UNIVERSIDADE ESTADUAL PAULISTA
FACULDADE DE CIÊNCIAS E TECNOLOGIA

8 PUZZLE
HEURÍSTICAS DE SOLUÇÃO
RELATÓRIO

INTELIGÊNCIA ARTIFICIAL
PROF. DR. ALMIR ARTERO

BRUNO SANTOS DE LIMA
LEANDRO UNGARI CAYRES

PRESIDENTE PRUDENTE
JUNHO - 2017

BRUNO SANTOS DE LIMA
LEANDRO UNGARI CAYRES

8 PUZZLE
HEURÍSTICAS DE SOLUÇÃO
RELATÓRIO

Relatório do trabalho prático referente a solução do problema 8 Puzzle através da utilização de heurísticas, realizado na disciplina de Inteligência Artificial, lecionada pelo docente Dr. Almir Olivette Artero, no curso Bacharelado em Ciência da Computação – Departamento de Matemática e Computação da Faculdade de Ciências e Tecnologia (FCT Unesp – Presidente Prudente).

PRESIDENTE PRUDENTE
JUNHO – 2017

SUMÁRIO

1 INTRODUÇÃO

Neste trabalho da disciplina de Inteligência Artificial, o objetivo consiste na implementação na implementação de diferentes heurísticas para a solução do problema proposto no jogo *8 Puzzle*, assim como, de forma específica, prover uma interface gráfica que viabilize a entrada de dados por parte do usuário e a exibição passo a passo da resolução do problema seguindo a abordagem selecionada pelo usuário.

A ferramenta foi desenvolvida utilizando a linguagem de programação Java através do ambiente de desenvolvimento integrado NetBeans. Para a execução deste trabalho somente é requerido um computador com a máquina virtual Java instalada, e em correto funcionamento.

Este relatório foi dividido nas seguintes seções: na Seção 2, são apresentados os algoritmos utilizados para a solução do referido problema, explicando para cada sua abordagem e funcionamento. Por fim, na Seção 3, é apresentado um modelo comparativo de desempenho para a solução através de entradas particulares para todos os algoritmos.

2 MÉTODOS DE RESOLUÇÃO E HEURÍSTICAS

Neste trabalho são apresentadas quatro heurísticas para a resolução do problema, sendo que três foram fixadas, enquanto a quarta foi definida como um proposta livre. Para a resolução, somente foram permitidas movimentações de peças nas quatro direções (cima, baixo, direita e esquerda), caso disponíveis, de forma a ocupar o espaço em branco, movimento a movimento.

2.1. Algoritmo Aleatório

Esta abordagem objetiva escolher o próximo movimento dentre os possíveis de forma aleatória, até que um tabuleiro resultante alcance o tabuleiro com as peças nas posições corretas.

```
  public void aleatorio() {

      int posicao;
      this.contadorSolucao = 0;
      this.listaResultado = new ArrayList<>();
      listaResultado.add(tabuleiroInicial);
      tabuleiroInicial = tabuleiroInicial.clone();

      while (!this.estaResolvido(this.tabuleiroInicial)) {

int[] lista = this.possiveisMovimentos(this.tabuleiroInicial.getPosVazia());
        posicao = lista[(int) (Math.random() * (lista.length))];
        this.troca(posicao);
        listaResultado.add(tabuleiroInicial);
        tabuleiroInicial = tabuleiroInicial.clone();
        this.contadorSolucao++;
      }
  }
```

2.2. Algoritmo Heurística de Peso em um Nível

Este método objetiva avançar através dos passos com tabuleiros com o menor peso dentre os tabuleiros possíveis. Em cada iteração, os tabuleiros são adicionados a uma fila de prioridades ordenados pelo peso do tabuleiro, desta forma, o primeiro elemento da fila é escolhido na iteração.

```
public void heuristicaEmUmNivel() {
         //Inicializando
        listaResultado = new ArrayList<>();
        listaDeVizitados = new HashMap<>();
        lista = new PriorityQueue<>();
```

```
            this.tabuleiroInicial.setNumMovimento(0);
            lista.add(this.tabuleiroInicial);
             //Lista contém apenas o inicio do jogo.

            this.tabuleiroInicial = distanciaUmNivel();
            //Tabuleiro recebe o resultado final

            //Coloca o caminho certo na lista
            Tabuleiro t = this.tabuleiroInicial;
            while (t != null) {
                listaResultado.add(0, t);
                t = t.getAntecessor();
            }

    }

    private Tabuleiro distanciaUmNivel() {
            Tabuleiro t = null;
            int numMovimento = 0;
            this.contadorSolucao = 0;

            while (!lista.isEmpty()) {

                t = lista.poll();
                listaDeVizitados.put(t, numMovimento);
                if (estaResolvido(t)) return t;
                numMovimento++;
                //Selecionando o conjunto de movimentos possiveis
                int[] possiveis = possiveisMovimentos(t.getPosVazia());

                for (Integer e : possiveis) {

                    Tabuleiro novo = t.clone();
                    novo.setNumMovimento(numMovimento);
                    novo.setAntecessor(t);
                    //Aplica o novo movimento no tabuleiro novo
                    troca(novo, e);

                    //Caso o novo tabuleiro já esteja na lista de vizitados
           ele é ignorado e é gerado um novo
                    if (listaDeVizitados.containsKey(novo)) continue;
                    lista.add(novo);
                 //Adicionando novo tabuleiro na lista de prioridade por
           peso
                }
                this.contadorSolucao++;
            }
            return t;
        }
```

