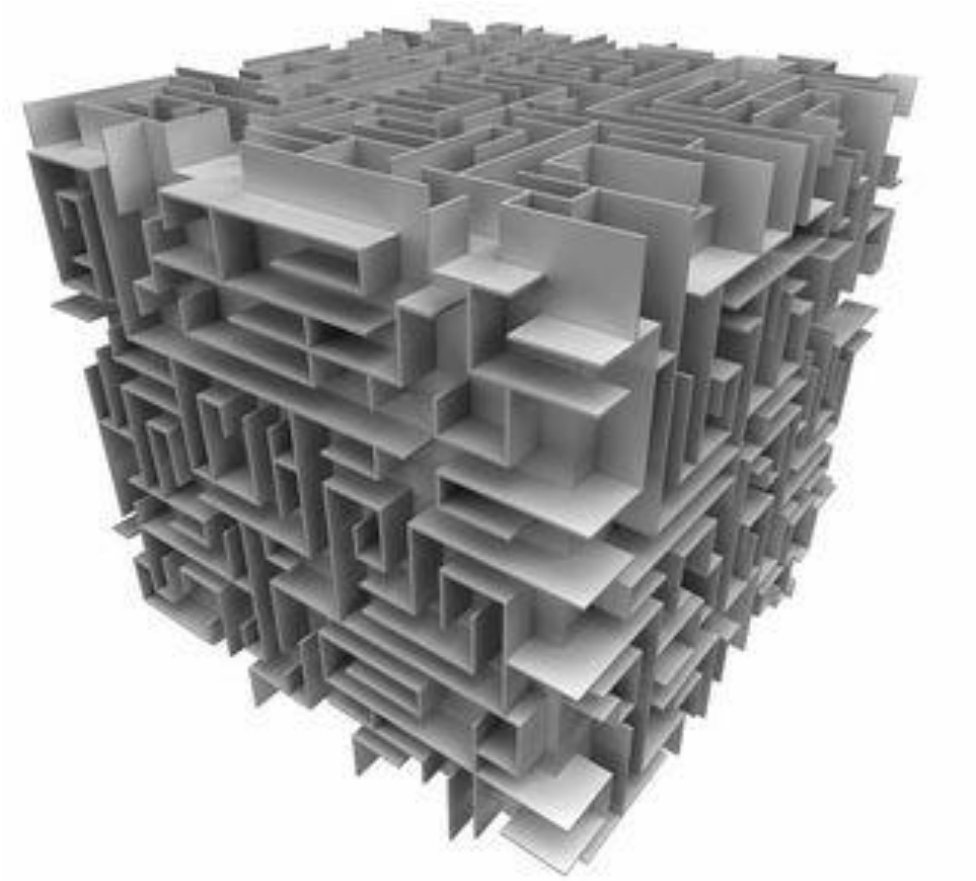
UNIVERSIDADE DE ÉVORA

Docentes:
Ligia Ferreira
João Leitão

Labirinto

Relatório do Trabalho Prático de EDA1



Trabalho elaborado por:
Ricardo Fusco, 29263
Luis Candeias, 29615

ENGENHARIA INFORMÁTICA – 2013/2014

Introdução

No âmbito da unidade curricular de Estruturas de Dados e Algoritmos I, incorporada no programa de Licenciatura em Engenharia Informática, segundo ano, semestre ímpar, foi-nos pedido que elaborássemos um trabalho prático como instrumento de avaliação.

O trabalho foi elaborado com base no enunciado fornecido pelo professor, onde explicitava as várias fases da sua concretização. O grupo é constituído por dois elementos: Luís Candeias e Ricardo Fusco, nºs 29615 e 29263 respetivamente.

Este projeto consiste na resolução de um labirinto, constituído por 0's e 1's, sendo que nos é dada as posições de entrada e saída do mesmo, correspondendo os 0's a posições por onde podemos passar para chegar ao fim do labirinto. A implementação deste projeto é elaborada através de estruturas de dados, dadas na disciplina, mais concretamente StackArrays e LinkedLists, sendo que o labirinto é fornecido através de um ficheiro de texto. Para resolução do labirinto foi nos indicado que o output deveria mostrar o labirinto a ser resolvido, em conjunto com uma sequência de movimentos que indicariam a saída, tais como, por exemplo, baixo-->esquerda-->direita. Foram criadas várias classes de maneira a auxiliar a resolução do labirinto.

