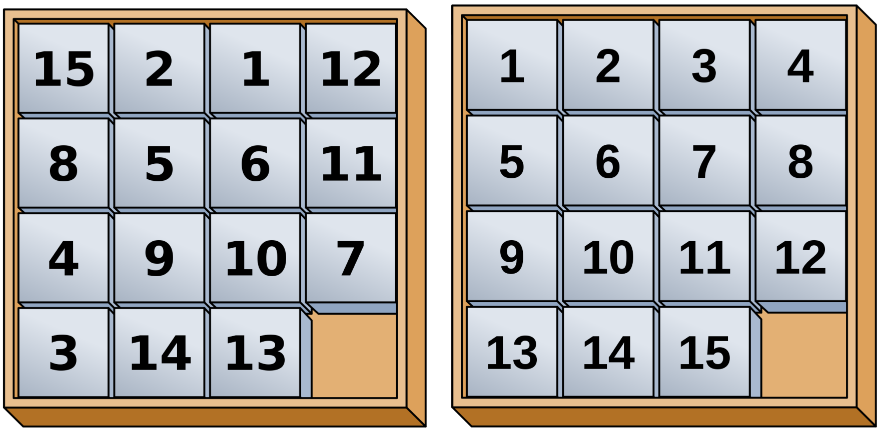
Nome: João Vitor de Freitas Barbosa **DRE:** 117055449
Nome: Gabriel Martins Machado Christo **DRE:** 117217732

Tarefa 5

Busca Informada - 15 Puzzle

Introdução do problema

O Puzzle 15 é um jogo famoso, muito utilizado no contexto da computação com a finalidade de testar a funcionalidade algoritmos de busca. O objetivo do jogador é deslizar as peças até que se chegue ao estado final:



[imagens disponíveis no wikipedia]

Neste trabalho estaremos interessados em modelar a solução do Puzzle 15 como um problema de busca e utilizar o algoritmo A* para encontrar o caminho ótimo de um estado inicial válido até o estado final.

Modelagem do problema

Com objetivo de representar o problema no PROLOG, faremos a modelagem para representar o tabuleiro e suas mudanças de estado.

Sejam

```
Q = conjunto de nós a serem pesquisados;
S = o estado inicial da busca
```

Faça:

1. Inicialize **Q** com o nó de busca **(S)** como única entrada;
2. Se **Q** está vazio, interrompa. Se não, escolha o melhor elemento de **Q**;
3. Se o estado **(n)** é um objetivo, retorne **n**;
4. (De outro modo) Remova **n** de **Q**;
5. Encontre os descendentes do estado **(n)** que não estão em visitados e crie todas as extensões de **n** para cada descendente;
6. Adicione os caminhos estendidos a **Q** e vá ao passo **2**;

```
caminhos expandidos;
```

Uma estimativa que sempre subestima o comprimento real do caminho ate o objetivo é chamada de admissível. O uso de uma estimativa admissível garante que a busca de custo-uniforme ainda encontrará o menor caminho.

[Algoritmo A*]

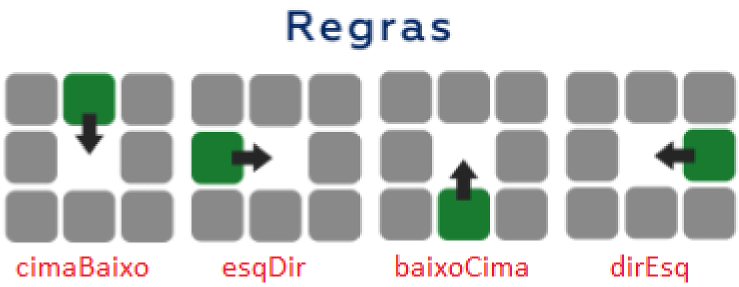
Representação do Tabuleiro

Os tabuleiros são representados como matrizes, com representação sendo feito por listas de listas. O espaço em branco é a constante x. A matriz inicial é dada pela seguinte cláusula Prolog:

```
matriz([ [15, 2, 1, 12],
         [8, 5, 6, 11],
         [4, 9, 10, 7],
         [3, 14, 13, x] ]).
```

O código foi feito para ser compatível com qualquer matriz quadrada com tamanho N > 1.

