

Relatório 15-puzzle

Dupla:

- João Pedro de Paula Oliveira - DRE: 113054857
- Rafael da Silva Fernandes - DRE: 117196229

Introdução

O 15-puzzle é um jogo constituído por um tabuleiro com dezesseis posições e quinze peças móveis numeradas. Dada uma configuração aleatória de peças, o objetivo é ordená-las de forma crescente a partir da casa no canto superior esquerdo. Como o tabuleiro só possui uma casa “vazia”, as únicas possibilidades de movimento seriam fazer as outras peças ocuparem essa casa, movendo-as para esquerda, direita, baixo ou para cima.

Neste trabalho visamos explorar uma implementação em prolog do algoritmo de busca informada A-Estrela (A*) para resolução do 15-puzzle. Uma busca informada se difere das buscas tradicionais por seguir uma função heurística de custo, o algoritmo sempre segue o caminho que melhor satisfaça essa função, enquanto uma busca simples procura soluções de forma exaustiva.

Modelagem do problema

• Representação do tabuleiro inicial

O tabuleiro é definido como uma lista simples de 16 posições. Cada posição da lista corresponde a uma casa do tabuleiro 4x4 ordenada pela ordem das casas da esquerda para a direita e das linhas de cima para baixo. Por exemplo, a configuração inicial:

15	2	1	12
8	5	6	11
4	9	10	7
3	14	13	

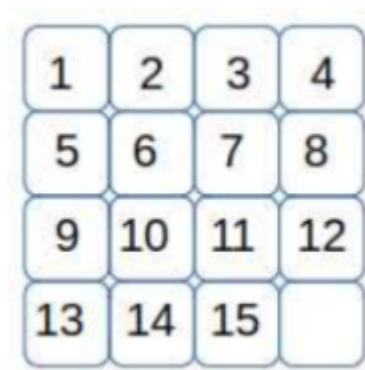
seria representado pela lista

[15, 2, 1, 12, 8, 5, 6, 11, 4, 9, 10, 7, 3, 14, 13, *]

onde o asterisco (*) representa a casa vazia.

No código, o tabuleiro inicial sempre é definido nas cláusulas que fazem uso dele, precisando-se realizar alterações nas linhas quando for desejado testar outras configurações iniciais.

Dessa forma, temos por objetivo organizar esses valores de forma crescente, como representado na figura abaixo.



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

A seguinte lista representa esse estado:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, *]

- **Funções-Heurística**

As funções heurísticas são utilizadas nos algoritmos de busca informada como um “guia” que fornece, a cada etapa, a informação de qual ramificações da árvore de derivação o algoritmo deverá escolher à seguir, evitando a expansão exaustiva de todos os nodos, como acontece nas buscas não-informadas. As heurísticas exploradas no trabalho foram a Distância de Hamming e a Distância Manhattan.

- **Distância Hamming:**

A distância de Hamming é calculada comparando-se a quantidade de posições no estado atual que correspondem às posições do estado final, somando 1 sempre que as posições não conferem com o estado final. Esta função basicamente contabiliza quantas peças ainda estão fora do lugar, quando comparadas ao estado final.

```

% - Número de peças fora do lugar

% Distância entre dois estados
numeroPecasForaLugar([], [], 0) :- !.
numeroPecasForaLugar([A|T1], [B|T2], N) :-
    dif(A, B), % A e B são diferentes
    dif(A, *), % exclui * ( posição vazia )
    !,
    numeroPecasForaLugar(T1, T2, M), % número de peças foras do lugar para os subestados
    N is M + 1. % incrementa distância

numeroPecasForaLugar([_|T1], [_|T2], N) :- numeroPecasForaLugar(T1, T2, N). % não incrementa

% Chamada da função
% Necessário trocar a configuração inicial no código
% Consulta:
% ?-h1(D).
h1(Dist):- objetivo(Objetivo), numeroPecasForaLugar([15, 2, 1, 12, 8, 5, 6, 11, 4, 9, 10, 7, 3, 14, 13, *],
                                                    Objetivo, Dist).