DESENVOLVIMENTO

Classes usadas:

StackArray – Esta classe é uma implementação de stacks com recurso a um array. São definidos os construtores e todos os métodos relativos às stacks tais como push(E e) que coloca um novo elemento no topo do array, pop() que retira o elemento que esta no topo do array (isto é, o último a ser colocado), top() que retorna o elemento que está no topo (fim) da stack, isEmpty() que retorna um booleano se a stack estiver cheia ou não, size() que retorna um inteiro correspondente ao tamanho da stack, e toString() que imprime a stack. Esta estrutura de dados é do tipo LIFO (Last In First Out) pelo que o fim corresponde ao 'top' da stack e é esse mesmo elemento o primeiro a ser retirado da pilha quando é feito um pop().

LinkedList – Esta é uma implementação de listas ligadas iteráveis. São definidos os métodos relativos às linked lists como size() que devolve o tamanho da lista, isEmpty() que retorna um boleano de acordo com o estado da lista (cheia/vazia), o add(T x) que adiciona um objecto do tipo T na cauda da lista criando um novo nó e ligando o fim da lista a este nó tornando-o a nova cauda da lista, o remove(int index) que remove o objecto no indíce dado como parâmetro, o getNode que devolve o nó do objecto no índice dado como parâmetro, o findNext que encontra o próximo nó, set(índex i, T x) que altera o elemento no índice dado como parâmetro para o elemento também dado como parâmetro, o get(int ind) que nos devolve o elemento do nó correspondente ao índice dado como parâmetro e o toString que devolve uma representação na forma de String da lista. Para a implementação desta classe são necessárias as classes LinkedListIterator e SingleNode.

Campo - Esta classe serve para ter um controlo maior sobre as posições(campos) que já foram visitadas ou que são inválidas(cujo valor é 1), isto é, no caso do elemento nessa posição ser um '0' a posição será válida para o caminho, mas se for um '1' a posição será inválida, uma vez que o '1' corresponde a uma 'parede' do labirinto. Foi necessária a criação desta classe pois chegamos a um ponto em que, nos casos em que havendo 4 ou mais casas

juntas fazendo um quadrado, o programa começava um loop infinito continuando às voltas em tal situação. Os métodos: isValido() retorna um booleano dependendo se a posição é válida(true) ou não (false); setValido(boolean valido) que vai alterar a posição para válida ou inválida de acordo com o que queiramos; isJavisitado() retorna um booleano caso a posição já tenha sido visitada ou não; setJaVisitado(boolean visitado) vai alterar o valor da variável boleana jaVisitado para true ou false. Depois de passar numa posição válida e que ainda não tenha sido visitada, utilizamos este método para controlar se essa posição já foi visitada, para que não seja novamente utilizada; e o método getValor() que nos retorna o valor da posição, se é '0' ou '1'.

ListaLabirinto - Esta classe tem uma Linked List de Campos que vai incluir todos os Campos lidos a partir do labirinto que irá ser dado no ficheiro de texto e uma StackArray que vai conter os índices dos campos da lista do labirinto que pertencem ao caminho. Contem todos os métodos necessários para a resolução do labirinto, pois é uma das classes mais importantes, uma vez que é através dos seus métodos que grande parte do labirinto é resolvido. Entre os métodos os mais relevantes são o **addCampo()**, **setMoldura()**, **stacktoOutput()**, **encontraSaida()** e **calcValActual()**.

O método **addCampo()**, adiciona um novo Campo á LinkedList com valor 1 ou 0, este método serve essencialmente para a leitura do labirinto, para encher a lista com os campos consoante o ficheiro de texto.

Já o método **setMoldura()**, coloca uma 'moldura' à volta do labirinto, inserindo 1's no inicio e fim de cada linha e coluna, isto vai servir para garantir que o caminho percorrido não salta de uma linha para outra, uma vez que os elementos estão todos seguidos na LinkedList.

Quanto ao método **calcValActual()** devolve o valor do campo actual multiplicando a linha actual pelo número de colunas e somando a coluna actual, este método é importante para percorrer o labirinto, uma vez que através dele podemos calcular todos os possíveis campos para avançar no labirinto em diferentes direções.