2.3. Algoritmo Heurística de Peso em dois Níveis

Este algoritmo consiste em uma modificação da abordagem anterior, no qual o cálculo do peso é verificado em dois níveis, ou seja, a decisão de qual tabuleiro deve ser o próximo leva em consideração o peso deste juntamente com o tabuleiro ‘filho’ de menor peso, dentre as opções, a opção de menor peso é escolhida.

```
    public void heuristicaEmDoisNiveis(){

            //Inicializando
            listaResultado = new ArrayList<>();
```

```
                listaDeVizitados = new HashMap<>();
                lista = new PriorityQueue<>();

                this.tabuleiroInicial.setNumMovimento(0);
                lista.add(this.tabuleiroInicial);
                //Lista contém apenas o inicio do jogo.

                this.tabuleiroInicial = distanciaDoisNiveis();
                //Tabuleiro recebe o resultado final

                //Coloca o caminho certo na lista
                Tabuleiro t = this.tabuleiroInicial;
                while (t != null) {
                    listaResultado.add(0, t);
                    t = t.getAntecessor();
                }

        }

        public Tabuleiro distanciaDoisNiveis(){

                Tabuleiro t = null;
                int numMovimento = 0;
                this.contadorSolucao = 0;
                PriorityQueue<Tabuleiro> listaNivel2 = new PriorityQueue<>();

                while (!lista.isEmpty()) {

                   t = lista.poll();
                   listaDeVizitados.put(t, numMovimento);
                   if(estaResolvido(t)) return(t);
                   numMovimento++;
                   //Selecionando o conjunto de movimentos possíveis
                   int[] movimentosNivel1 = possiveisMovimentos(t.getPosVazia());

                    for (Integer e : movimentosNivel1) {

                        Tabuleiro novo = t.clone();
                        novo.setNumMovimento(numMovimento);
                        novo.setAntecessor(t);
                        troca(novo, e); //Aplica o novo movimento no tabuleiro
novo

                            //Caso o novo tabuleiro não esteja na lista de
                     visitados realiza processamento
                        if (!listaDeVizitados.containsKey(novo)) {

                            //Lista de movimentos para gerar o nível 2
                            int[]               movimentosNivel2               =
                     possiveisMovimentos( novo.getPosVazia());
                            for(Integer b : movimentosNivel2){

                                //Gerando o tabuleiro de nível 2
                                Tabuleiro novo2 = novo.clone();
                                troca(novo2, b);

        //Adicionando todos os tabuleiros de nível 2 em uma fila com os filhos
nível 2
                                if(!listaDeVizitados.containsKey(novo2)){
                                    listaNivel2.add(novo2);
                                }
                            }
                            if(!listaNivel2.isEmpty()){

                                //Retorna o nível 2 de menor peso
                                Tabuleiro tAux = listaNivel2.poll();
                            //Variáveis auxiliares para cálculo de peso da
                     heurística 2
```

```
                        novo.nivel2 = tAux;
                        novo.alvoNivel2 = true;
                    }
                    lista.add(novo);
                //Adicionando novo tabuleiro na lista de prioridade por
        peso
                    novo.nivel2 = null;
                    novo.alvoNivel2 = false;
                }

            }

            this.contadorSolucao++;
        }
        return(t);
}
```

2.4. Algoritmo Pessoal

Por fim, nesta abordagem, foi decidido pela implementação de um algoritmo que faça o cálculo de peso considerando apenas um nível, assim como na Heurística de Peso em um Nível, porém em uma busca bidirecional, partindo tanto do tabuleiro inicial quanto da solução do exercício, até que ambas trajetórias encontrem um estado comum e idêntico, desta forma, o caminho completo desde o estado inicial até o final é obtido.