Regras do Jogo



Podemos pensar nas regras de transição do jogo como a troca de posição do x (espaço em branco) com alguma peça vizinha. No código PROLOG nossa abordagem foi:

- Acessar a matriz do nó atual.
- Encontrar a posição da célula vazia.
- Encontrar a posição de um vizinho.
- Trocar essas peças de posição.

Mas existem quatro possibilidades quando usamos esta abordagem, cada uma relacionada a posição do vizinho. Por conta disso, criamos quatro cláusulas, uma para cada regra. Um exemplo da regra $dirEsq$:

```
regra(Node, dirEsq, (Final, G2, _)):-
    Node = (Matriz, G, _),
    matriz_encontrar(Matriz, x, (X,Y)),
    AlvoX is X+1,
    AlvoY is Y,
    matriz_trocaPeca(Matriz, (X,Y), (AlvoX, AlvoY), Final),
    G2 is G + 1.
```

O predicado exposto acima recebe um Nó da árvore e seu valor de G (custo até chegar no nó). A aplicação da regra aumenta o valor em G, representando o custo de andar do Nó atual até o Nó vizinho. Além disso, podemos destacar os predicados $matriz\_encontrar$ e $matriz\_trocaPeca$, desenvolvidos para ajudar na manipulação da matriz em si. As regras restantes são semelhantes, mudando apenas a forma como a tupla $(AlvoX,\ AlvoY)$ é calculada.

Tabuleiro Válido

Sabemos que não é qualquer tipo de tabuleiro que possui solução. Portanto, é necessário criar um predicado que valida o tabuleiro, o qual retorna se o mesmo possui solução ou não. As restrições foram consultadas na [página geeksforgeeks]() e traduzidas para o prolog em três predicados:

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

N = 4 (Even)
Position of X from bottom = 2 (Even)
Inversion Count = 56 (Even)
➔ Not Solvable

```
matriz_valida(Matriz):-
    matriz_tamanho(Matriz, N, N),
    impar(N),
    matriz_quantInversoes(Matriz, Inversoes),
    not(impar(Inversoes)).

matriz_valida(Matriz):-
    matriz_tamanho(Matriz, N, N),
    not(impar(N)),
    matriz_encontrar(Matriz, x, (_,I)),
    Row is N - I,
    not(impar(Row)),
    matriz_quantInversoes(Matriz, Inversoes),
    impar(Inversoes),!.

matriz_valida(Matriz):-
    matriz_tamanho(Matriz, N, N),
    not(impar(N)),
    matriz_encontrar(Matriz, x, (_,I)),
    Row is N - I,
    impar(Row),
    matriz_quantInversoes(Matriz, Inversoes),
    not(impar(Inversoes)),!.
```

Os predicados expostos acima representam respectivamente que a matriz é válida, se respeita as seguintes regras:

- Seu tamanho for ímpar e sua quantidade de inversões for par;
- Seu tamanho for par e um dos casos abaixo for satisfeito:
    - A posição do buraco contando de baixo para cima é par e a quantidade de inversões é ímpar.
    - A posição do buraco contando de baixo para cima é ímpar e a quantidade de inversões é par.

Qualquer outro caso a matriz é inválida e, portanto, não possui uma solução.

matriz dada pela proposta do trabalho

Estes predicados avaliaram a matriz dada no enunciado do trabalho e concluíram que ela é inválida, pois:

- Tamanho N = 4, ou seja, par;
- Posição do espaço em branco de baixo para cima é 1, ou seja, ímpar
- Possui 45 inversões, ou seja, ímpar.

Heurísticas

h1: número de peças fora do lugar (errados)

O programa percorre a matriz e soma 1 sempre que a distância manhattan for maior que 0 (o número está fora do lugar).

```
% heuristica(++Tipo, +Matriz, -Estimativa)
heuristica(errados, Matriz, Estimativa):-
    matriz_paraLista(Matriz, Lista),
    erradoLista(Lista, Matriz, Estimativa).

% erradoLista (+Lista, +Matriz, -Custo)
erradoLista([], _, 0).
erradoLista([Elemento|Proximos], Matriz, Custo):-
    distanciaManhattan(Elemento, Matriz, Manhattan),
    Manhattan > 0, !,
    erradoLista(Proximos, Matriz, C1),
    Custo is C1 + 1.
erradoLista([_|Proximos], Matriz, Custo):-
    erradoLista(Proximos, Matriz, Custo).
```

h2: distância Manhattan (Manhattan)

