Resolução do *N-Puzzle* utilizando Métodos de Pesquisa em Linguagem JAVA (Tema 5 / Grupo 53)

Manuel Monteiro (201504445)
MIEIC
FEUP
Porto, Portugal
up201504445@fe.up.pt

Diogo Moreira (201504359)
MIEIC
FEUP
Porto, Portugal
up201504359@fe.up.pt

Ao longo das últimas semanas, o grupo tem estado a desenvolver um programa que consiga a melhor solução possível para alguns níveis do N-Puzzle em IART.

Por causa de algumas limitações, como por exemplo, o tempo curto definido para desenvolver este projeto, foi difícil obter uma solução ótima, no entanto, pensamos ter conseguido um bom resultado.

I. Introdução

No âmbito da unidade curricular Inteligência Artificial do Mestrado Integrado em Engenharia Informática e Computação, foi-nos sugerida a elaboração de um projeto que conseguisse encontrar soluções para certos problemas. Esse problema foi selecionado pelo grupo dentro de um leque de várias opções disponibilizadas pelos docentes.
A escolha do N-Puzzle baseou-se no estilo que o prescrevia. A jogabilidade simples e a combinação de jogadas possíveis tornaram-no numa boa opção e a motivação do grupo para a sua resolução foi forte.
O objetivo deste trabalho é a aplicação dos primeiros conceitos interiorizados nas aulas teóricas e desenvolvidos nas aulas práticas da unidade curricular.

II. Descrição do Problema

O jogo *N-Puzzle* é um jogo do estilo *puzzle*, que é jogado já há alguns anos em todas a plataformas mais usadas como android, iOS, até PC e em formato analógico através de uma placas de plástico.

Fig. 1. Imagem do jogo

Regras

Este jogo é constituído pelas seguintes regras:

a) Movimentos possíveis: só existem quatro movimentos possíveis - norte, sul, este, oeste.

Objetivo

O objetivo do jogo é deixar a casa do canto inferior direito livre e todas as casas anteriores

estarem de modo crescente da esquerda para a direita.

Fig. 3. Exemplo de utilizador a ganhar

III. Formulação do Problema

O estado de jogo tem de ser caracterizado por uma matriz com o tabuleiro em que a largura e comprimento são determinados pela maior distância entre os quadrados nos extremos, tanto na horizontal como vertical. Cada posição da matriz pode assumir os seguintes valores:

- 0 - Célula vazia;
- {1...X} - número de uma célula em que X é (comprimento x largura - 1)

Estado inicial

O estado inicial começa com tabuleiro em que todas as células são aparentemente aleatórias e uma célula a {0}, célula inicial a partir do qual se vai começar o algoritmo.

Testes

- Minimizar número de movimentos;
- Todas as células indo de modo crescente da esquerda para a direita, sendo que a última célula tem de ser vazia {0}.

Operadores

Mover o célula para norte, este, oeste ou sul:

Tabela 1
Operadores

Operador	Pré-condição	Efeito	Custo
Norte	Tabuleiro[x - 1][y] != 0	Tabuleiro[x][y] = Tabuleiro[x-1][y] Tabuleiro[x-1][y] = 0	1
Sul	Tabuleiro[x + 1][y] != 0	Tabuleiro[x][y] = Tabuleiro[x+1][y] Tabuleiro[x+1][y] = 0	1
Este	Tabuleiro[x][y + 1] != 0	Tabuleiro[x][y] = Tabuleiro[x][y+1] Tabuleiro[x][y+1] = 0	1
Oeste	Tabuleiro[x][y - 1] != 0	Tabuleiro[x][y] = Tabuleiro[x][y-1] Tabuleiro[x][y-1] = 0	1

V. Implementação do Jogo

A nossa abordagem para encontrar resoluções para o *N-Puzzle* passou por enquadrar o mapa do jogo, a célula vazia (representada por 0) e espaços disponíveis (representada por {1..X}), numa matriz (Board.java). De seguida, corremos os algoritmos de pesquisa que se ajustam ao nosso problema (AStar.java e BFS.java).

