Conceção e Análise de Algoritmos

Tema 8 - Parte II

Turma 2MIEIC02 Grupo E

João Filipe Lopes de Carvalho - 201504875

José Pedro Dias de Almeida Machado - 201504779

Renato Alexandre Sousa Campos - 201504942

Nota: Para efeitos de facilidade de compreensão deste relatório, consideremos que:
Uma local tanto pode ser uma Rua, como uma Avenida, Praça, Largo ou Estação.
Um local tirado da lista de locais do grafo chama-se Text, e um local dado como input pelo utilizador chama-se Pattern.

Introdução

Neste trabalho foi nos proposto usar algoritmos já estudados, para melhorar o nosso programa realizado na Parte II, de forma a permitir ao utilizador escolher diretamente os locais iniciais e finais do percurso duma forma mais intuitiva. Isto é, agora o utilizador pode escolher por exemplo: "Praça dos Aliados" e "Estação da Trindade" como pontos iniciais ou finais do seu trajeto, em vez de escolher diretamente o número identificador do nó do grafo.

Implementação

Foram propostos para desenvolvimento e aplicação 2 algoritmos diferentes.

O primeiro, conhecido pelo nome Knuth–Morris–Pratt ou simplesmente KMP, foi usado para fazer pesquisa exata. Isto é, dada uma lista de ruas presentes no grafo, escolhe-se o nome de uma rua (Text) e compara-se com o input do utilizador (Pattern). O KMP deteta se o padrão Pattern se encontra no padrão Text. A aplicação deste método foi direta, e o nosso programa retorna ao utilizador todas as ruas nas quais o Pattern se encontra no Text.

O segundo, conhecido pelo nome de Distância de Levenshtein, é um algoritmo mais "avançado" que indica a distância de edição entre o Pattern e o Text. A distância de edição é calculada pelo número mínimo de operações que é preciso realizar para converter o Pattern no Text. Uma operação é defenida como uma das seguintes 3 situações:

- Remover uma letra da string;
- Adicionar uma letra à string;
- Substituir uma letra da string;

Na nossa implementação do algoritmo de Levenshtein cada uma das operações de edição tem um custo igual a 1 unidade. Sendo assim, para efeitos de exemplo, para converter a string "Paralelo" na string "Papel", é suficiente remover as letras "a, l, o" e substituir a letra "r" pela letra "p", totalizando 4 operações, ou seja, a distância de edição será igual a 4.

No entanto aplicar este algoritmo diretamente, poderia dar resultados inesperados, por exemplo, dada o Text = "Estação de São Bento" e umo Pattern.1 = "Bento" e umo Pattern.2 = "Estação da Rita", a distância de edição no primeiro caso seria 15 e no segundo caso 8, ou seja o Pattern.2 seria melhor classificada do que o Pattern.1;

Para resolver este problema, várias tentativas foram feitas e a que demonstrou melhores resultados, observados pela execução do programa, foi a seguinte implementação:

- antes de aplicar o algoritmo de Levenshtein todas as ruas do grafo têm de ser analisadas, pois os dados contêm locais repetidos e locais sem nome. Sendo assim, todos os locais repetidos e vazios são eliminados e um novo vetor de locais é obtido. Para que este

preprocessamento não afete o desempenho do programa, este foi posto a correr numa thread paralela à thread principal, desta forma o utilizador nem se apercebe de que o preprocessamento acontece. De seguida, o Pattern é posta à prova contra todas as Text. De forma a obter resultados minimamente parecidos, foi desenvolvido o algoritmo "aproximate_matching" que por sua vez usa o algoritmo de Levenshtein.

Pseudo-código – Aproximate_matching

```
int aproximate_matching(string pattern,string text){
int totalEditDistance = 0, currentTotalDistance = 0, currentEditDistance;
vector<string> textSplitted = splitText(text);
vector<string> patternSplitted = splitText(pattern);
if (textSplitted.size() != patternSplitted.size()){
        totalEditDistance += abs(textSplitted.size() - patternSplitted.size());
}
eliminateRedudantWords(textSplitted);
eliminateRedudantWords(patternSplitted);
for (Pattern in patternSplitted){
        for (Text in textSplitted ){
                currentEditDistance = LevenshteinDistance(Pattern,Text);
                if (currentEditDistance == 0){
                        currentTotalDistance = currentTotalDistance/2;
                        break;
                }
                currentTotalDistance += currentEditDistance;
        }
        totalEditDistance+=currentTotalDistance;
}
 return totalEditDistance;
}
```

O algoritmo splitText, usado pelo *Aproximate_matching* apenas converte uma string num vetor de strings, separando as strings pelo caractér ' ' (espaço);

O algoritmo eliminateRedundanteWords, usado também pelo *Aproximate_matching* permitiu-nos obter muito melhores resultados, mas tem a desvantagem que a sua implementação depende do contexto da aplicação. No nosso caso ele é usado para remover palavras como "Avenida", "Rua", "Largo", "de", "dos", "da" das strings, para que estas palavras não sejam consideradas pelo algoritmo de Levenshtein.

