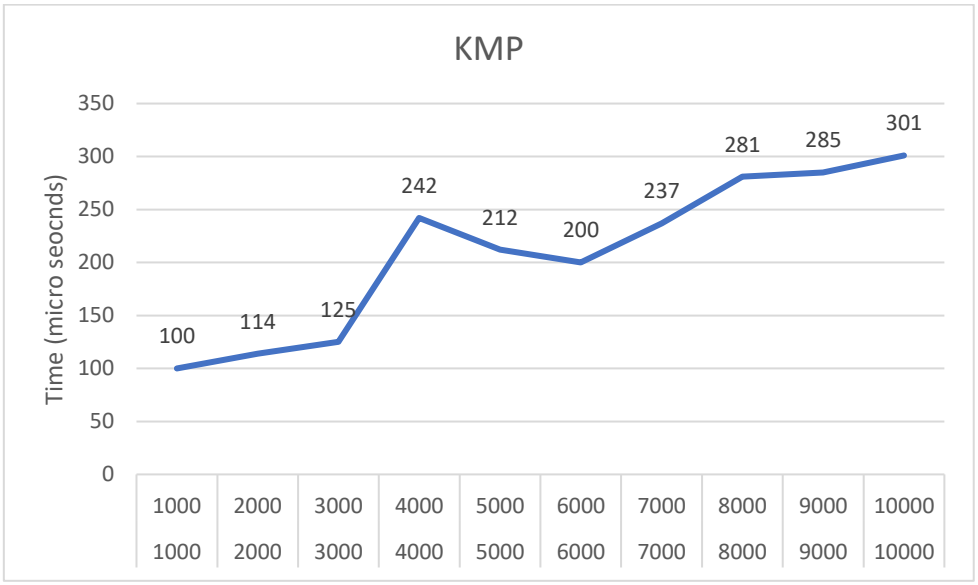
Análise Temporal e Espacial

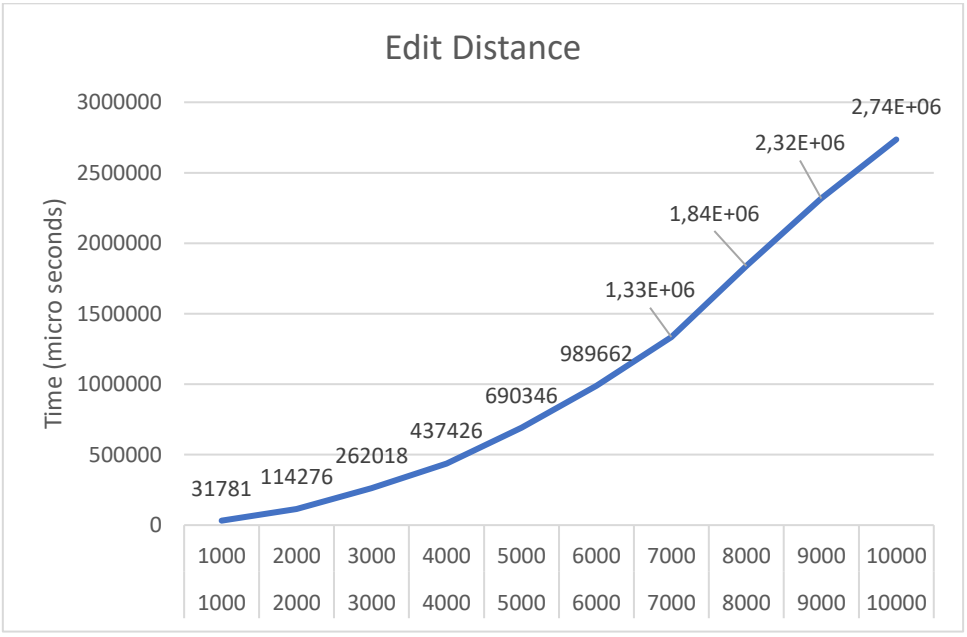
Relativamente à eficiência temporal, pelo pseudo-código apresentado em cima, podemos verificar que no total o algoritmo possui 3 ciclos, linhas 6, 9 e 12, sendo o 12 interior ao ciclo em 9. O número de iterações do primeiro ciclo é $|T|$, do segundo ciclo 9 $|P|$ e do ciclo interior $|T|$. Juntado tudo, obtemos a complexidade $O(|P| + |T| * |P|)$ ou simplesmente $O(|T| * |P|)$.