```

Figura W: Implementação da heurística das peças fora do lugar.

O algoritmo recebe duas listas, o estado atual e o estado final (objetivo), além do valor de retorno da heurística. O predicado numeroPecasForaLugar tem como caso base a situação onde nenhum dos estados possui qualquer elemento, e portanto, a heurística é zero. No caso genérico, é comparado se a cabeça das listas são diferentes entre si, desconsiderando a casa vazia e é feita a recursão com as caudas das duas listas. Caso a variável A não seja diferente de vazia, a recursão é feita sem incrementar o valor da função heurística.

○ Distância Manhattan:

A distância Manhattan, também conhecida como geometria do táxi, calcula quantos movimentos verticais e horizontais precisam ser feitos para alcançar o objetivo. Esta heurística tem a propriedade de que o menor caminho sempre terá o mesmo comprimento, independentemente da escolha das direções dos movimentos, contando que não haja ciclos.

```

% - Distância Manhattan
% Necessário trocar a configuração inicial na linha 96
% Consulta:
% ?-h2(D).
h2(DistManhattan):-
    [A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P] = [15, 2, 1, 12, 8, 5, 6, 11, 4, 9, 10, 7, 3, 14, 13, *],
    a(A, Da), b(B, Db), c(C, Dc), d(D, Dd),
    e(E, De), f(F, Df), g(G, Dg), h(H, Dh),
    i(I, Di), j(J, Dj), k(K, Dk), l(L, Dl),
    m(M, Dm), n(N, Dn), o(O, Do), p(P, Dp),
    DistManhattan is Da + Db + Dc + Dd + De + Df + Dg + Dh + Di + Dj + Dk + Dl + Dm + Dn + Do + Dp.

```

Figura X: Heurística da Distância Manhattan em Prolog.