É no método **stackToOutput()** que o output, ou seja, os passos para a resolução do labirinto são construídos, na qual a ordem do caminho que está na stack é invertida colocando os valores de uma stack para outra, pois é necessário começar a ver o caminho a partir da entrada comparando assim o elemento do topo da stack, com o elemento seguinte para saber qual a direção a mover de modo a construir a string do output até chegar ao fim do labirinto.

Por fim, o método mais importante, **encontraSaida()**, que devolve uma stack com os valores dos índices dos campos do labirinto que se encontram na LinkedList, desde a entrada do labirinto até ao fim. Para que tal suceda, os Campos vão ser verificados, primeiro para ver se são válidos, para diferenciar 0's e 1's, e é verificado se já foram visitados anteriormente. Neste algoritmo verifica-se primeiro o lado esquerdo, baixo, direita e cima, sendo que se for possível avançar da posição atual e a posição em qualquer uma das direcções (segundo a ordem de prioridades) for válida e ainda não tenha sido visitada, essa posição vai ser adicionada á stack e avança para lá. Caso cheguemos a uma posição que não tenha mais movimentos possiveis, essa posição será retirada da stack tornando esta posição inválida na lista do labirinto e recuaremos até à ultima posição com alguma direcção possível de percorrer, de modo a procurar outro caminho. Se o labirinto não tiver resolução, é devolvido um null, para que no main da classe Labirinto seja verificado se o labirinto é ou não impossível de resolver.

Labirinto

É aqui que se encontra o main onde o labirinto vai ser resolvido com recurso ás classes e respectivos métodos auxiliares anteriormente referidos. É pedido ao utilizador que digite o nome do ficheiro.txt que contem o labirinto que deseja ver resolvido, de acordo com as regras, isto é, uma matriz de 0's e 1's, e as coordenadas de entrada e saída do labirinto. Após a abertura do ficheiro esta classe irá ler todos os caracteres do ficheiro e guardar todos os 0's e 1's num objeto do tipo ListaLabirinto, usando LinkedLists. São guardados também os caracteres correspondentes às colunas e linhas de entrada e saída do labirinto, e é contado o número de colunas do labirinto, para um manuseamento mais fácil para encontrar

caminhos. De seguida são usados todos os métodos referentes às classes auxiliares para a resolução do labirinto e será dada a resposta ao mesmo.

As restantes classes são as classes relativas às excepções que poderão ocorrer ao longo da execução do código como OverflowException, NoSuchElementException, EmptyException, InvalidNodeException e IndexOutOfBoundsException.

Conclusão

Para concluir, achamos que conseguimos executar os objectivos do trabalho. Apesar de termos conseguido colocar o programa a resolver qualquer tipo de labirintos (até agora), podemos afirmar que não o conseguimos fazer da maneira mais ideal ou optimizada, mas como já tínhamos começado desta maneira achamos por bem continuarmos o que já tínhamos feito. O algoritmo para a resolução dos labirintos é relativamente simples, basta definir uma ordem de prioridades e ir verificando os campos nas direcções de acordo com as prioridades definidas. Claro está que, como já tínhamos referido, o caminho encontrado não vai ser o caminho mais curto/óptimo pelo facto de ter que satisfazer a ordem das prioridades. A ordem que nós definimos foi : 1º Esquerda, 2º Baixo, 3º Direita e 4º Cima, (contra o ponteiro dos relogios). Inicialmente tivemos alguns problemas, antes de criarmos a classe Campo, em sair de um loop infinito quando é encontrado um zero rodeado por zeros do tipo:

0 0 0 1
0 0 0 0
0 0 0 1

Isto era devido ao facto de não estarmos a controlar quais os campos que já tinham sido visitados, para tal criámos a classe Campo para, não só controlar os que têm 1's (inválidos) como também os que já foram visitados. Outro problema com o qual nos deparámos foi o facto de quando a verificação do caminho chegava ao primeiro zero de uma linha e por coincidência o valor do último campo na linha acima era também zero, era feito o push do índice do zero na linha acima para a stack quando verificada a esquerda, o que não deveria ser possível. Foi para resolver este problema que fizemos um método para colocar 1's à volta do labirinto, ou seja, cria uma linha com 1's acima do labirinto, outra linha de 1's abaixo do labirinto e coloca um 1 no inicio e fim de cada linha. Deste modo nunca irá saltar de um extremo de uma linha para outro.