U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Supermercado ao Domicílio

Tema 6 – Parte 2

Turma 2MIEIC01 - Grupo A

Tiago Lascasas dos Santos - up201503616

Leonardo Gomes Capozzi - up201503708

Ricardo Miguel Oliveira Rodrigues de Carvalho – up201503717

22/05/2017

Índice

Descrição do Tema

O tema deste projeto, “Supermercado ao Domicílio”, tem como objetivo a criação de um sistema que permita a uma cadeia de supermercados gerir as entregas ao domicílio de compras feitas pela internet.

Como continuação do projeto anterior, esta segunda parte do projeto acrescenta a funcionalidade de pesquisa por *strings*. Utilizando mapas semelhantes aos usados na primeira entrega, considera-se agora como dados principais de estudo os nomes dos mercados e das ruas do mapa.

Os nomes das ruas e supermercados foram, então, sujeitos a pesquisas exatas e aproximadas, por forma a facilitar ao utilizador a localização de pontos de interesse através do nome do mercado ou da rua. Neste projeto foi considerado, tal como proposto no enunciado, que cada mercado se encontra sempre no cruzamento de duas ruas.

Descrição das Soluções

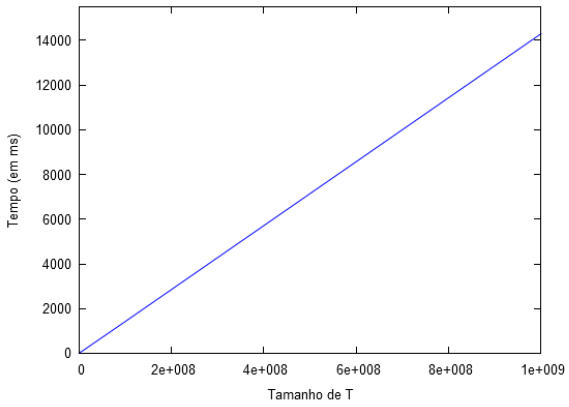
A partir da segunda parte do enunciado, foram isolados dois problemas, um envolvendo pesquisa exata de *strings* e outro pesquisa aproximada de *strings*, os quais se encontram seguidamente detalhados:

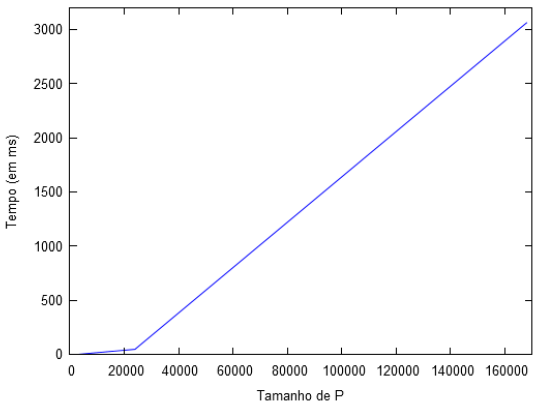
1. Pesquisa Exata de *strings*

De forma a pesquisar por um mercado ou por uma rua considerando uma pesquisa exata, prosseguiu-se à implementação do algoritmo de Knuth-Morris-Pratt, seguidamente descrito:

- **Input inicial** – texto, padrão, caseSensitive. Sendo;
  - **texto** (**t**)– um vetor com as strings que se pretende comparar.
  - **padrão** (**p**)– a *string* que se usa como referência para a comparação.
  - ***caseSensitive*** – valor booleano para definir se a comparação de carateres distinguirá, ou não, letras maiúsculas de minúsculas.
- **1ª parte** – Começa-se por declarar uma array de inteiros **v** de tamanho igual à string que estamos à procura, ou seja, a string **p**.
  - A seguir, percorre-se **p** e por cada letra de **p** escreve-se em **v** o índice a partir do qual se deve continuar a pesquisar caso não coincida a letra de **t** com a letra de **p**.
- **2ª parte** – Percorre-se **p** em paralelo com **t,** verificando se cada letra é igual nos dois. Caso alguma letra falhe, procura-se na array **v** o índice de **p** a partir do qual deveremos continuar a pesquisa. Se todas as letras de **p** coincidirem com as letras de **t,** quer dizer que este foi encontrado, mas se for percorrido o **t** todo sem fazer coincidir a última letra de **p,** quer dizer que **p** não existe em **t**.