Na figura X, temos a implementação do predicado h2, que calcula a distância Manhattan, em Prolog. A configuração inicial precisa estar definida no código, e os valores dos termos no somatório são dados em função dos fatos definidos na figura Y.

```
a(*, 6). a(1, 0). a(2, 1). a(3, 2). a(4, 3). a(5, 1). a(6, 2). a(7, 3). a(8, 5). a(9, 2). a(10, 3). a(11, 4). a(12, 5). a(13, 3). a(14, 4). a(15, 5).
b(*, 5). b(1, 1). b(2, 0). b(3, 1). b(4, 2). b(5, 2). b(6, 1). b(7, 2). b(8, 3). b(9, 3). b(10, 2). b(11, 3). b(12, 4). b(13, 4). b(14, 3). b(15, 4).
c(*, 4). c(1, 2). c(2, 1). c(3, 0). c(4, 1). c(5, 3). c(6, 2). c(7, 1). c(8, 2). c(9, 4). c(10, 3). c(11, 2). c(12, 3). c(13, 5). c(14, 4). c(15, 3).
d(*, 3). d(1, 3). d(2, 2). d(3, 1). d(4, 0). d(5, 4). d(6, 3). d(7, 2). d(8, 1). d(9, 5). d(10, 4). d(11, 3). d(12, 2). d(13, 6). d(14, 5). d(15, 4).
e(*, 5). e(1, 1). e(2, 2). e(3, 3). e(4, 4). e(5, 0). e(6, 1). e(7, 2). e(8, 3). e(9, 1). e(10, 2). e(11, 3). e(12, 4). e(13, 2). e(14, 3). e(15, 4).
f(*, 4). f(1, 2). f(2, 1). f(3, 2). f(4, 3). f(5, 1). f(6, 0). f(7, 1). f(8, 2). f(9, 2). f(10, 1). f(11, 2). f(12, 3). f(13, 3). f(14, 2). f(15, 3).
g(*, 3). g(1, 3). g(2, 2). g(3, 1). g(4, 2). g(5, 2). g(6, 1). g(7, 0). g(8, 1). g(9, 3). g(10, 2). g(11, 1). g(12, 2). g(13, 4). g(14, 3). g(15, 2).
h(*, 2). h(1, 4). h(2, 3). h(3, 2). h(4, 1). h(5, 3). h(6, 2). h(7, 1). h(8, 0). h(9, 4). h(10, 3). h(11, 2). h(12, 1). h(13, 5). h(14, 4). h(15, 3).
i(*, 4). i(1, 2). i(2, 3). i(3, 4). i(4, 5). i(5, 1). i(6, 2). i(7, 3). i(8, 4). i(9, 0). i(10, 1). i(11, 2). i(12, 3). i(13, 1). i(14, 2). i(15, 3).
j(*, 3). j(1, 3). j(2, 2). j(3, 3). j(4, 4). j(5, 2). j(6, 1). j(7, 2). j(8, 3). j(9, 1). j(10, 0). j(11, 1). j(12, 2). j(13, 2). j(14, 1). j(15, 2).
k(*, 2). k(1, 4). k(2, 3). k(3, 2). k(4, 3). k(5, 3). k(6, 2). k(7, 1). k(8, 2). k(9, 2). k(10, 1). k(11, 0). k(12, 1). k(13, 3). k(14, 2). k(15, 1).
l(*, 1). l(1, 5). l(2, 4). l(3, 3). l(4, 2). l(5, 4). l(6, 3). l(7, 2). l(8, 1). l(9, 3). l(10, 2). l(11, 1). l(12, 0). l(13, 4). l(14, 3). l(15, 2).
m(*, 3). m(1, 3). m(2, 4). m(3, 5). m(4, 6). m(5, 2). m(6, 3). m(7, 4). m(8, 5). m(9, 1). m(10, 2). m(11, 3). m(12, 4). m(13, 0). m(14, 1). m(15, 2).
n(*, 2). n(1, 4). n(2, 3). n(3, 4). n(4, 5). n(5, 3). n(6, 2). n(7, 3). n(8, 4). n(9, 2). n(10, 1). n(11, 2). n(12, 3). n(13, 1). n(14, 0). n(15, 1).
o(*, 1). o(1, 5). o(2, 4). o(3, 3). o(4, 4). o(5, 4). o(6, 3). o(7, 2). o(8, 3). o(9, 3). o(10, 2). o(11, 1). o(12, 2). o(13, 2). o(14, 1). o(15, 0).
p(*, 0). p(1, 6). p(2, 5). p(3, 4). p(4, 3). p(5, 5). p(6, 4). p(7, 3). p(8, 2). p(9, 4). p(10, 3). p(11, 2). p(12, 1). p(13, 3). p(14, 2). p(15, 1).
```

Figura Y: Fatos que definem a distância Manhattan de cada possibilidade.

Na figura Y, foram definidas todas as possibilidades de distâncias como fatos. O fato j(8,3) significa, por exemplo, que na casa “j” temos o valor 8, e que este está à três posições de sua casa “correta”, a casa “h”.

Voltando ao código da Figura X, as variáveis representadas pelas letras de A até P são inicializadas com os valores da lista que representa o estado inicial, as variáveis representadas por Da até Dp guardam os valores que casam com os fatos correspondentes e então é feito o somatório de todas elas para obter o valor final da Distância Manhattan.

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

- **Apresentação e explicação de cada regra**

- **Movimentos**

Os movimentos possíveis são feitos na vertical ou na horizontal, onde apenas uma peça é movida para o espaço da casa vazia, deixando um novo espaço vazio em seu lugar.

No prolog, foram criados onze fatos para cada tipo de movimento possível, representando para um dado estado, todas as possibilidades de configurações antes e depois do movimento.

O fato:

`esquerda([A,*,C,D,E,F,G,H,I,J,K,L,M,N,O,P], [*,A,C,D,E,F,G,H,I,J,K,L,M,N,O,P]).`

significa que haverá match quando o segundo estado mantiver toda a configuração do primeiro estado, exceto a casa em vazia (o asterisco) que troca de lugar com a casa que estiver à sua esquerda. A mesma lógica é válida para os fatos direita, cima e baixo.