Desta forma, no caso anterior (Text = "Estação de São Bento" e Pattern.1 = "Bento" e Pattern.2 = "Estação da Rita") o Pattern.1 seria classificada com distância 5 em vez de 15 e o Pattern.2 com distância 9 em vez de 8. Sendo assim, o Pattern.1 já é cosiderada mais parecida com o Text do que o Pattern.2;

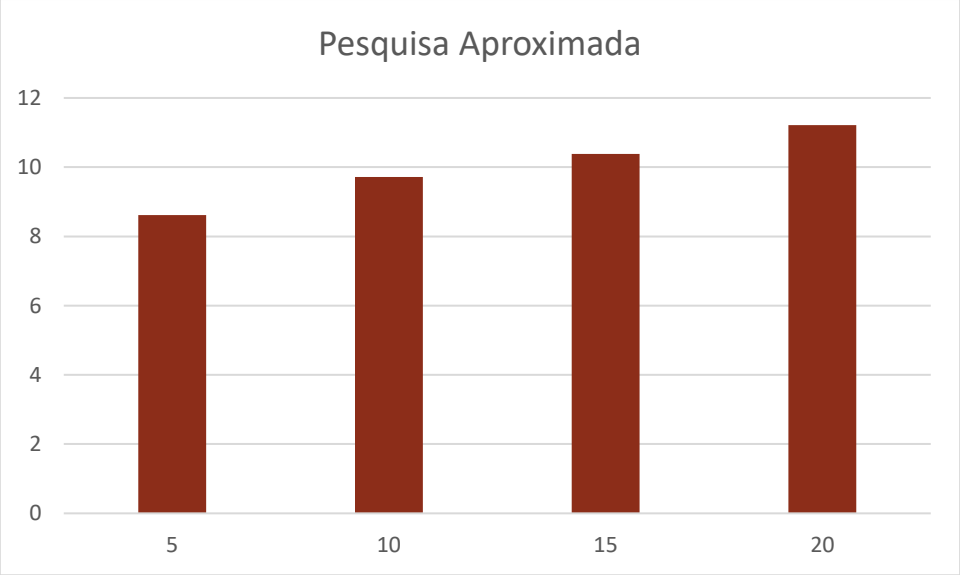
Análise Teórica de Complexidades

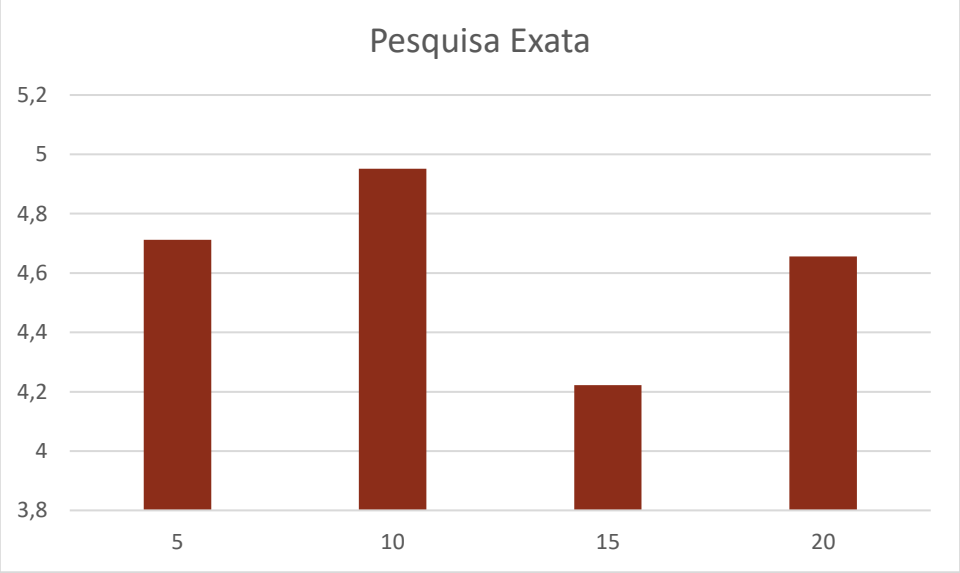
O algoritmo **KMP,** para testar se um Pattern se encontra numo Text, necessita de calcular uma vez o array auxiliar, O(|Pattern|), e de executar o KMP em si, O(|Text|), logo a sua complexidade temporal é de O(|Pattern| + |Text|);

Caso o algoritmo seja aplicado no contexto do problema, seja N o número de Texts diferentes a analisar, o array auxiliar só precisa de ser calculado uma vez, O(|Pattern|), e a comparação em si tem de ser feita N vezes, O(N |Text|), totalizando uma complexidade temporal O(|Pattern| + N |Text|).

O algoritmo **Aproximate_matching**, faz uso do algoritmo splitText duas vez, que corre em tempo O(|Pattern| + |Text|). De seguida usa o algoritmo eliminateRedudantWords também 2 vezes, que corre em tempo médio O(|Pattern| + |Text|). O algoritmo LevenshteinDistance tem complexidade temporal O(|Pattern|/numberOfPatterns * |Text|/numberOfTexts) e é chamado em média NumberOfPatterns * NumberOfTexts vezes. Logo a complexidade temporal média do **Aproximate_matching** é O(2|Pattern| + 2|Text| + |Pattern| * |Text|);

Análise Empírica das Complexidades





Conclusões

Apesar da existência de algoritmos sofisticados na procura de certos padrões dentro de strings, como o KMP, e de algoritmos como o de Levenshtein que retornam uma distância de edição entre textos, a aplicação destes ao mundo real não pode ser feita diretamente. Apesar das tentativas feitas na procura duma solução decente, capaz de retornar as strings provavelmente mais desejadas pelo utilizador, sabemos que a solução encontrada está longe de ser óptima. No entanto, é quase sempre melhor do que a aplicação direta do algoritmo.