O programa faz o somatório das distâncias Manhattan de cada elemento. A distância manhattan é a soma dos movimentos horizontais e verticais necessários para o elemento estar no lugar ideal.

```
% heuristica(++Tipo, +Matriz, -Estimativa)
heuristica(manhattan, Matriz, Estimativa):-
    matriz_paraLista(Matriz, Lista),
    manhattanLista(Lista, Matriz, Estimativa).

% manhattanLista (+Lista, +Matriz, -Custo)
manhattanLista([], _, 0).
manhattanLista([Elemento|Proximos], Matriz, Custo):-
    manhattanLista(Proximos, Matriz, C1),
    distanciaManhattan(Elemento, Matriz, C2),
    Custo is C1 + C2.

% distanciaManhattan (+Elemento, +Matriz, -Custo)
distanciaManhattan(Elemento, Matriz, Custo):-
    matriz_encontrar(Matriz, Elemento, (J, I)),
    coordenadasIdeais(Elemento, Matriz, (E_J, E_I)),
    Custo is abs(I-E_I) + abs(J-E_J).

% coordenadasIdeais(+Elemento, +Matriz, -Coordenadas)
```

```
coordenadasIdeais(Elemento, Matriz, (E_J, E_I)):-
    matriz_tamanho(Matriz, Tam, _),
    Value is Elemento - 1,
    E_J is mod(Value, Tam),
    E_I is (Value-E_J)/Tam.
```

Apresentação dos resultados obtidos

O programa retorna as ações feitas até o objetivo e a quantidade de nós gerados, para chamar o programa basta consultar:

```
% Para a Heuristica Fora do Lugar
busca_AStar(errados, Acoes, QuantidadeErrados),

% Para a Heuristica Distância Manhattan
busca_AStar(manhattan, Acoes, QuantidadeManhattan).
```

Análise dos resultados

Ao realizar os testes, constatamos que a nossa implementação consegue resolver facilmente casos simples e com profundidades pequenas.

No	Configuração Inicial	Profund.	Heurística	Achou?	Gerados
1	`matriz([[ 1, 2, 3], [ 5, 6, x], [ 7, 8, 4]]).`	13	Errados	S	264
			Manhattan	S	157
2	`matriz([[ 1, 2, 3, 4], [ 5, 6, x, 7], [ 9, 10, 11, 8], [13, 14, 15, 12]]).`	3	Errados	S	9
			Manhattan	S	9
3	`matriz([[ 5, 1, 3, 4], [ 2, x, 7, 8], [ 9, 6, 10, 12], [13, 14, 11, 15]]).`	7	Errados	S	30
			Manhattan	S	28
4	`matriz([[ 2, 6, 3, 4], [ 1, 11, x, 7], [ 5, 14, 10, 8], [ 9, 13, 15, 12]]).`	13	Errados	S	42
			Manhattan	S	31

5	`matriz([[ 2,  11,  6,  4],`
	`[ 1, 14,  3,  7],`
	`[ 5, 13, 10,  8],`
	`[ 9, 15, x, 12]]).`
		19	Errados	S	475

			Manhattan	S	62

6	`matriz([[ 3,  9,  1,  15],`
	`[ 14, 11,  4,  6],`
	`[ 13, x, 10,  12],`
	`[ 2, 7, 8, 5]]).`
		Inválida	Errados	N	

			Manhattan	N	

7	`matriz([[ 15,  2,  1,  12],`
	`[ 8, 5,  6,  11],`
	`[ 4, 9, 10,  7],`
	`[ 3, 14, 13, x]]).`
		Inválida	Errados	N	

			Manhattan	N	

O aumento da profundidade aumenta o uso de memória, pois está armazenando a fronteira e os nós gerados, ambos tendem a crescer ao longo da busca. Percebemos também que nos testes, a heurística $Manhattan$ sempre se saiu igual ou superior à $Errados$. Em casos com profundidade maior, isso teve como consequência, um número muito menor de nós gerados, em especial no caso 5.

O algoritmo A* se mostrou muito ineficiente com uso de memória, o que pode ser piorado com uso de heurísticas que não estimam bem o custo de chegar no estado final, ainda mais considerando que este puzzle é considerado de categoria NP Completo, que significa que no pior caso, achar a sequência de ações mais curta envolve percorrer todos os nós. Pelas nossas execuções ficou claro que a Heurística `Manhattan` se saiu melhor que a `Errados`, gerando menos nós e, portanto, ocupamos menos memória e sendo mais eficientes.