A	*	C	D	→	*	A	C	D
E	F	G	H	→	E	F	G	H
I	J	K	L	→	I	J	K	L
M	N	O	P	→	M	N	O	P

■ Esquerda

% Movimentos

% - Esquerda

```
esquerda([A,*,C,D,E,F,G,H,I,J,K,L,M,N,O,P], [*,A,C,D,E,F,G,H,I,J,K,L,M,N,O,P]).
esquerda([A,B,*,D,E,F,G,H,I,J,K,L,M,N,O,P], [A,*,B,D,E,F,G,H,I,J,K,L,M,N,O,P]).
esquerda([A,B,C,*,E,F,G,H,I,J,K,L,M,N,O,P], [A,B,*,C,E,F,G,H,I,J,K,L,M,N,O,P]).
esquerda([A,B,C,D,E,*,G,H,I,J,K,L,M,N,O,P], [A,B,C,D,*,E,G,H,I,J,K,L,M,N,O,P]).
esquerda([A,B,C,D,E,F,*,H,I,J,K,L,M,N,O,P], [A,B,C,D,E,*,F,H,I,J,K,L,M,N,O,P]).
esquerda([A,B,C,D,E,F,G,*,I,J,K,L,M,N,O,P], [A,B,C,D,E,F,*,G,I,J,K,L,M,N,O,P]).
esquerda([A,B,C,D,E,F,G,H,I,*,K,L,M,N,O,P], [A,B,C,D,E,F,G,H,*,I,K,L,M,N,O,P]).
esquerda([A,B,C,D,E,F,G,H,I,J,*,L,M,N,O,P], [A,B,C,D,E,F,G,H,I,*,J,L,M,N,O,P]).
esquerda([A,B,C,D,E,F,G,H,I,J,K,*,M,N,O,P], [A,B,C,D,E,F,G,H,I,J,*,K,M,N,O,P]).
esquerda([A,B,C,D,E,F,G,H,I,J,K,L,M,*,O,P], [A,B,C,D,E,F,G,H,I,J,K,L,*,M,O,P]).
esquerda([A,B,C,D,E,F,G,H,I,J,K,L,M,N,*,P], [A,B,C,D,E,F,G,H,I,J,K,L,M,*,N,P]).
```

■ Direita

% - Direita

```
direita([*,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P], [B,*,C,D,E,F,G,H,I,J,K,L,M,N,O,P]).
direita([A,*,C,D,E,F,G,H,I,J,K,L,M,N,O,P], [A,C,*,D,E,F,G,H,I,J,K,L,M,N,O,P]).
direita([A,B,*,D,E,F,G,H,I,J,K,L,M,N,O,P], [A,B,D,*,E,F,G,H,I,J,K,L,M,N,O,P]).
direita([A,B,C,D,*,F,G,H,I,J,K,L,M,N,O,P], [A,B,C,D,F,*,G,H,I,J,K,L,M,N,O,P]).
direita([A,B,C,D,E,*,G,H,I,J,K,L,M,N,O,P], [A,B,C,D,E,G,*,H,I,J,K,L,M,N,O,P]).
direita([A,B,C,D,E,F,*,H,I,J,K,L,M,N,O,P], [A,B,C,D,E,F,H,*,I,J,K,L,M,N,O,P]).
direita([A,B,C,D,E,F,G,H,*,J,K,L,M,N,O,P], [A,B,C,D,E,F,G,H,J,*,K,L,M,N,O,P]).
direita([A,B,C,D,E,F,G,H,I,*,K,L,M,N,O,P], [A,B,C,D,E,F,G,H,I,K,*,L,M,N,O,P]).
direita([A,B,C,D,E,F,G,H,I,J,*,L,M,N,O,P], [A,B,C,D,E,F,G,H,I,J,L,*,M,N,O,P]).
direita([A,B,C,D,E,F,G,H,I,J,K,L,*,N,O,P], [A,B,C,D,E,F,G,H,I,J,K,L,N,*,O,P]).
direita([A,B,C,D,E,F,G,H,I,J,K,L,M,*,O,P], [A,B,C,D,E,F,G,H,I,J,K,L,M,O,*,P]).
direita([A,B,C,D,E,F,G,H,I,J,K,L,M,N,*,P], [A,B,C,D,E,F,G,H,I,J,K,L,M,N,P,*]).
```