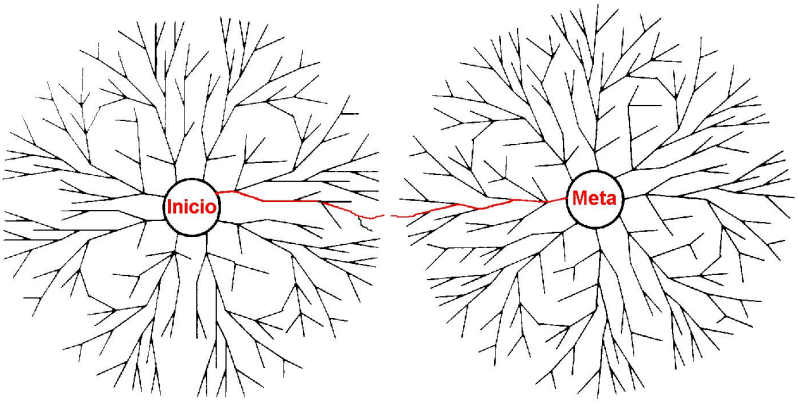

Figura 1 - Representação abstrata de uma busca bidirecional.

```
public void heuristicaPessoal() {

        listaOrigem = new PriorityQueue<>();
        listaDestino = new PriorityQueue<>();
        listaDeVizitadosOrigem = new HashMap<>();
        listaDeVizitadosDestino = new HashMap<>();
        listaResultado = new ArrayList<>();

        tabuleiroInicial.alvoModificado = true;

        Tabuleiro resultado = new Tabuleiro();

        listaOrigem.add(this.tabuleiroInicial);
        listaDestino.add(resultado);
```

```
        tabuleiroInicial.alvoModificado = true;
        resultado.alvoModificado = true;
        Tabuleiro resposta = distanciaAEstrelaBidirecional();
        tabuleiroInicial.alvoModificado = false;
        resultado.alvoModificado = false;

        Tabuleiro t = resposta;
        if (matchFirst) {

            while (t != null) {

                listaResultado.add(0, t);
                t = t.getAntecessor();
            }

            t = resposta.getAlvo();
            while (t.getAntecessor() != null) {
                listaResultado.add(t.getAntecessor());
                t = t.getAntecessor();
            }
        }
        else{

            while (t != null) {

                listaResultado.add(t);
                t = t.getAntecessor();
            }

            t = resposta.getAlvo();
            while (t.getAntecessor() != null) {
                listaResultado.add(0,t.getAntecessor());
                t = t.getAntecessor();
            }
        }


        tabuleiroInicial = listaResultado.get(listaResultado.size()-1);
        tabuleiroInicial.alvoModificado = false;
        tabuleiroInicial.numMovimento = resposta.numMovimento;
    }

private Tabuleiro distanciaAEstrelaBidirecional() {

        Tabuleiro tOrigem;
        Tabuleiro tDestino;
        int[] possiveis;
        int numOrigem = 0, numDestino = 0;
        int count = -1;

        tOrigem = listaOrigem.poll();
        listaDeVizitadosOrigem.put(tOrigem, numOrigem);

        tDestino = listaDestino.poll();
        listaDeVizitadosDestino.put(tDestino, numDestino);

        tOrigem.setAlvo(tDestino);
        tDestino.setAlvo(tOrigem);

        this.contadorSolucao = 0;

        while (tOrigem != null && tDestino != null) {

            if (estaResolvido(tOrigem)) {

                matchFirst = true;
                return tOrigem;
```

```
            } else if (estaResolvido(tDestino)) {

                matchFirst = false;
                return tDestino;
            }

            count++;
            if (count % 2 == 0) {
                possiveis = possiveisMovimentos(tDestino.posVazia);
                numDestino++;
                for (Integer e : possiveis) {

                    Tabuleiro novo = tDestino.clone();
                    novo.setNumMovimento(numDestino);
                    novo.setAntecessor(tDestino);
                    troca(novo, e);
                    if (listaDeVizitadosDestino.containsKey(novo)) {
                        continue;
                    }
                    listaDestino.add(novo);
                }

                tDestino = listaDestino.poll();
                listaDeVizitadosDestino.put(tDestino, numDestino);

            } else {
                possiveis = possiveisMovimentos(tOrigem.posVazia);
                numOrigem++;
                for (Integer e : possiveis) {

                    Tabuleiro novo = tOrigem.clone();
                    novo.setNumMovimento(numOrigem);
                    novo.setAntecessor(tOrigem);
                    troca(novo, e);
                    if (listaDeVizitadosOrigem.containsKey(novo)) {
                        continue;
                    }
                    listaOrigem.add(novo);
                }

                tOrigem = listaOrigem.poll();
                listaDeVizitadosOrigem.put(tOrigem, numOrigem);

            }
            this.contadorSolucao++;
        }
        return tOrigem;
    }
```