Este algoritmo tem complexidade espacial e temporal O(|t| + |p|). Foi também realizado uma medição empírica da complexidade temporal do algoritmo, tendo sido considerado um caso em que o tamanho de t varia quando p é constante e vice-versa, obtendo-se, para valores crescentes de cada variável, os seguintes resultados:





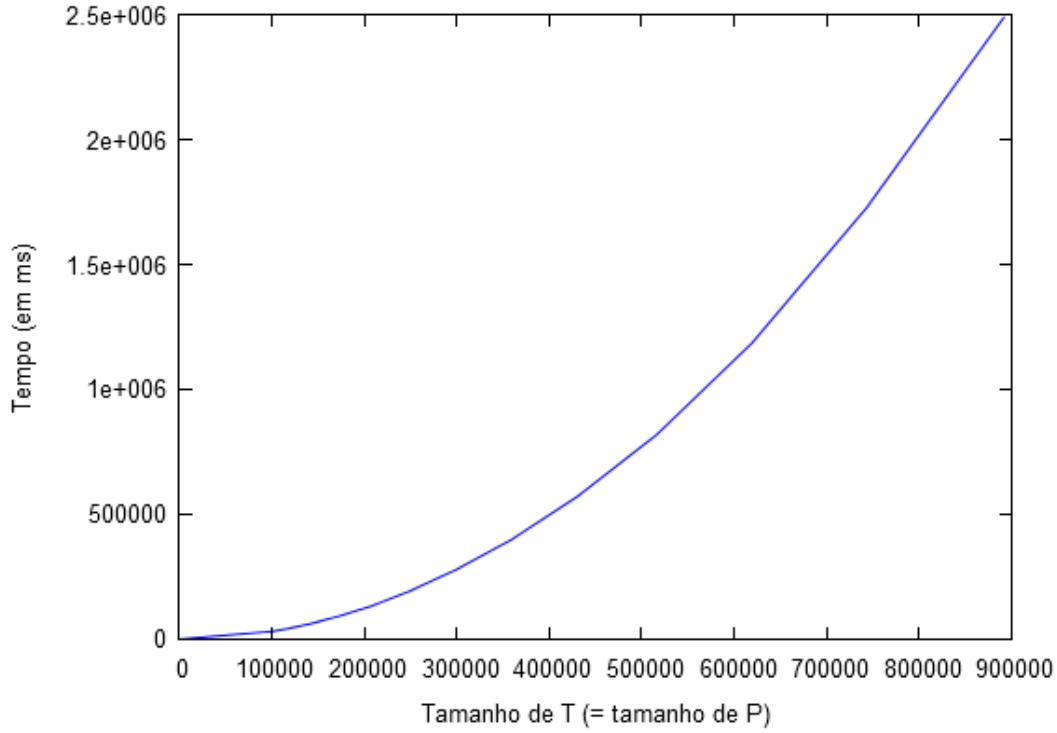
Como podemos averiguar, a complexidade do primeiro caso é $O(|\boldsymbol{t}|)$, enquanto que no segundo é $O(|\boldsymbol{p}|)$. Como são ambas lineares, somando as duas obtemos uma complexidade $O(|\boldsymbol{p}| + |\boldsymbol{t}|)$, confirmando, assim, a previsão analítica da complexidade temporal do algoritmo.

2. Pesquisa Aproximada de *strings*

Também é possível pesquisar aproximadamente por uma rua ou um mercado. Para tal efeito, é utilizado um algoritmo de distância de um padrão a outro baseado na distância de Levenshtein (aqui define-se distância como o número de operações de deleção, inserção ou substituição necessárias para a partir de um padrão obter o outro). Após obter a distância do padrão de input a cada string num vetor dado (nomes de ruas ou mercados) usando o algoritmo mencionado, as strings são associadas à respetiva distância e introduzidas numa fila de prioridades, que é devolvida pela função, ordenada de forma que a que tem menor distância esteja no topo. A versão do algoritmo de Levenshtein implementada pode, então, ser definida da seguinte forma:

- **Input inicial** – texto, padrão, *caseSensitive*. Sendo;
  - **texto** (**t**)– um vetor com as *strings* que se pretende comparar.
  - **padrão** (**p**)– a *string* que o utilizador quer pesquisar.
  - ***caseSensitive*** – valor booleano para definir se a comparação de carateres distinguirá, ou não, letras maiúsculas de minúsculas.
  - Inicialmente é criado um *array*, d, com o comprimento $|\boldsymbol{t}| + 1$ que é preenchido com valores sequenciais de 0 a $|\boldsymbol{t}| + 1$.
  - De seguida, para todos os valores de i desde 1 até $|p|$(inclusive) é atribuído a d[0] o valor de i e são percorridos os valores de j desde 1 até $|t|$(inclusive)
    - São, então, comparados os conteúdos de t e p nos índices j-1 e i-1, respetivamente, tendo em conta se a comparação deve distinguir letras maiúsculas de minúsculas. O resultado desta comparação influencia um parâmetro, *substitutionCost*, que é 0 caso seja verdade e 1 caso contrário.
    - A partir daqui é feita a seguinte atribuição:
      - $d[j] = min(d[j] + 1, d[j-1] + 1, last_{diagonal} + substitutionCost)$. Se:
        - $d[j] = d[j] + 1$, é realizada uma deleção
        - $d[j] = d[j-1] + 1$, é realizada uma inserção
        - $d[j] = last_{diagonal} + substitutionCost$, é realizada uma substituição, sendo last_diagonal o valor de $d[j]$ na iteração anterior.
  - No final, o resultado é o valor na última posição de d, isto é, $d[|\boldsymbol{t}|]$