■ Cima

% - Cima

```
cima([A,B,C,D,*,F,G,H,I,J,K,L,M,N,O,P], [*,B,C,D,A,F,G,H,I,J,K,L,M,N,O,P]).
cima([A,B,C,D,E,*,G,H,I,J,K,L,M,N,O,P], [A,*,C,D,E,B,G,H,I,J,K,L,M,N,O,P]).
cima([A,B,C,D,E,F,*,H,I,J,K,L,M,N,O,P], [A,B,*,D,E,F,C,H,I,J,K,L,M,N,O,P]).
cima([A,B,C,D,E,F,G,*,I,J,K,L,M,N,O,P], [A,B,C,*,E,F,G,D,I,J,K,L,M,N,O,P]).
cima([A,B,C,D,E,F,G,H,*,J,K,L,M,N,O,P], [A,B,C,D,*,F,G,H,E,J,K,L,M,N,O,P]).
cima([A,B,C,D,E,F,G,H,I,*,K,L,M,N,O,P], [A,B,C,D,E,*,G,H,I,F,K,L,M,N,O,P]).
cima([A,B,C,D,E,F,G,H,I,J,*,L,M,N,O,P], [A,B,C,D,E,F,*,H,I,J,G,L,M,N,O,P]).
cima([A,B,C,D,E,F,G,H,I,J,K,*,M,N,O,P], [A,B,C,D,E,F,G,*,I,J,K,H,M,N,O,P]).
cima([A,B,C,D,E,F,G,H,I,J,K,L,*,N,O,P], [A,B,C,D,E,F,G,H,*,J,K,L,I,N,O,P]).
cima([A,B,C,D,E,F,G,H,I,J,K,L,M,*,O,P], [A,B,C,D,E,F,G,H,I,*,K,L,M,J,O,P]).
cima([A,B,C,D,E,F,G,H,I,J,K,L,M,N,*,P], [A,B,C,D,E,F,G,H,I,J,*,L,M,N,K,P]).
cima([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,*], [A,B,C,D,E,F,G,H,I,J,K,*,M,N,O,L]).
```

■ Baixo

% - Baixo

```
baixo([*,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P], [E,B,C,D,*,F,G,H,I,J,K,L,M,N,O,P]).
baixo([A,*,C,D,E,F,G,H,I,J,K,L,M,N,O,P], [A,F,C,D,E,*,G,H,I,J,K,L,M,N,O,P]).
baixo([A,B,*,D,E,F,G,H,I,J,K,L,M,N,O,P], [A,B,G,D,E,F,*,H,I,J,K,L,M,N,O,P]).
baixo([A,B,C,*,E,F,G,H,I,J,K,L,M,N,O,P], [A,B,C,H,E,F,G,*,I,J,K,L,M,N,O,P]).
baixo([A,B,C,D,*,F,G,H,I,J,K,L,M,N,O,P], [A,B,C,D,I,F,G,H,*,J,K,L,M,N,O,P]).
baixo([A,B,C,D,E,*,G,H,I,J,K,L,M,N,O,P], [A,B,C,D,E,J,G,H,I,*,K,L,M,N,O,P]).
baixo([A,B,C,D,E,F,*,H,I,J,K,L,M,N,O,P], [A,B,C,D,E,F,K,H,I,J,*,L,M,N,O,P]).
baixo([A,B,C,D,E,F,G,*,I,J,K,L,M,N,O,P], [A,B,C,D,E,F,G,L,I,J,K,*,M,N,O,P]).
baixo([A,B,C,D,E,F,G,H,*,J,K,L,M,N,O,P], [A,B,C,D,E,F,G,H,M,J,K,L,*,N,O,P]).
baixo([A,B,C,D,E,F,G,H,I,*,K,L,M,N,O,P], [A,B,C,D,E,F,G,H,I,N,K,L,M,*,O,P]).
baixo([A,B,C,D,E,F,G,H,I,J,*,L,M,N,O,P], [A,B,C,D,E,F,G,H,I,J,O,L,M,N,*,P]).
baixo([A,B,C,D,E,F,G,H,I,J,K,*,M,N,O,P], [A,B,C,D,E,F,G,H,I,J,K,P,M,N,O,*]).
```

○ Predicados de movimento

```
movimento(X, Y, esquerda, 1):- esquerda(X, Y).
movimento(X, Y, direita, 1):- direita(X, Y).
movimento(X, Y, cima, 1):- cima(X, Y).
movimento(X, Y, baixo, 1):- baixo(X, Y).
```

Figura T: Predicados para os movimentos.

