# Fazendo um ratinho sair de um labirinto

Dado um array de 2 dimensões contendo os seguintes caracteres

\* '.' = espaço vazio

\* '#' = parede

\* 'S' = posição inicial (6,1)

\* 'E' = saída (0,30)

## Por exemplo:

0 1 2 3

0123456789012345678901234567890123456

0 #################################E###

1 ###....#.....#......###...#...#...#.#

2 #...##...###...####.....#..##.#.##..#

3 #.#########.###....##.####.##.#.#..##

4 #.............##.######..#......#...#

5 #####.#####.#..........#.##########.#

6 #S..........###.#.#.##..............#

7 #####################################

Observação: O array apresentado é apenas um exemplo o arquivo contendo o array deve ser lido do disco .

## Objetivos

Desenvolver uma aplicação que responda (colocadas aqui em ordem de dificuldade):

1. É possível chegar a uma saída? (Sim ou Não)

2. Quantos passos são necessários para chegar a saída seguindo o caminho mais curto?

3. Imprimir o caminho que foi seguido para chegar a saída

4. Criar os testes unitários para os objetos usados

5. Apresentar o labirinto e demais objetos de cena graficamente

## Alcançando a saída

O problema deve ser tradado como um grafo (graph) onde os elementos 'S', 'E' e '.' devem ser tratados como vértices (vertex). Cada vertice adjacente é conectado por uma aresta (edge). O problema deve ser resolvido usado busca em largura (breadth-first search ou BFS)

Dado que o grafo esta implicitamente informado pela matriz de caracteres armazenada no labirinto, precisamos apenas de uma estrutura adicional para controlar as visitas (color matrix):

int color[num\_rows][num\_columns];

\* white = um vértice não visitado (ainda não esta na fila)

\* gray = um vértice agurdando na fila (queue)

\* black = um vértice finalizado (já visitado - não esta mais na fila)

No início todos os vértices São brancos. Quando eles entram na fial de processamento se tornam cinza, e virar preto quando saem da fila (dequeue)

for row from 0 to num\_rows-1 {

for column from 0 to num\_columns-1 {

color[row][column] = white;

}

}

A fila de processamento deve ser inicializada com a entrada 'S' do labirinto:

q = new queue();

q.enqueue(start\_row,start\_column);

color[start\_row][start\_column] = gray;

Enquanto a fila não estiver vazia - realizamos a procura BFS.

while (q.head != q.tail) {

v = q.dequeue();

color[v.row][v.column] = black;

for each (r,c) adjacent to (row,column) {

if (maze[r][c]!='#' && color[r][c]==white) {

q.enqueue(r,c);

color[r][c] = gray;

}

}

}

Após o loop finalizado, todos nós que representam um caminho viável do início ao fim estão pretos os demais permanecem brancos.

switch (color[exit\_row][exit\_column]) {

case white:

print("Exit is unreachable");

break;

case black:

print("Exit is reachable");

break;

}

## Calculando as distancias

Usamos outra matriz para armazenar as distâncias:

int distance[num\_rows][cum\_columns];

\* Setamos o nó inicial com Zero - 0

\* todos outros nós são inicializados como "infinito"

\* quando um nó é retirado da fila a sua distância é igual ao do predecessor +1

for row from 0 to num\_rows-1 {

for column from 0 to num\_columns-1 {

color[row][column] = white;

distance[row][column] = MAXINT;

}

}

q = new queue();

q.enqueue(start\_row,start\_column);

color[start\_row][start\_column] = gray;

distance[start\_row][start\_column] = 0;

while (q.head != q.tail) {

v = q.dequeue();

color[v.row][v.column] = black;

for each (r,c) adjacent to (row,column) {

if (maze[r][c]!='#' && color[r][c]==white) {

q.enqueue(r,c);

color[r][c] = gray;

distance[r][c] = distance[row][column] + 1;

}

}

}

switch (color[exit\_row][exit\_column]) {

case white:

print("Exit is unreachable.");

break;

case black:

print("Exit is reachable in "+distance[exit\_row][exit\_column+" steps");

break;

}

## Calculando o menor caminho

Outro array deve ser usado para armazenar os predecessores de cada nó.

vertex pred[num\_rows][num\_columns];

O primeiro nó 'S'não tem predecessor. Para todos os outros nós, o predecessor é definido quando nó é retirado da fila.

for row from 0 to num\_rows-1 {

for column from 0 to num\_columns-1 {

color[row][column] = white;

distance[row][column] = MAXINT;

}

}

q = new queue();

q.enqueue(start\_row,start\_column);

color[start\_row][start\_column] = gray;

distance[start\_row][start\_column] = 0;

pred[start\_row][start\_column] = null;

while (q.head != q.tail) {

v = q.dequeue();

color[v.row][v.column] = black;

for each (r,c) adjacent to (row,column) {

if (maze[r][c]!='#' && color[r][c]==white) {

q.enqueue(r,c);

color[r][c] = gray;

distance[r][c] = distance[row][column] + 1;

pred[r][c] = vertex(row,column);

}

}

}

switch (color[exit\_row][exit\_column]) {

case white:

print("Exit is unreachable.");

break;

case black:

print("Exit is reachable in "+distance[exit\_row][exit\_column]+" steps: ");

print\_path\_to(exit\_row,exit\_column);

break;

}

No final do processamento da fila, o array de predecessores possui as informações necessário para reconstruir o caminho.

print\_path\_to (r,c) {

if (pred[r][c]!=null) {

print\_path\_to(pred[r][c].row,pred[r][c].column);

}

print("("+r+","+c+")");

}

Se as arestas tiverem tamanhos diferentes?

What if edges have different lengths?

\* Arestas tem tamanho (length) 1 no primeiro exercício 1.

\* Se as arestas tiverem tamanhos diferentes deve ser usado o algoritmo de Dijkstra's

\* BFS usa uma fila (queue). O algoritmo de Dijkstra's usa uma fila com prioridade (priority queue) com nós ordenados por distância.

for row from 0 to num\_rows-1 {

for column from 0 to num\_columns-1 {

color[row][column] = white;

distance[row][column] = MAXINT;

}

}

q = new PriorityQueue();

q.enqueue(start\_row,start\_column);

color[start\_row][start\_column] = gray;

distance[start\_row][start\_column] = 0;

pred[start\_row][start\_column] = null;

while (q.head != q.tail) {

v = q.dequeue();

color[v.row][v.column] = black;

for each (r,c) adjacent to (row,column) {

if (maze[r][c]!='#' && color[r][c]!=black) {

d = distance[row][column] + edge\_length[row][column][r][c];

if (color[r][c]==white) {

q.enqueue(r,c);

color[r][c] = gray;

distance[r][c] = d;

pred[r][c] = vertex(row,column);

}

if (color[r][c]==gray && d < distance[r][c]) {

distance[r][c] = d;

push up vertex (r,c) in priority queue;

pred[r][c] = vertex(row,column);

}

}

}

}

<http://aduni.org/courses/algorithms/courseware/handouts/Reciation_05.html>