BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RELATÓRIO DO TRABALHO PRÁTICO 1

INTELIGÊNCIA ARTIFICIAL

PROF. ALMIR OLIVETTE ARTERO

EYMAR FERRARIO DE LIMA

MATHEUS PRACHEDES BATISTA

PRESIDENTE PRUDENTE

2017

# 1 – Funcionamento das Heurísticas

Como Heurística pessoal resolvemos adotar a “Heurística em N Níveis” que é muito similar as propostas em aula, porem quem entra com a quantidade de níveis (N>2) que serão analisados é o jogador. Desta forma, para as três Heurísticas, o programa executa da seguinte maneira:

O algoritmo funciona como uma implementação da árvore A\* e usa 2 estruturas de dados auxiliares, sendo elas: uma tabela hash para o armazenamento dos estados já visitados pelo programa; E uma fila com prioridade para recuperar o próximo estado (consiste na soma do estado que tiver a menor heurística com a quantidade de movimentos realizados). A repetição do código consiste em pegar o primeiro elemento da fila de prioridade que não seja um nó visitado, adicionar os estados adjacentes a fila e repetir o processo até que o estado atual esteja resolvido ou a fila de prioridade esteja vazia.

Surgiu a necessidade de criar uma classe auxiliar que fica responsável por armazenar o estado do tabuleiro, assim como informações auxiliares como o número de movimentos para chegar no estado atual e a heurística desse estado. Essa classe também é utilizada para armazenar a tabela hash e a fila de prioridade citadas anteriormente.

# 2 – Número de Movimentos de cada Heurística

O teste foi realizado com um tabuleiro embaralhado cem vezes, e executado 1000 vezes para cada uma das Heurísticas, os resultados apresentados a seguir serão uma média das execuções. A Seguir podemos observar as médias:

* Heurística 1 (Análise em um nível): 18.37 Movimentos;
* Heurística 2 (Análise em dois níveis): 18.29 Movimentos;
* Heurística Pessoal (Para N = 3): 18.31 Movimentos;
* Heurística Pessoal (Para N = 4): 18.43 Movimentos;
* Heurística Pessoal (Para N = 5): 18.30 Movimentos;
* Resolvendo Aleatoriamente: 547545.2 Movimentos;

# 3 – Código Fonte

public class Puzzle {

/\*\*

\* Valor da peça no tabuleiro que representa o espaço vazio

\* Em um tablueiro 3x3 o vazio é representado por 8 (3x3-1)

\*/

public int vazio;

/\*\*

\* Estado do tabuleiro.

\* As peças começam pelo numero 0

\*/

private int[][] puzzle;

/\*\*

\* Posição no tabuleiro onde se encontra o espaço vazio

\*/

private int posBrancoI;

private int posBrancoJ;

/\*\*

\* Usado para embaralhar o tabuleiro

\*/

private Random r = new Random();

/\*\*

\* Constroi o tabuleiro com dimensão quadrada tamxtam

\* @param tam Tamanho dos lados do tabuleiro

\*/

public Puzzle(int tam) {

init(tam);

}

/\*\*

\* Inicializa o tabuleiro com as peças nos lugares correto e inicializa as

\* variaveis de posição do espaço vazio

\* @param tam Tamanho dos lados do tabuleiro

\*/

public void init(int tam) {

puzzle = new int[tam][tam];

for (int i = 0; i < tam; i++) {

for (int j = 0; j < tam; j++) {

puzzle[i][j] = i \* tam + j;

}

}

vazio = tam \* tam - 1;

posBrancoI = tam - 1;

posBrancoJ = posBrancoI;

}

/\*\*

\* Ordena o tabuleiro de maneira aleatoria e retorna uma fila que contem os

\* movimentos realizados. A fila é utilizada para fazer a animação na interface

\* grafica

\* @return Fila com os movimentos realizados

\*/

public Queue<Integer> rand() {

Queue<Integer> sequenciaMovimentos = new LinkedList<>();

while (!completo()) {

sequenciaMovimentos.add(rand(r));

}

return sequenciaMovimentos;

}

/\*\*

\* Embaralha o tabuleiro realizando numMov movimentos

\* @param numMov numero de movimentos utilizados para embaralhar

\*/

public void embaralhar(int numMov) {

for (int i = 0; i < numMov; i++) {

rand(r);

}

}

/\*\*

\* Verifica se a posição (i,j) é valida no tabuleiro

\* @param i linha

\* @param j coluna

\* @return

\*/

public boolean posValida(int i, int j) {

if (i < 0 || i >= puzzle.length) {

return false;

}

if (j < 0 || j >= puzzle.length) {

return false;

}

return true;

}

public int getPosBrancoI() {

return posBrancoI;

}

public void setPosBrancoI(int posBrancoI) {

this.posBrancoI = posBrancoI;