Conforme a Figura T, foram definidos predicados que utilizam os fatos abordados anteriormente para realizar os movimentos nos tabuleiros. X e Y são variáveis que correspondem às configurações de entrada e saída. Também são passados a direção do movimento e seu custo, que neste problema é constante e tem valor 1.

○ Busca A*

O A* é um algoritmo de busca heurístico baseado na busca em largura em que uma função $f(n)$ combina o custo $g(n)$ de chegar até o nodo com o custo $h(n)$ de chegar ao objetivo a partir do nodo, ou seja, a função tem a forma de

$$f(n) = g(n) + h(n)$$

Como $g(n)$ fornece o custo do caminho do início até o nodo n e $h(n)$ é o custo estimado para o caminho mais barato de n até o objetivo, nós temos

$$f(n) = \text{custo mais barato estimado da solução que passa por } n$$

Portanto, se estivermos tentando chegar na solução mais barata, uma estratégia razoável seria tentar chegar primeiro ao nodo com o menor valor de $g(n) + h(n)$. Acontece que esta estratégia é mais do que razoável: desde que a função heurística $h(n)$ satisfaça certas condições, a busca A* é tanto completa quanto ótima.

O algoritmo é idêntico à Busca de Custo Uniforme, exceto que A* usa $g + h$, em vez de apenas g .

```
% Adiciona o nodo numa fila
addNodo(Nodo, [], [Nodo]) :- !.
addNodo(node(Estado1, Caminho1, Custo1), [node(Estado2, Caminho2, Custo2)|Lista],
        [node(Estado1, Caminho1, Custo1),node(Estado2, Caminho2, Custo2)|Lista]) :- Custo1 =< Custo2, !.
addNodo(node(Estado1, Caminho1, Custo1), [node(Estado2, Caminho2, Custo2)|Lista0],
        [node(Estado2, Caminho2, Custo2)|Lista]) :-
    addNodo(node(Estado1, Caminho1, Custo1), Lista0, Lista).

% Adiciona uma lista de nodos na fila
addLista([], Lista, Lista).
addLista([Nodo|Resto], Lista0, Lista):- addNodo(Nodo, Lista0, Lista1), addLista(Resto, Lista1, Lista).

% Consulta: ?-soluciona(Caminho)
soluciona(Caminho) :-
    Estado = [13, 2, 10, 3, 1, 12, 8, 4, 5, 0, 9, 6, 15, 14, 11, 7],
    % h1(H),
    h2(H),
    aEstrela([node(Estado, [], H)], [Estado], Caminho).

% Algoritmo de busca A*
% Está sempre retornando false e não descobrimos à tempo o porquê
aEstrela([node(Objetivo, Caminho, _)|_], _, Caminho) :- objetivo(Objetivo).
aEstrela([node(Estado, Caminho, Custo)|Fila], Visitado, Solucao) :-
    findall(Vizinhos, movimento(Estado, Vizinhos, _, 1), EstadoVizinhos), % encontra estados vizinhos
    expandeFilhos(node(Estado, Caminho, Custo), EstadoVizinhos, Visitado, Filhos), % expande filhos
    addLista(Filhos, Fila, FilaOrdenados), % adiciona filho à Lista
    aEstrela(FilaOrdenados, [Estado|Visitado], Solucao).

% Expande filhos do nodo
expandeFilhos(_, [], _, []) :- !. % nodo não possui filhos
expandeFilhos(node(Pai, Caminho, FPai), [Filho|Resto], Visitado,
        [node(Filho, [Filho|Caminho], F)|Filhos]) :-
    not(member(Filho, Visitado)), % estado nodo filho não foi visitado
    %length(Caminho, G1), h1(H), F is G1 + 1 + H, % f(nodo filho)
    length(Caminho, G1), h2(H), F is G1 + 1 + H, % f(nodo filho)
    expandeFilhos(node(Pai, Caminho, FPai), Resto, [Filho|Visitado], Filhos), !.
expandeFilhos(Nodo, [_|Resto], Visitado, Filhos) :- % nodo filho foi visitado
    expandeFilhos(Nodo, Resto, Visitado, Filhos). % tente outros filhos
```


