# Laboratório 03:

## Sistemas de Busca e Algoritmos Adversariais

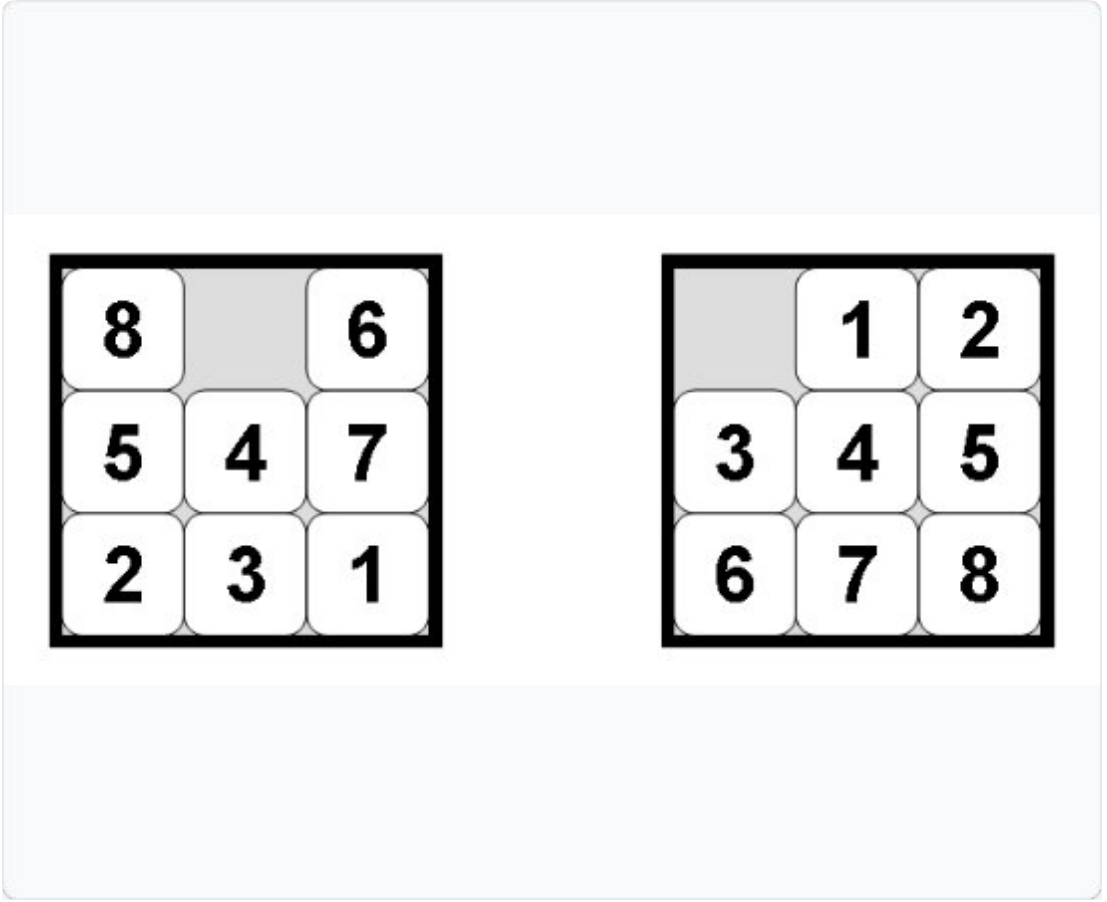
CC7711 - Inteligência Artificial

Prof. Dr. Murillo | Depto. de Ciência da Computação

# O Problema: Jogo dos Oito

Para entender os mecanismos de busca, resolveremos um problema clássico.

- Espaço de Estados: Uma grade 3x3 com 8 peças numeradas e um espaço vazio.
- Objetivo: Ordenar as peças de 1 a 8 movendo o espaço vazio.
- Desafio: Encontrar a sequência de movimentos que leva do estado inicial bagunçado ao estado ordenado.



# Representação: Tupla vs. Lista

## Por que não usar Listas `[1, 2, 3]`?

Em Python, listas são *mutáveis*. Isso significa que elas não podem ser "hasheadas" (transformadas em um código único).