}

public int getPosBrancoJ() {

return posBrancoJ;

}

public void setPosBrancoJ(int posBrancoJ) {

this.posBrancoJ = posBrancoJ;

}

/\*\*

\* Altera a posição da peça vazia com a peça na posição (i,j)

\* @param i

\* @param j

\*/

public void swap(int i, int j) {

puzzle[posBrancoI][posBrancoJ] = puzzle[i][j];

puzzle[i][j] = vazio;

posBrancoI = i;

posBrancoJ = j;

}

/\*\*

\* Realiza um movimento aleatorio valido e retorna o movimento realizado

\* @param rand Objeto usado para calcular o numero aletorio

\* @return

\*/

public int rand(Random rand) {

int movimento = rand.nextInt(4);

int[] movimentosI = new int[]{-1, 1, 0, 0};

int[] movimentosJ = new int[]{0, 0, -1, 1};

if (posValida(posBrancoI + movimentosI[movimento], posBrancoJ + movimentosJ[movimento])) {

swap(posBrancoI + movimentosI[movimento], posBrancoJ + movimentosJ[movimento]);

return movimento;

} else {

return rand(rand);

}

}

/\*\*

\* Resolve o quebra-cabeça utilizando o algoritmo A\* adaptado para trabalhar

\* em niveis

\* @param nivel nivel de busca

\* @return Retona o HashNode que contem o estado resolvido, é possivel recuperar

\* o caminho percorrido deste esado usando o atributo pai

\*/

public HashNode heuristicaEmNiveis(int nivel){

int numIterações = 0;

HashNode estado = new HashNode(this.puzzle, dist1(), 0, null,numIterações);

int[] movimentosI = new int[]{-1, 1, 0, 0};

int[] movimentosJ = new int[]{0, 0, -1, 1};

int numMovimentos = -1;

//Lista de nos já visitados

HashMap<HashNode, Integer> visitados = new HashMap<>();

//Lista de nos para visitar

PriorityQueue<HashNode> listaParaVisitar = new PriorityQueue<>(14 \* 4);

listaParaVisitar.add(estado);

while (!listaParaVisitar.isEmpty()) {

numIterações++;

estado = listaParaVisitar.remove();

numMovimentos = estado.numMovimentos;

if (visitados.containsKey(estado)) {//Estado já visitado

continue;

}

visitados.put(estado, numMovimentos);

puzzle = estado.estado;

atualizaPosBranco();

if (completo()) {

break;//Estado final, termina o algoritmo

}

//Testa os 4 movimentos

for (int i = 0; i < 4; i++) {

//Se o movimento for valido, insere na lista para visitar

if (posValida(posBrancoI + movimentosI[i], posBrancoJ + movimentosJ[i])) {//Realiza o movimento

swap(posBrancoI + movimentosI[i], posBrancoJ + movimentosJ[i]);

int distancia = nivelHeuristica(nivel-1);

listaParaVisitar.add(new HashNode(clonePuzzle(), dist1() + numMovimentos + 1, numMovimentos + 1, estado,numIterações));

swap(posBrancoI - movimentosI[i], posBrancoJ - movimentosJ[i]);//Volta o movimento

}

}

}

if (completo()) {

return estado;

} else {

return null;

}

}

/\*\*

\* Realiza uma busca em nivel

\* @param altura nivel para realizar a busca

\* @return retona a menor heuristica encontrada no nivel buscado

\*/

public int nivelHeuristica(int altura) {

if (altura == 0) {//Caso a altura seja 0 retorna a heuristica no estado atual

return dist1();

}

//Altura diferente de 0, realiza os 4 movimentos possiveis e busca a menor

//heurisica dos estados adjacentes

int[] movimentosI = new int[]{-1, 1, 0, 0};

int[] movimentosJ = new int[]{0, 0, -1, 1};

int menorDistancia = Integer.MAX\_VALUE;

int distanciaAtual;

for (int j = 0; j < 4; j++) {

if (posValida(posBrancoI + movimentosI[j], posBrancoJ + movimentosJ[j])) {

swap(posBrancoI + movimentosI[j], posBrancoJ + movimentosJ[j]);//Realiza o movimento

if (altura == 1) {//Caso a altura seja 1, calcula a heuristica desse estado

distanciaAtual = dist1();

} else {//Caso contrario a heuristica desse estado é a heuristica do nivel abaixo

distanciaAtual = nivelHeuristica(altura - 1);

}

if (distanciaAtual < menorDistancia) {

menorDistancia = distanciaAtual;

}

swap(posBrancoI - movimentosI[j], posBrancoJ - movimentosJ[j]);//Volta o movimento

if (completo()) {

return 0;

}

}

}

return menorDistancia;