- **Apresentação e explicação do teste para descobrir há ou não uma solução para a configuração inicial do 15-puzzle apresentada**

Em geral, para uma dada grade de comprimento N, nós conseguimos chegar se um $(N * N - 1)$ -puzzle tem solução ou não seguindo simples regras.

No nosso caso, temos que N possui o valor de 4, que é um número par, e portanto essas são as regras que devemos seguir:

- a casa vazia está em uma linha de número par, contando de baixo para cima, e o número de inversões for ímpar
- a casa vazia está em uma linha de número ímpar, contando de baixo para cima, e o número de inversões for par

O número de inversões é calculado considerando os ladrilhos em uma única linha, ao invés de uma matriz, um par de ladrilhos (a, b) formam uma inversão se a aparece antes de b, mas $a > b$.

Por exemplo, consideremos a seguinte situação:

[2, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, *]

A lista acima possui apenas uma inversão, isto é, (2, 1).

```
% Teste para saber se há caminho final ou não
```

```
% Caso haja, retorna true
```

```
% Caso não, retorna false
```

```
%
```

```
% Consulta:
```

```
% ?-solucionavel.
```

```
% Necessário trocar as configurações iniciais nas linhas 131 e 140
```

```
solucionavel:-
```

```
Estado = [15, 2, 1, 12, 8, 5, 6, 11, 4, 9, 10, 7, 3, 14, 13, *],
```

```
posicao(Estado, Pos),
```

```
linha(Pos, Linha), % confere em que linha está * ( posição vazia )
```

```
par(Linha), % confere se essa linha é par
```

```
!, % caso seja par, segue
```

```
contaInversao(Estado, Res),
```

```
impar(Res).
```

```
solucionavel:-
```

```
Estado = [15, 2, 1, 12, 8, 5, 6, 11, 4, 9, 10, 7, 3, 14, 13, *],
```

```
posicao(Estado, Pos),
```

```
linha(Pos, Linha), % confere em que linha está * ( posição vazia )
```

```
impar(Linha), % confere se essa linha é ímpar
```

```

posicao([* | _], 1):- !.
posicao([_ | Tail], Pos):-
    posicao(Tail, Pos1),
    Pos is Pos1 + 1.

linha(Pos, 1):-
    Pos =< 16,
    Pos >= 13, !.
linha(Pos, 2):-
    Pos =< 12,
    Pos >= 9, !.
linha(Pos, 3):-
    Pos =< 8,
    Pos >= 5, !.
linha(Pos, 4):-
    Pos =< 4,
    Pos >= 1.

par(Num):- Num mod 2 == 0, !.

impar(Num):- Num mod 2 \= 0.

analisaDupla(Head, [* | Tail], Res) :-
    !, analisaDupla(Head, Tail, Res).
analisaDupla(_, [], 0).
analisaDupla(Head, [Head2 | Tail], Res) :-
    Head > Head2, !,
    analisaDupla(Head, Tail, Res1),
    Res is Res1 + 1.

analisaDupla(Head, [Head2 | Tail], Res) :-
    Head < Head2,
    analisaDupla(Head, Tail, Res).

% Consulta:
% ?-contaInversao([13, 2, 10, 3, 1, 12, 8, 4, 5, *, 9, 6, 15, 14, 11, 7], R).
contaInversao([* | Tail_Est], Total) :-
    contaInversao(Tail_Est, Total).
contaInversao([], 0).
contaInversao([Head_Est | Tail_Est], Total) :-
    analisaDupla(Head_Est, Tail_Est, Res),
    contaInversao(Tail_Est, Total1),
    Total is Res + Total1, !.