2.5. Codificação Auxiliares

A implementação foi organizada em duas classes, sendo estas a classe Jogo e Tabuleiro. A classe Tabuleiro é a que contém as implementações próprias do Tabuleiro utilizado pela classe Jogo, esta última por sua vez é a responsável por resolver o Tabuleiro através de uma das três heurísticas ou busca aleatória e armazenar o conjunto resultado final para que possa ser visualizado na interface gráfica. Os algoritmos de resolução são métodos da classe Jogo, além desses algoritmos existem outros métodos

da classe jogo que são utilizados e importantes para que os algoritmos de resolução funcionem corretamente, tais métodos auxiliares são dispostos a seguir:

- *calcularPeso()*: este método é utilizado para calcular o peso do tabuleiro nos algoritmos de heurísticas utilizados.

```
  protected static int calcularPeso(Tabuleiro tabuleiro) {
        int soma = 0;
        int size = tabuleiro.getSize();

        if(tabuleiro.alvoNivel2){

            int somaTabNivel1 = 0;
            int somaTabNivel2 = 0;

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

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

                        if (tabuleiro.matriz[i][j] == -1) continue;
                        somaTabNivel1 += Math.pow((i*size+j+1) -
tabuleiro.matriz[i][j], 2);
                    }
            }

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

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

                        if (tabuleiro.nivel2.matriz[i][j] == -1) continue;
                        somaTabNivel2 +=
                       Math.pow((i*size+j+1) - tabuleiro.nivel2.matriz[i][j],
                    2);
                    }
            }
            soma = somaTabNivel1 + somaTabNivel2;

        }
        else{

            if (tabuleiro.alvoModificado) {
                for (int i = 0; i < size; i++) {

                    for (int j = 0; j < size; j++) {
                        soma +=
                      Math.pow(tabuleiro.getAlvo().matriz[i][j] -
                    tabuleiro.matriz[i][j],2);
                    }
                }
            } else {

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

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

                        if (tabuleiro.matriz[i][j] == -1) continue;
                        soma += Math.pow((i*size+j+1) - tabuleiro.matriz[i][j],
2);
                    }
                }

            }
```

```
        }

        Return(soma);
    }
```

- *estaResolvido()*: os métodos informam se o tabuleiro está resolvido, isto só ocorre caso o peso do tabuleiro é igual a zero.

```
    public boolean estaResolvido() {

        return (Jogo.estaResolvido(tabuleiroInicial));
    }

    private static boolean estaResolvido(Tabuleiro t) {

        return (Jogo.calcularPeso(t) == 0);
    }
```

- *embaralhar()*: é utilizado para realizar o embaralhamento automático, passando como parâmetro um inteiro que significa o número de movimentos que será feito para desorganizar o tabuleiro inicial.

```
    public void embaralhar(int numPassos) {
        while (numPassos > 0) {

              int[] possibilidades =
       this.possiveisMovimentos(tabuleiroInicial.getPosVazia());
              int casa =
              possibilidades[(int)(Math.random()*possibilidades.length)];
              this.troca(casa);
              numPassos--;
        }
    }
```

Existem uma função *troca()* que basicamente realiza um movimento no tabuleiro atual, ou seja, dado o tabuleiro atual com uma posição vazia pode-se fazer alguns movimentos trocando sua posição vazia por outra que é adjacente a ela, assim realizando um movimento no tabuleiro.

- Por fim, existe um método *possiveisMovimentos()* que retorna as possíveis trocas de peças, ou seja, as posições adjacentes da posição vazia, observe o código abaixo:

```
private int[] possiveisMovimentos(int posicao) {

        switch (posicao) {
            case 0: return new int[]{1, 3};
            case 1: return new int[]{0, 2, 4};
            case 2: return new int[]{1, 5};
            case 3: return new int[]{0, 4, 6};
            case 4: return new int[]{1, 3, 5, 7};
            case 5: return new int[]{2, 4, 8};
```

```
            case 6: return new int[]{3, 7};
            case 7: return new int[]{4, 6, 8};
            default:return new int[]{5, 7};
        }
    }
```

Para este método, é considerado que as posições da matriz que representa o tabuleiro não são posições com índices de linha e coluna, mas sim posições virtuais que vão de 0 até 8, observe ao tabuleiro abaixo, assim a posição vazia for a 2, só podem fazer dois movimentos o que está na posição 1 ir para a 2 ou o que está na posição 5 ir para a 2, código a posição vazia é preenchida com o valor -1.