Como já foi referido, a complexidade espacial é então $O(|T|)$, dado que mantém apenas uma matriz unidimensional de tamanho igual ao comprimento da *string,* $|T|$.

Análise empírica

Após feita a análise teórica dos resultados, decidimos implementar alguns testes, com medição de tempo, para verificar se o tempo de execução dos algoritmos era o esperado. Fizemos diferentes testes, como mantendo o tamanho do *pattern*, $|P|$, constante e apenas variando o tamanho do texto, $|T|$. Também fizemos o oposto. Tal como seria de esperar, pela análise teórica, mantendo $|T|$ ou $|P|$ constantes não introduz nenhum comportamento especial dos algoritmos, já que a contribuição quer de $|P|$ quer de $|T|$ para a complexidade é a mesma. Por isso, abaixo apresenta-se apenas o teste em que ambos $|T|$ e $|P|$ foram variando em conjunto.





Como se pode ver pela análise dos gráficos, claramente o *KMP* consome muito menos tempo de processamento que o algoritmo de *Edit Distance*. Para o caso em que $|P| = |T| = 10000$, o *Edit Distance* tem um tempo de execução na ordem dos milhões de microssegundos, enquanto o algoritmo *KMP* mantem-se na ordem das centenas.

Este resultado é espectável. Em termos de processamento genérico, o KMP iria processar o *pattern* e por sua vez o texto com auxílio do *pattern*, como $|P| = |T| = 10000$, isso são cerca de 20000 operações. Já o *Edit Distance*, para cada carater em T, faz P iterações, resultando em $10000 * 10000$ iterações, pelo que se estimava que o *Edit Distance* demoraria 5000 vezes mais lento. Contudo, na prática verificou-se que foi 9000 vezes mais lento. No entanto, importa salientar que os resultados não foram sempre assim, o que se poderá dever às circunstâncias no momento de execução (escalonamento de processos, etc.).

Implementação

Para o efeito de string *matching*, tanto poderíamos usar o algoritmo *KMP (Knuth-Morris-Pratt Matcher)* como *Edit Distance*. Contudo, tal como verificado na análise acima, o algoritmo KMP é muito mais eficiente em termos temporais. No entanto, no contexto do problema, poucas serão as vezes em que o utilizador escreve o nome da paragem que pretende sem erros, tratando-se mais de um problema de aproximação do que *matching*. Assim, decidimos optar pela seguinte sequência na chamada dos algoritmos:

Utilizador fornece input:

1. Invoca-se *KMP* com o *input* (menor complexidade temporal que *Edit Distance*);
2. Se for obtido pelo menos um *match*, lista a informação;
3. Se não, chama-se o algoritmo *Edit Distance*;

Para a utilização do Edit Distance foi necessário especificar um critério de seleção. Caso contrário, seriam consideradas todas as paragens. Através de vários testes, chegámos à conclusão que uma *Edit Distance* máximo de 60% do comprimento da string fornecida pelo utilizador obtinha os resultados que queríamos.

Nas situações em que não ocorre nenhum *match*, foi aplicada uma rotina mais pesada em termos temporais para testar todas as possibilidades, considerando um possível erro elevado da parte do input do utilizador. Por exemplo, no nosso sistema existe a paragem "Hospital de São João", no entanto, se o utilizador introduzir "S Joao", que se pode considerar até certo ponto válido, nenhuma das pesquisas acima mapeará o *input* à paragem certa (com os nossos dados de teste, a nenhuma). Assim, para lidar com estes casos é efetuada a seguinte rotina:

1. Separar todos os nomes das paragens nas suas palavras, ou seja, criar *tokens*; Aplicar o mesmo processo ao *input* do utilizador;
2. Eliminar certas palavras bastante comuns entre os nomes, por exemplo artigos definidos, através da criação de um dicionário, otimizando assim a rotina;
3. Efetuar o algoritmo de *Edit Distance* a cada um dos *tokens* obtidos, e se houver um *match* bastante bom (*Edit Distance* normalmente inferior ou igual a 1), considerar a paragem que deu origem a esse *token* como uma hipótese. Como podemos verificar, em termos de complexidade este procedimento será bastante pesado, dado que teremos de efetuar este processo para cada paragem e para cada paragem efetuar tantas vezes quantas palavras simples existentes no nome da paragem (melhorando um pouco ao remover certas palavras especiais) e tantas vezes quantas as palavras no user *input*.