A célula vazia pode então só ser movida em quatro direções: norte, este, sul, oeste.

Todos os movimentos possíveis para uma dada posição da célula vazia são calculados numa classe

à parte (utils.java). Esta classe contém uma função que devolve os nós adjacentes, se for possível a célula vazia deslocar-se nessa direção.

Utilizando a informação fornecida pela classe utils.java decidimos então se devemos adicionar o node à nossa stack ou priorityQueue, dependendo do algoritmo utilizado.

Para sabermos se o algoritmo teve sucesso em encontrar o caminho até à posição final só precisamos de uma condição, cuja função é apenas a de verificar se a célula vazia está na posição do canto inferior direito e o todos os nodes para trás são sequenciais.

Como cada Node contém uma variável denominada parent do tipo Node, podemos reconstruir o caminho percorrido pelo bloco de forma recursiva.

VI. Algoritmos de Pesquisa

Os algoritmos de pesquisa utilizados foram o A*b e o BFS (Breadth First Search).
O algoritmo do Custo-Uniforme demonstrou ter uma implementação demasiado semelhante ao A* neste problema, por isso não foi implementado.

Para o algoritmo BFS, percorremos todos os Nodes num nível e só depois avançamos para o seguinte. Esta implementação é vantajosa pois assim que encontrar uma solução sabemos que é, sem dúvida, uma solução ótima já que estará no menor nível possível (= menor número de moves possível). Expandimos todos os Nodes, excepto se o Node seguinte atual existir na lista de Nodes vistos ou se o Node seguinte for negado pela classe Utils.java, o que significa que não é um movimento permitido.

Para o algoritmo A*, ao contrário dos outros algoritmos, usamos uma priorityQueue onde os Nodes são organizados por menor custo. O custo é calculado pela soma da distância do Node à posição final (Manhattan) mais o custo de movimento total até essa posição, em que o custo de movimento é maior se esta movimentação retirar uma célula da sua posição final.
Assim que a célula vazia está na posição do canto inferior direito e o todos os nodes para trás são sequenciais temos a solução ótima.

VII. Experiências e Resultados

Tabela 2
Resultados do Nível 1

Nível 1	**Nodes** (uni)	**Tempo** (ms)
BFS	51	3
A*	44	1

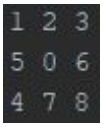

Fig. 4. Nível 1

Tabela 3
Resultados do Nível 2

Nível 2	**Nodes** (uni)	**Tempo** (ms)
BFS	209	5
A*	142	2

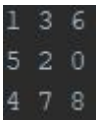

Fig. 5. Nível 2

Tabela 4
Resultados do Nível 3

Nível 3	**Nodes** (uni)	**Tempo** (ms)
BFS	1205	11
A*	596	5

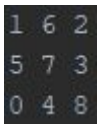

Fig. 6. Nível 3

Tabela 5
Resultados do Nível 4

Nível 4	**Nodes** (uni)	**Tempo** (ms)
BFS	8465	109
A*	3193	23

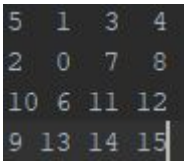

Fig. 7. Nível 4

VIII. Conclusões

As conclusões a que chegamos após o desenvolvimento deste trabalho podem ser interpretadas de várias formas.
O A* é sempre mais rápido e ocupa menos memória que o BFS.
Nós estamos certos de que podemos melhorar ainda mais a heurística usada no A*.
No futuro, gostaríamos de explorar mais formas para conseguir diferenciar os pesos entre os movimentos de maneira a expandir um menor número de nós, chegando mais rapidamente e eficientemente à solução ótima.

Referências Bibliográficas

[1] "Amit's A* Pages", 2019, [online], available at: http://theory.stanford.edu/~amitp/GameProgramming/ , consulted on March 2019.