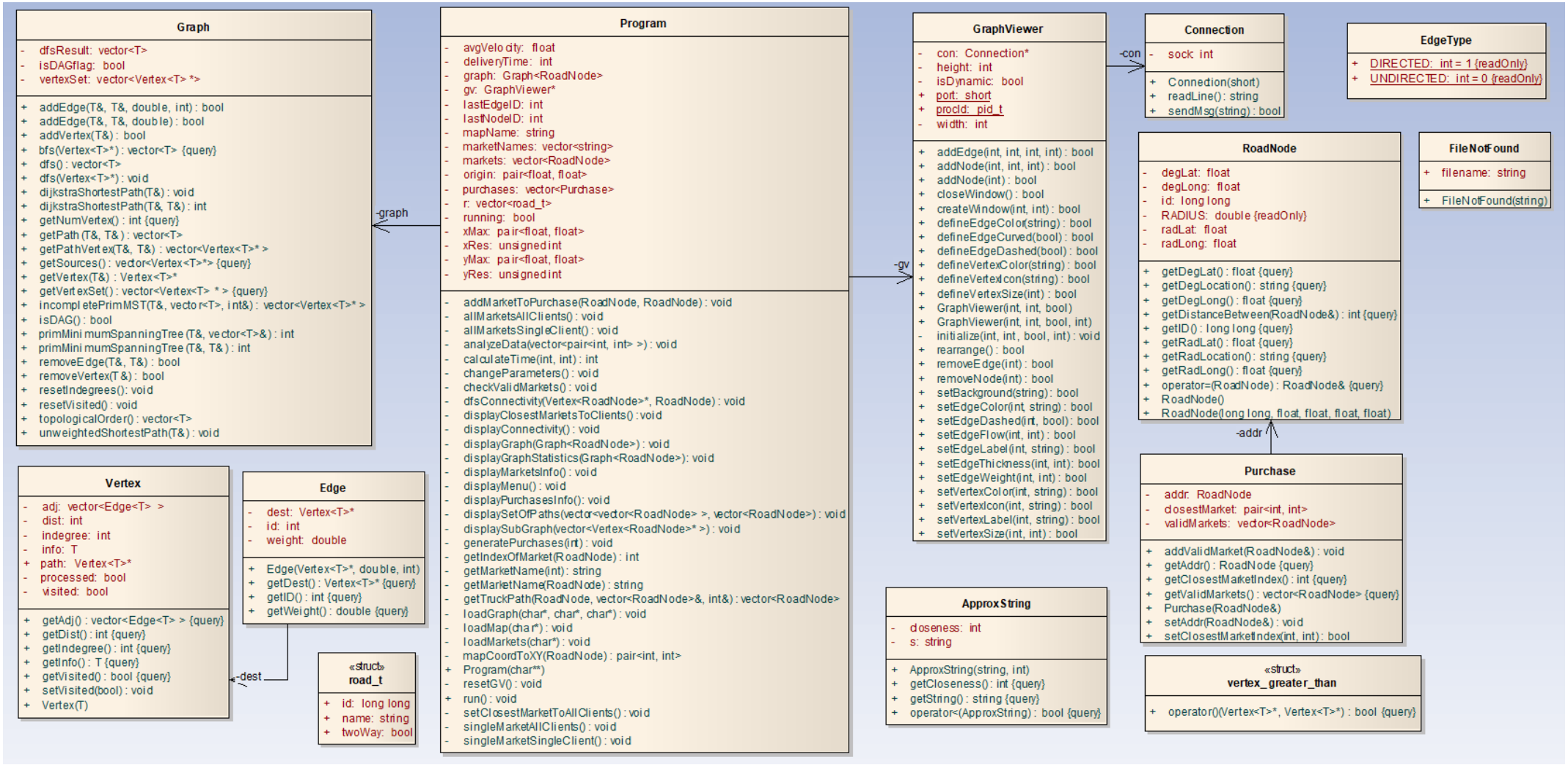
Este algoritmo tem complexidade espacial $O(|\boldsymbol{t}|)$ e complexidade temporal $O(|\boldsymbol{t}| \cdot |\boldsymbol{p}|)$. Foi também realizada uma medição empírica da complexidade temporal, considerando que $|\boldsymbol{t}| = |\boldsymbol{p}|$, ou seja, esperando obter-se uma complexidade $O(|\boldsymbol{t}|^2)$. Como podemos comprovar, o resultado obtido está em conformidade com essa asserção:



Considerações sobre a implementação prática das soluções

A implementação das soluções segue as descrições acima apresentadas, sendo que não foi introduzida complexidade extra para além da mencionada. A comparação sem considerar maiúsculas é realizada aquando da comparação de dois caracteres, não havendo qualquer conversão para uma *string* com todos os caracteres minúsculos. Quando um resultado é obtido, este serve de chave a um *std::unordered_map*, que é usado de modo a obter o mercado/rua correspondente ao resultado obtido. Esta estrutura possui um tempo de acesso médio $O(1)$, evitando-se assim introduzir um overhead extra na apresentação do resultado.

Diagrama de Classes



Lista de Casos de Utilização

Nesta parte do trabalho foi acrescentada uma opção no menu principal que adiciona um sub-menu contendo os vários tipos de pesquisa:

11. Pesquisa de ruas/mercados
    1. Pesquisa exata por rua
        - Funcionalidade que, com recurso ao algoritmo de pesquisa exata acima descrito, procura o nome de uma rua no grafo e, caso encontre, lista os mercados adjacentes, caso haja.
    2. Pesquisa exata por mercado
        - De forma semelhante à pesquisa por rua, procura entre os mercados e, se encontrar alguma correspondência, indica quais as ruas adjacentes a esse mercado.
    3. Pesquisa aproximada por rua
        - Pesquisa no grafo as ruas com nomes semelhantes ao dado. listando-as por ordem decrescente de proximidade ao padrão dado e limitando o resultado às dez ruas mais parecidas, mostrando também quais os mercados adjacentes a essas ruas, se houverem. Caso encontre a rua, mostra apenas essa rua, junto com qualquer mercado que lhe esteja adjacente.
    4. Pesquisa aproximada por mercado
        - De forma semelhante à pesquisa aproximada por rua, é efetuada uma pesquisa nos mercados, resultando na listagem destes por ordem decrescente de proximidade, também limitada aos dez resultados mais parecidos, juntamente com as duas ruas adjacentes a esses mercados. Caso encontre o mercado, mostra as duas ruas a ele adjacente.

Principais Dificuldades

No decorrer da segunda parte deste projeto, foram menos as adversidades, sendo que a maior dificuldade foi a necessidade de converter os ficheiros de ruas num formato sem caracteres especiais, por forma a compatibilizar e assegurar a correta leitura dos nomes das ruas. Foi também complicado efetuar a medição empírica da complexidade temporal dos algoritmos, visto que valores muito grandes de input causavam más alocações de memória.

Esforço de cada elemento do grupo

Tiago Lascasas dos Santos – 33%

Leonardo Gomes Capozzi – 33%

Ricardo Miguel Oliveira Rodrigues de Carvalho – 33%

Conclusão

Após a realização deste trabalho, pudemos concluir que os algoritmos de pesquisa em *strings* têm aplicações práticas extremamente úteis, e que se encontram extremamente otimizados. A pesquisa exata KMP é um algoritmo elegante, que consegue otimizar ao máximo a complexidade temporal da pesquisa, visto que depende apenas da soma do tamanho do texto e do padrão, não introduzindo uma complexidade acima da linear. Já o algoritmo modificado de Levenshtein, apesar de oferecer uma complexidade temporal aproximadamente quadrática, está otimizado em espaço de modo a ter uma complexidade linear que depende apenas do tamanho de um dos *inputs*, o que é particularmente bom para inputs grandes. Foi também possível observar, através dos resultados obtidos com os dados usados, que menores distâncias significam, realmente, uma maior parecença com o padrão que se quer encontrar, demonstrando, assim, uma prova visual da exatidão deste algoritmo.

Bibliografia e referências:

- Steven S. Skiena, The Algorithm Design Manual, 2th edition (Springer-Verlag London Limited, 2008), pág. 628–635
- Slides disponibilizados no Moodle pelos docentes
-
- Biblioteca ncurses: http://www.invisible-island.net/ncurses/ncurses.html
-