Consequência: Você não conseguirá salvar estados visitados em um `set()` (conjunto) para evitar loops infinitos.

### O Erro Comum:

```
visited = set()
 state = [1, 2, 3]
 visited.add(state)
TypeError: unhashable type: 'list'
```

### A Solução (Tupla Imutável):

```
state = (1, 2, 3)
 visited.add(state) ✔ Sucesso!
```

# Sua Tarefa: A Classe `PuzzleState`

O foco da aula é implementar a lógica de geração d

estados  sucessores.

Note que o estado é uma tupla única de 9 elementos

O índice 0 a 2 é a primeira linha, 3 a 5 a segunda, e

```python
class PuzzleState:
    def __init__(self, board, parent=None, action=None, cost=0):
        self.board = board  # Tupla, ex: (1, 2, 3, 4, 0, 5, 6, 7, 8)
        self.parent = parent
        self.action = action
        self.cost = cost

    def is_goal(self):
        return self.board == (1, 2, 3, 4, 5, 6, 7, 8, 0)

    def get_successors(self):
        successors = []
        zero_idx = self.board.index(0)
        row, col = divmod(zero_idx, 3)
        moves = {'Cima': (-1, 0), 'Baixo': (1, 0), 'Esquerda': (0, -1), 'Direita': (0, 1)}

        #Completar o restante da lógica
```

# Estratégias de Busca Cega

## Busca em Largura (BFS)

Explora camada por camada. Garante o menor caminho, mas a memória explode exponencialmente.

Estrutura: Fila (FIFO)

## Busca em Profundidade (DFS)

Mergulha fundo. Economiza memória, mas pode se perder em caminhos muito longos ou infinitos.

Estrutura: Pilha (LIFO)

# Busca em Largura (BFS) - Implementação

**Busca em Largura (BFS)**

Explora camada por camada. Garante o menor caminho, mas a memória explode exponencialmente.

Estrutura: Fila (FIFO)

```python
def bfs(start_state):
    queue = [start_state]
    explored = set()
    nodes_expanded = 0

    while queue:

        #Completar o restante da lógica
```