```

Apresentação do resultado:

O predicado *solucionavel* é utilizado nas consultas para saber se a configuração inicial representada no código possui solução ou não. Dependendo do tabuleiro, a consulta ? – *solucionavel* retorna true ou false, caso o problema tenha solução, ou não.

As seguintes configurações iniciais foram utilizadas para os testes:

- [15, 2, 1, 12, 8, 5, 6, 11, 4, 9, 10, 7, 3, 14, 13, *]
 - ?-h1(D) | D = 13
 - ?-h2(D) | D = 32
 - ?-solucionavel | false
 - ?-contaInversao([15, 2, 1, 12, 8, 5, 6, 11, 4, 9, 10, 7, 3, 14, 13, *], R) | R = 45
- [13, 2, 10, 3, 1, 12, 8, 4, 5, *, 9, 6, 15, 14, 11, 7]
 - ?-h1(D). | D = 13
 - ?-h2(D). | D = 32
 - ?-solucionavel. | true
 - ?-contaInversao([13, 2, 10, 3, 1, 12, 8, 4, 5, *, 9, 6, 15, 14, 11, 7], R) | R = 41
- [6, 13, 7, 10, 8, 9, 11, *, 15, 2, 12, 5, 14, 3, 1, 4]
 - ?-h1(D) | D = 15
 - ?-h2(D) | D = 42
 - ?-solucionavel | true
 - ?-contaInversao([6, 13, 7, 10, 8, 9, 11, *, 15, 2, 12, 5, 14, 3, 1, 4], R) | R = 62
- [3, 9, 1, 15, 14, 11, 4, 6, 13, *, 4, 10, 12, 2, 7, 8, 5]
 - ?-h1(D) | D = 14
 - ?-h2(D) | D = 46
 - ?-solucionavel | false
 - ?-contaInversao([3, 9, 1, 15, 14, 11, 6, 13, *, 4, 10, 12, 2, 7, 8, 5], R) | R = 58

Análise dos resultados:

Os resultados cumpriram com a expectativa para os cláusulas que foram testadas, entretanto, o algoritmo de busca A* resulta sempre em *false* e não conseguimos descobrir a tempo o porquê disso estar acontecendo. Dessa forma, podemos apenas calcular as heurísticas, e dizer se um dado 15-puzzle possui ou não solução, e essas cláusulas funcionam como o esperado.

Dificuldades e melhorias futuras:

A implementação do A* não ficou satisfatória e não conseguimos identificar as causas do problema no algoritmo.

A função heurística acabou ficando “hardcoded”, precisando comentar ou tirar o comentário das linhas respectivas à heurística a ser utilizada. O ideal seria conseguir passá-las como parâmetro durante a consulta.

Consideramos que a modelagem do problema poderia ter sido implementada de uma forma melhor, sem a necessidade de criar inúmeros fatos para cada situação possível da distância Manhattan ou dos movimentos em cada direção. Se fossemos criar um tabuleiro maior isso seria inviável.

Contornando estes problemas apresentados, consideramos que o trabalho estaria satisfatório e poderíamos pensar em expandi-lo, por exemplo, para usar novas heurísticas, ou algoritmos de busca, ou implementar a possibilidade de receber uma configuração de tabuleiro pela consulta.

Referências:

- seção 3.5.2 do arquivo cap 3-busca informada.pdf
- <https://stackoverflow.com/questions/24630245/how-to-solve-15-puzzle-paradigm-in-prolog-with-manhattan-hamming-heuristics>
- https://www.cpp.edu/~jrfisher/www/prolog_tutorial/5_1.html
- https://www.cpp.edu/~jrfisher/www/prolog_tutorial/5_2.html
- https://www.cpp.edu/~jrfisher/www/prolog_tutorial/8_puzzlepl.txt
- <https://www.geeksforgeeks.org/check-instance-15-puzzle-solvable/>