Se nenhuma destas formas funcionar, é pedido ao utilizador para tentar outra vez.

Nota: Tendo em conta o tamanho das *strings* com que se está a lidar, nomes de paragens, o nosso método de primeiro usar pesquisa exata e posteriormente pesquisa aproximada, não trás grandes ganhos, aliás, na maioria dos casos espera-se que o utilizador não consiga fazer um *match*, pelo que começar com *Edit Distance* seria melhor. Assim, poder-se-ia apenas utilizar pesquisa aproximada. No entanto, isso implicaria que apenas implementássemos um dos algoritmos estudados, e como tal, optamos por esta abordagem possibilitando a implementação, estudo e respetiva análise de ambos os algoritmos.

Manual de utilização

O programa tem um ecrã inicial onde é apresentado o nome da aplicação, “Trip Planner”. De seguida, são apresentadas três opções:

- Visualizar o mapa
- Planear a viagem
- Ver informações acerca de uma paragem

A visualização do mapa consiste em usar o *GraphViewer* fornecido pelos docentes, que foi integrado no projeto, e que ilustra todas as estações existentes no nosso mapa e como se ligam entre si. Os nós, que representam as estações, são identificados pelo nome da mesma. As arestas têm cores diferentes, sendo que cada linha, de metro ou autocarro, terá uma cor única. As arestas estão devidamente identificadas pelo nome da linha. As ligações a pé entre as várias paragens foram omitidas, uma vez que o seu número é muito elevado tornando o mapa ilegível. A janela deverá ser fechada dentro do programa.

A opção de planear a viagem é onde o utilizador indica o local de partida e chegada, e define um critério. Aqui o utilizador deverá introduzir o nome da paragem, inicialmente de partida e posteriormente de paragem. Dependendo do input do utilizador, podem acontecer os seguintes casos:

- O programa encontra um match ou uma única aproximação e utiliza esse resultado;
- O programa encontra múltiplos matches ou várias aproximações. Aqui serão listados os resultados encontrados, e o utilizador deverá selecionar qual pretende.
- Quando não é possível obter qualquer resultado aproximado, então o programa volta a pedir o input do utilizador.

Estando as paragens de partida e chegada selecionadas, falta o utilizador indicar um critério de planeamento da viagem. As opções são as seguintes:

- Número máximo de transbordos à escolha do utilizador
- Apenas trajetos sem troços a ser percorridos a pé
- O percurso com o menor preço
- O percurso com o menor tempo de viagem

Finalmente o percurso final é apresentado no ecrã de duas maneiras diferentes, um **modo detalhado** em que é apresentado passo a passo o percurso (em cada paragem que transporte e linha apanhar) e o **modo resumido** em que é apresentada a sequência das paragens percorridas. É também apresentado o **preço** e a **duração total** da viagem.

Além disso, o programa irá abrir uma janela com o *GraphViewer*, ilustrando toda a rede intermodal, tal como explicado acima, mas marcando as paragens do percurso a vermelho.

Conclusão

Todos os objetivos pretendidos do projeto foram concluídos: foram implementados todos os algoritmos de pesquisa de padrões em *strings* e integrados no projeto de forma a ser possível pesquisar as paragens pelo seu nome, mesmo que de forma aproximada.

Quanto às dificuldades na execução do projeto, o maior desafio enfrentado foi a compreensão do pseudocódigo presente nos slides das aulas teóricas, pois os índices dos vetores eram confusos de entender.

Quanto à contribuição dos diversos membros do grupo na realização do trabalho, a distribuição das tarefas foi feita de forma igualitária, tendo cada membro contribuído de igual forma para a conclusão deste projeto.

Bibliografia

- Thomas H. Cormen et al. Introduction to Algorithms, Third Edition. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.
- Slides fornecidos pelos docentes