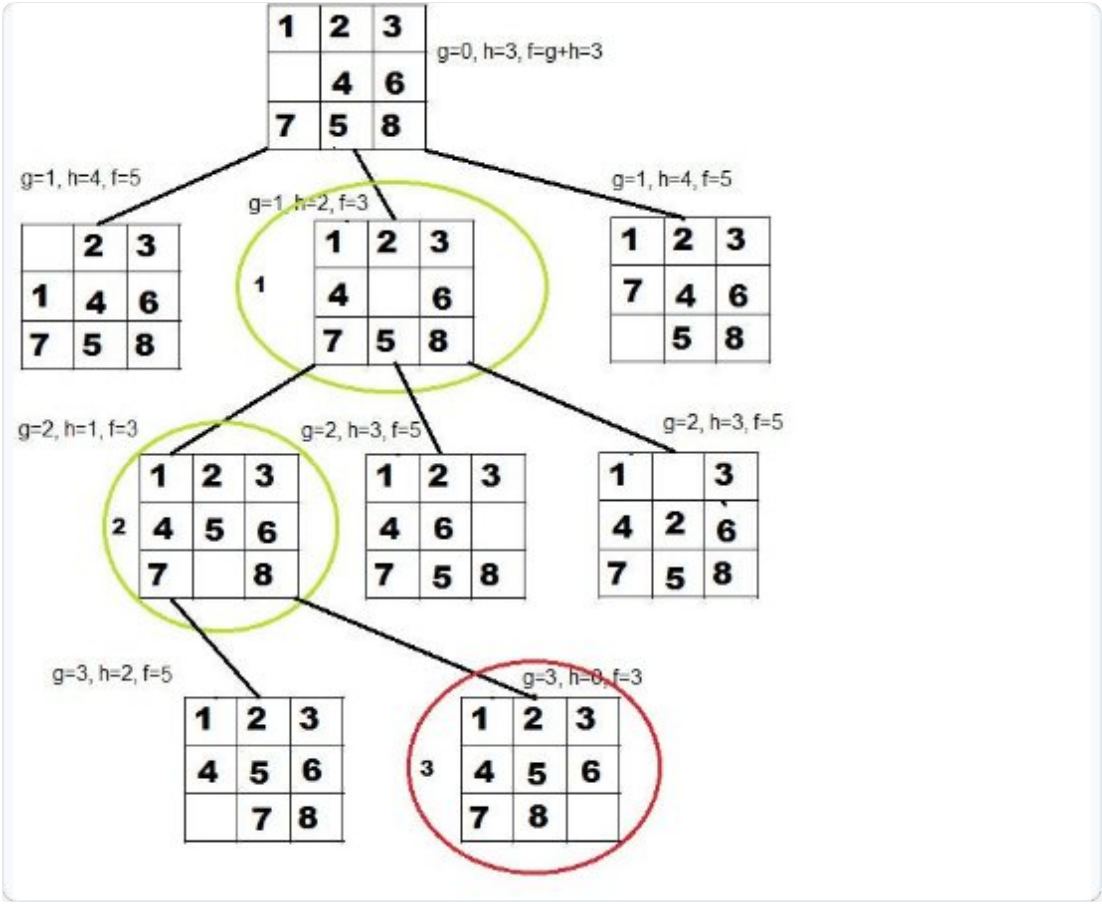
# A Árvore de Busca e o A*

Observe a árvore ao lado. Cada nó é um estado do tabuleiro. A partir da raiz, geramos os filhos movendo o espaço vazio.

O algoritmo A* (A-Estrela) usa a função de custo para decidir qual ramo expandir:

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

Ele evita expandir ramos "caros" ou que se afastam do objetivo.



# A Árvore de Busca e o A*

Observe a árvore ao lado. Cada nó é um estado do tabuleiro. A partir da raiz, geramos os filhos movendo o espaço vazio.

O algoritmo A* (A-Estrela) usa a função de custo para decidir qual ramo expandir:

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

Ele evita expandir ramos "caros" ou que se afastam do objetivo.

```python
def a_star(start_state):
    # Fila guarda tuplas: (custo_f, estado)
    queue = [(manhattan_distance(start_state.board), start_state)]
    #(implementar distância de manhattan)
    explored = set()
    nodes_expanded = 0

    while queue:
        #Completar o restante da lógica
```

# Roteiro do Laboratório

- Passo 1: Complete a classe `PuzzleState`. Implemente o método `get_successors` para gerar os filhos válidos.
- Passo 2: Implemente o `bfs`.
- Passo 3: Implemente o `a_star` com a heurística de Manhattan.
- Análise: Compare o número de nós visitados pelo BFS vs A*.  A diferença deve ser brutal (não sobra nada pro BFS).

