Relatório do projeto de Conceção e Análise de Algoritmos

**Trabalho 2**

# Tema 3: Encontra a Palavra

****

**Turma 2 Grupo F:**

Joel Márcio Torres Carneiro 201100775 ei11053@fe.up.pt

Daniel Arménio Silva Mendonça 200906506 ei12167@fe.up.pt

Francisco Teixeira Lopes 201106912 ei11056@fe.up.pt

**Breve descrição da aplicação:**

Este é um jogo para dois jogadores, *A* e *B*. Inicialmente é colocada na mesa uma palavra de *N* letras (*N*<5). Os jogadores tentam construir novas palavras a partir da palavra da mesa, isto é, palavras que contenham a palavra da mesa. Assim, os jogadores jogam alternadamente adicionando uma letra à palavra da mesa (no início ou no fim).  O jogo termina quando um dos jogadores não consegue adicionar qualquer letra à palavra da mesa, e perde o jogo. O jogador vencedor deve mostrar a palavra em que estava a pensar.

Esta aplicação consiste em implementar o jogo com um jogador automático, que jogará contra um humano.

 São aplicados algoritmos de pesquisa de strings adequados, e é considerada a existência de um ficheiro de texto que contém as palavras conhecidas que podem ser usadas no jogo.

Formalização:

D = dicionário

L = letra introduzida pelo utilizador

I = palavra inicial da aplicação

P = palavra construída na aplicação

G = palavra em jogo

Dados de entrada:

D(dicionário)

I(palavra inicial)

L(letra introduzida)

Dados de saída:

P(palavra final)

Objetivo:

Formar uma nova palavra a partir da que está em jogo para que o oponente não possa jogar.

Restrições:

A palavra formada pela adição de uma nova letra tem de pertencer ao dicionário.

(G(palavra em jogo) + L(letra introduzida))D(dicionário)

A palavra inicial tem de ter menos do que cinco letras

I(palavra inicial) < 5

**Descrição da solução encontrada:**

Para a resolução deste problema o grupo decidiu implementar 6 classes: City, Vertex, Edge, EdgeType, GraphViewer e Graph.

**City:**

Nesta classe a cidade é “criada”. Cada cidade tem um nome, um tempo de visita, se tem aeroporto e/ou alojamento ou não e o tempo de viagem. É possível aceder a cada uma das especificações através de funções criadas na aplicação.

**Vertex:**

Nesta classe, através das propriedades do grafo, e das funções então criadas, é possível saber-se se uma cidade já foi visitada e assinala-la como tal.

**Edge:**

Nesta classe é possível saber-se os tempos de viagem de umas cidades para as outras. Desta forma é possível analisar-se qual a cidade mais próxima.

**Graph:**

Aqui são reunidos todos os resultados das restantes classes e tratados de forma a ser possível obter uma solução do problema.

**EdgeType:**

Nesta classe são enumerados os tipos de arestas. Existem então dois tipos: arestas com e sem direção.

**GraphViewer**:

Esta classe trata de representar um grafo criado previamente. Todas as suas funções retornam um booleano a indicar se a sua execução decorreu ou não com sucesso.

Para a obtenção de uma solução foi necessário utilizar algoritmos que nos ajudassem a calcular o pretendido. Desta forma foi usado o algoritmo Floyd-Warshall para calcular o menor caminho entre cidades através de uma matriz recebida como entrada.

\theta(|V|^3)O algoritmo de Floyd-Warshall recebe como entrada uma [matriz de adjacência](http://pt.wikipedia.org/wiki/Matriz_de_adjac%C3%AAncia) que representa um grafo orientado e valorado. O valor de um [caminho](http://pt.wikipedia.org/wiki/Caminho) entre dois [vértices](http://pt.wikipedia.org/wiki/V%C3%A9rtice) é a soma dos valores de todas as [arestas](http://pt.wikipedia.org/wiki/Aresta_(teoria_dos_grafos)) ao longo desse caminho. As arestas do grafo podem ter valores negativos, mas o grafo não pode conter nenhum [ciclo](http://pt.wikipedia.org/wiki/Ciclo_(teoria_dos_grafos)) de valor negativo. O algoritmo calcula, para cada par de vértices, o menor de todos os caminhos entre os vértices. Por exemplo, o caminho de menor custo. A ordem de complexidade é .

Foi também utilizada uma abordagem greedy para obter uma melhor solução. Um algoritmo greedy é um [algoritmo](http://translate.googleusercontent.com/translate_c?depth=1&hl=pt-PT&prev=_dd&rurl=translate.google.pt&sl=en&tl=pt-BR&u=http://en.wikipedia.org/wiki/Algorithm&usg=ALkJrhhLsQ6VkpgfW6yakw46z8llGnIxbg) que segue a [resolução de problemas](http://translate.googleusercontent.com/translate_c?depth=1&hl=pt-PT&prev=_dd&rurl=translate.google.pt&sl=en&tl=pt-BR&u=http://en.wikipedia.org/wiki/Problem_solving&usg=ALkJrhh4a63o_EJaLpcbt3z3CxK2deUoag) [heurísticos](http://translate.googleusercontent.com/translate_c?depth=1&hl=pt-PT&prev=_dd&rurl=translate.google.pt&sl=en&tl=pt-BR&u=http://en.wikipedia.org/wiki/Heuristic_(computer_science)&usg=ALkJrhgwkhMEvYyCaLg6Crg2gc86NzbmRA) de forma a fazer a escolha ideal localmente em cada fase, com a esperança de encontrar um valor ótimo global.

Em muitos dos problemas as estratégias gananciosas não produzem soluções ideais sendo então aproximadas. Ainda assim, uma estratégia gananciosa pode produzir soluções localmente ótimas que se aproximam de uma solução ótima global num tempo razoável.

Desta forma, foi necessário a utilização destas abordagens. Para obtermos uma solução em pouco tempo só mesmo utilizando um método ganancioso o torna possível. Queremos que seja relativamente rápida a solução, caso contrário seria demorado encontrar a solução ótima no meio de tantas possíveis pois teriam de ser analisados todos os caminhos. No caso de umas férias pequenas com poucos locais de destino seria viável mas se fosse um número maior de cidades começava a tornar-se muito demorado para que se obtivesse a solução.

**Diagrama de Classes UML**

**Lista de Casos de Utilização Identificados para a Aplicação**

1. Criar grafo
2. Criar cidade
3. Criar nós
4. Atribuir custo
5. Atribuir tempo de viagem
6. Atribuir alojamento
7. Atribuir aeroporto
8. Atribuir local de interesse
9. Cancelar viagem que não dê tempo suficiente
10. Eliminar viagem que não dê tempo suficiente

**Principais dificuldades encontradas na resolução do trabalho**

O desenvolvimento desta aplicação trouxe algumas adversidades na escrita do código e na sua estruturação. Contudo, com alguma reflecção foram ultrapassadas, como por exemplo fazer a divisão de classes e a utilização do graph e das suas propriedades. Após a execução deste projeto ficaram mais clarificadas as noções de graph, edges weight, vertex, entre outros, o que nos trás vantagens para podermos trabalhar de melhor forma em qualquer projeto futuro.