Para maiores detalhes o projeto construído no NetBeans encontra-se em anexo.

3. ANÁLISE DE DESEMPENHO

Nesta seção será discutido uma breve análise de desempenho entre os métodos de resolução sendo eles a busca aleatória e as três heurísticas citadas anteriormente. Para o processo de análise serão considerados quatro tabuleiros, ilustrados na Figura 2, cada tabuleiro foi rotulado com um número de identificação sendo de 1 à 4.

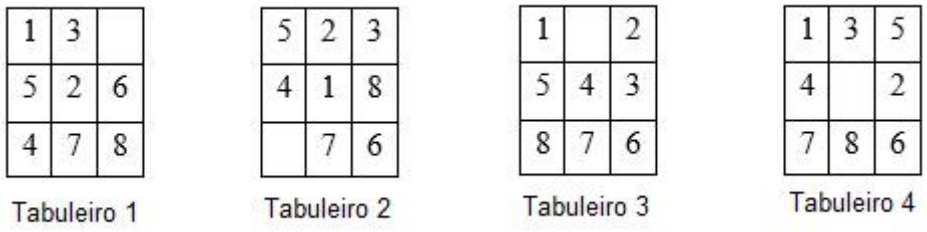

Figura 2 - Tabuleiros utilizados no processo de análise.

Foi analisado o número de movimentos tentados para encontrar a solução do jogo, além disso três algoritmos trazem um conjunto de solução melhor, ou seja, por serem implementados formando uma estrutura de árvore é possível saber uma melhor solução com um número menor de movimentos.

A primeira análise feita foi com base no número total de tentativas de cada algoritmo para encontrar a solução do tabuleiro, tal análise é ilustrada na Figura 3.

Tabuleiro	Busca Aleatória	Heurística um nível	Heurística dois níveis	Heuristica Pessoal
1	2324686	6	6	2388
2	804588	12066	47707	74954
3	1401865	38557	758	36284
4	1313220	135782	30294	125579

Figura 3 - Tentativas de movimentos para encontrar a solução.

É notório que o algoritmo de Busca Aleatória apresenta um conjunto maior e bem superior de tentativas e movimentos até encontrar a solução isto ocorre para os quatro tabuleiros testados, vale ressaltar que para o método aleatório os valores de números de tentativas podem variar, mesmo em um tabuleiro especifico, isto ocorre

devido a sua natureza de encontrar caminhos aleatoriamente, entretanto os resultados de número de tentativas geralmente são maiores que os demais algoritmos.

Em geral, as três heurísticas se sobressaem com relação ao aleatório, entretanto entre eles a Heurística em dois níveis tem um melhor desempenho nos casos testados, obtendo respostas com um número menor de tentativas, a Heurística Pessoal gera sempre um número grande de tentativas, inclusive para tabuleiros com uma baixa complexidade de resolução, isto deve-se muito a sua característica de resolução bidirecional, como citado anteriormente.

A Heurística em um nível obteve melhores resultados nos tabuleiros 1 e 2 entretanto no tabuleiro 2 e 3 apresenta um desempenho pior que a Heurística em dois níveis e levemente inferior que a Heurística pessoal.

Tabuleiro	Busca Aleatória	Heurística um nível	Heurística dois níveis	Heuristica Pessoal
1	2324686	6	6	9944
2	804588	6002	23438	18430
3	1401865	18999	379	8925
4	1313220	63960	61646	30732

Figura 4 - Análise de melhor solução.

Os algoritmos que utilizam heurísticas baseiam-se em árvores de melhor caminho para chegar ao conjunto solução, esses melhores caminhos são o caminho completo com somente tentativas que obtiveram sucesso, tais que essas tentativas formam um caminho de solução. A Figura 4 ilustra esta análise.

Os três algoritmos de heurística apresentam o melhor caminho com as tentativas de sucesso de resolução, sendo esses caminhos superiores ao total de tentativas gerais informados na analise anterior, é observado que a Heurística em dois níveis ainda apresenta um melhor desempenho com um número de tentativas menores, sendo apenas maior no tabuleiro 2, com relação a Heurística pessoal.