}

/\*\*

\* Calcula a distancia de Manhathan do estado atual

\* @return

\*/

public int dist1() {

int soma = 0;

int peça;

for (int i = 0; i < puzzle.length; i++) {

for (int j = 0; j < puzzle.length; j++) {

peça = puzzle[i][j];

int posJ = peça % puzzle.length;

int posI = peça / puzzle.length;

soma += Math.abs(i - posI) + Math.abs(j - posJ);

}

}

return soma;

}

/\*\*

\* Verifica se o estado atual está resolvido

\* @return

\*/

public boolean completo() {

for (int i = 0; i < puzzle.length; i++) {

for (int j = 0; j < puzzle.length; j++) {

if (puzzle[i][j] != i \* puzzle.length + j) {

return false;

}

}

}

return true;

}

/\*\*

\* Clona o estado do tabuleiro

\* @return

\*/

public int[][] clonePuzzle() {

int[][] ret = new int[puzzle.length][puzzle.length];

for (int i = 0; i < puzzle.length; i++) {

for (int j = 0; j < puzzle.length; j++) {

ret[i][j] = puzzle[i][j];

}

}

return ret;

}

/\*\*

\* Retorna a representação do tabuleiro em String

\* @return

\*/

public String toString() {

String s = "";

for (int i = 0; i < puzzle.length; i++) {

for (int j = 0; j < puzzle.length; j++) {

if (puzzle[i][j] == vazio) {

s += " ";

} else {

s += puzzle[i][j] + " ";

}

}

s += "\n";

}

return s;

}

public void setPuzzle(int[][] puzzle) {

this.puzzle = puzzle;

}

public int[][] getPuzzle() {

return puzzle;

}

/\*\*

\* Procura a peça vazia e atualiza as variaveis de posição

\*/

public void atualizaPosBranco() {

for (int i = 0; i < puzzle.length; i++) {

for (int j = 0; j < puzzle.length; j++) {

if (puzzle[i][j] == vazio) {

posBrancoI = i;

posBrancoJ = j;

return;

}

}

}

}

public static void main(String[] args) {

Puzzle p = new Puzzle(3);

long cont = 0;

for (int i = 0; i < 1000; i++) {

p.embaralhar(100);

cont += p.rand().size();

}

System.out.println(cont / 1000f);

}

/\*\*

\* Classe usada para armazenar os estados do tabuleiro, assim como informações

\* auxiliares (numero de movimentos até chegar no estado atual, numero de

\* iterações no algoritmo até chegar no estado atual, etc..)

\*/

public class HashNode implements Comparable<HashNode> {

/\*\*

\* Estado do tabuleiro

\*/

private int[][] estado;

/\*\*

\* Usado para ordenar a fila de prioridade, representa a soma da distancia

\* percorrida até chegar neste estado com a heuristica deste estado

\*/

private int distancia;

/\*\*

\* Numero de movimentos utilizados para chegar neste estado

\*/

private int numMovimentos;

/\*\*

\* Numero de iterações utilizados para chegar neste estado

\*/

private int numIterações;

/\*\*

\* Estado anterior a este estado. Utilizado para recuperar o caminho

\* percorrido para chegar neste estado.

\*/

private HashNode pai;

public HashNode(int[][] estado, int distancia, int numMovimentos, HashNode pai,int numIterações) {

this.estado = new int[estado.length][estado.length];

for (int i = 0; i < estado.length; i++) {

for (int j = 0; j < estado.length; j++) {

this.estado[i][j] = estado[i][j];

}

}

this.distancia = distancia;

this.numMovimentos = numMovimentos;

this.pai = pai;

this.numIterações = numIterações;

}

@Override

public int compareTo(HashNode o) {

int diferença = Integer.compare(distancia, o.distancia);

if (diferença == 0) {

diferença = Integer.compare(numMovimentos, numMovimentos);

}

if (diferença != 0) {

return diferença;

}

if (equals(o)) {

return 0;

}

return -1;

}

@Override

public boolean equals(Object obj) {

if (obj instanceof HashNode) {

HashNode o = (HashNode) obj;

for (int i = 0; i < estado.length; i++) {

for (int j = 0; j < estado.length; j++) {

if (estado[i][j] != o.estado[i][j]) {

return false;

}

}

}

return true;

} else {

return false;

}

}

@Override

public int hashCode() {

return Arrays.deepHashCode(estado);

}

public int[][] getEstado() {

return estado;

}

public void setEstado(int[][] estado) {

this.estado = estado;

}

public int getDistancia() {

return distancia;

}

public void setDistancia(int distancia) {

this.distancia = distancia;

}

public int getNumMovimentos() {

return numMovimentos;

}

public void setNumMovimentos(int numMovimentos) {

this.numMovimentos = numMovimentos;

}

public HashNode getPai() {

return pai;

}

public void setPai(HashNode pai) {

this.pai = pai;

}

public int getIterações(){

return numIterações;

}

}

}