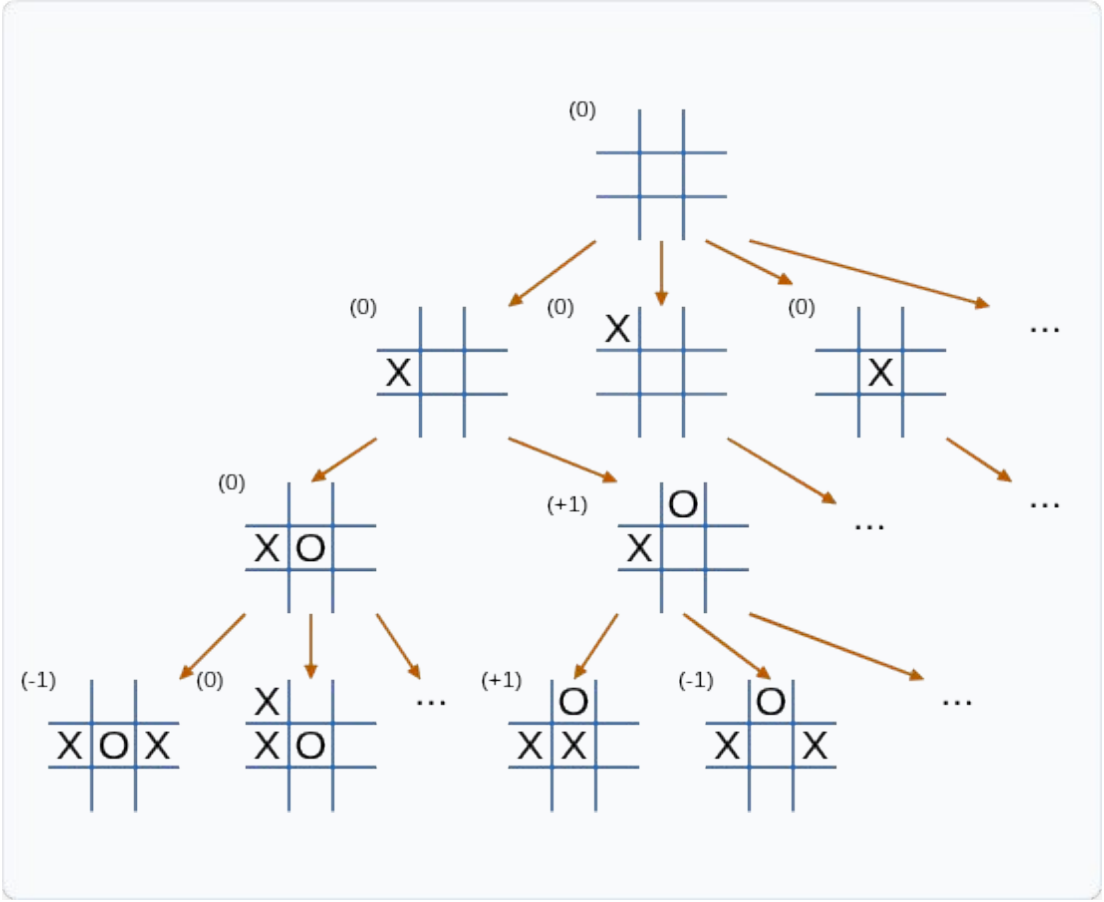
# Bônus: Jogos Adversariais

## Algoritmo Min-Max

Em jogos como Xadrez ou Jogo da Velha, você não está sozinho. Existe um oponente tentando te impedir.

- MAX (Você - X): Tenta maximizar o score (+10).
- MIN (Oponente - O): Tenta minimizar o score (-10).

A árvore ao lado mostra como o valor sobe da folha até a raiz.



# Desafio: Preencha a Lógica

## Implementação do algoritmo

Sua função deve simular todas as jogadas possíveis.

Dica : Lembre-se de desfazer a jogada antes de tentar a próxima!

Use `math.inf` para inicializar a melhor pontuação.

```python
class TicTacToeAI:
    def __init__(self):
        self.board = [' ' for _ in range(9)]
        self.ai = 'X'       # O Maximizador
        self.human = 'O'    # O Minimizador

    def minimax(self, depth, is_maximizing):
        """Algoritmo Minimax com penalidade de profundidade."""
        score = self.check_winner()

        if score is not None:
            if score == 10: return score - depth
            elif score == -10: return score + depth
            else: return 0

        if is_maximizing:
            best_score = -math.inf
            for move in self.get_available_moves():
                #implemente get_available_moves()
                self.board[move] = self.ai
                eval_score = self.minimax(depth + 1, False)
                self.board[move] = ' ' # Backtracking
                best_score = max(eval_score, best_score)
            return best_score
        else:
            #Complete a lógica da implementação
```

# Dicas para o Tic-Tac-Toe Invencível

## Detalhes que fazem a diferença na implementação:

- Sistema de Pontuação: Retorne +10 se a IA ganhar, -10 se o Humano ganhar, e 0 para empate.
- Penalidade de Profundidade: Para a IA preferir ganhar RÁPIDO, subtraia a profundidade da pontuação (ex: `10 - depth`). Isso faz vitórias em menos passos valerem mais.
- Backtracking: É crucial limpar o tabuleiro (`board[i] = ' '`) após a recursão voltar. Se esquecer isso, o tabuleiro ficará cheio de jogadas de teste!