In [95]:
```
import random
import time
import tracemalloc
import copy
from collections import deque
```

BR pt-br: A classe `EightPuzzle` representa a estrutura e a lógica do quebra-cabeça 8-puzzle. Ao instanciar um novo objeto, um estado inicial aleatório é gerado automaticamente por meio do método `_generate_random_state(self)`, que utiliza o método `_is_solvable(self, grid)` para carantir que o estado inicial seja solucionável. O método `move_to(self, position, direction)` movimenta o espaço vazio (representado pelo número 0) na direção especificada. Ele retorna True se o movimento for válido e realizado com sucesso, ou False caso contrário, mantendo o estado anterior inalterado. O método `find_zero(self)` localiza e retorna a posição atual do número 0 na matriz que representa o estado do puzzle.

US en: The `EightPuzzle` class represents the structure and logic of the 8-puzzle game. When a new object is instantiated, an initial random state is automatically generated using the `_generate_random_state(self)` method, which uses the `_is_solvable(self, grid)` method to ensure that the initial state is solvable. The `move_to(self, position, direction)` method moves the blank space (represented by the number 0) in the specified direction. It returns True if the move is valid and successfully executed, or False otherwise, keeping the previous state unchanged. The `find_zero(self)` method locates and returns the current position of the number 0 in the matrix that represents the puzzle state.

In [96]:
```
class EightPuzzle:
    def __init__(self):
        self.state = self._generate_random_state()

    def _generate_random_state(self):
        while True:
            numbers = list(range(9))
            random.shuffle(numbers)
            grid = [numbers[i:i+3] for i in range(0, 9, 3)]
            if self._is_solvable(grid):
                break
        return grid

    def _is_solvable(self, grid):
        flat_list = [num for row in grid for num in row]
        inversions = 0
        for i in range(len(flat_list)):
            for j in range(i + 1, len(flat_list)):
                if flat_list[i] != 0 and flat_list[j] != 0 and flat_list[i] > flat_list[j]:
                    inversions += 1
        return inversions % 2 == 0

    def move_to (self, position, direction):
        row, col = position
        new_row, new_col = row, col

        match direction:
            case 'up':
                new_row = row - 1
            case 'down':
                new_row = row + 1
            case 'left':
                new_col = col - 1
            case 'right':
                new_col = col + 1
            case _:
                print("Invalid direction!")
                return

        if 0 <= new_row < 3 and 0 <= new_col < 3:
            self.state[row][col], self.state[new_row][new_col] = self.state[new_row][new_col], self.state[row][col]
            return True
        else:
            return False

    def find_zero(self):
        for i in range(3):
            for j in range(3):
                if self.state[i][j] == 0:
                    return (i, j)
```

BR pt-br: A classe `BFSPuzzleAgent` representa um agente que resolve o problema do 8-puzzle utilizando o algoritmo de busca em largura (BFS). Ao instanciar um novo objeto, a classe recebe um objeto do tipo `EightPuzzle` representando o estado inicial do puzzle. O objetivo do agente é alcançar o estado final:

(1, 2, 3) (4, 5, 6) (7, 8, 0)

O método `_state_to_tuple(self, state)` converte o estado do puzzle (uma lista de listas) em uma tupla de tuplas, para que possa ser armazenado e verificado em um conjunto de estados visitados. O método `solve(self)` implementa o algoritmo BFS para resolver o puzzle. Ele inicia a busca a partir do estado inicial, explorando todos os movimentos possíveis até encontrar o estado objetivo. O algoritmo utiliza uma fila

(queue) para explorar os estados em ordem de profundidade. Ele retorna um dicionário com informações sobre o caminho até o objetivo, o custo do caminho, o número de nós expandidos, o tamanho da "franja", o uso máximo de memória, entre outras métricas.

us en: The `BFSPuzzleAgent` class represents an agent that solves the 8-puzzle problem using the Breadth-First Search (BFS) algorithm. When a new object is instantiated, the class receives an `EightPuzzle` object representing the initial state of the puzzle. The goal of the agent is to reach the target state:

(1, 2, 3) (4, 5, 6) (7, 8, 0)

The `_state_to_tuple(self, state)` method converts the puzzle state (a list of lists) into a tuple of tuples, so it can be stored and checked in a set of visited states. The `solve(self)` method implements the BFS algorithm to solve the puzzle. It starts the search from the initial state, exploring all possible moves until the goal state is found. The algorithm uses a queue to explore states in breadth-first order. It returns a dictionary with information about the path to the goal, the cost of the path, the number of nodes expanded, the fringe size, the maximum memory usage, and other metrics.

In [97]:
```python
class BFSPuzzleAgent:
    def __init__(self, puzzle):
        self.puzzle = puzzle
        self.goal_state = (
            (1, 2, 3),
            (4, 5, 6),
            (7, 8, 0)
        )
        self.directions = ['up', 'down', 'left', 'right']

    def _state_to_tuple(self, state):
        return tuple(tuple(row)for row in state)

    def solve(self):
        start_time = time.process_time()
        tracemalloc.start()

        visited = set()
        queue = deque()
        queue.append((copy.deepcopy(self.puzzle.state), [], 0))
        visited.add(self._state_to_tuple(self.puzzle.state))

        nodes_expanded = 0
        max_fringe_size = 1
        max_search_depth = 0

        while queue:
            max_fringe_size = max(max_fringe_size, len(queue))
            current_state, path, depth = queue.popleft()

            if self._state_to_tuple(current_state) == self.goal_state:
                end_time = time.process_time()
                current, peak = tracemalloc.get_traced_memory()
                tracemalloc.stop()

                return {
                    'path_to_goal': path,
                    'cost_of_path': len(path),
                    'nodes_expanded': nodes_expanded,
                    'fringe_size': len(queue),
                    'max_fringe_size': max_fringe_size,
                    'search_depth': len(path),
                    'max_search_depth': max_search_depth,
                    'running_time': round(end_time - start_time, 8),
                    'max_ram_usage': round(peak / 1024**2, 8)
                }

            nodes_expanded += 1

            for direction in self.directions:
                temp_puzzle = EightPuzzle()
                temp_puzzle.state = copy.deepcopy(current_state)
                zero_pos = temp_puzzle.find_zero()

                if temp_puzzle.move_to(zero_pos, direction):
                    new_state = temp_puzzle.state
                    state_tuple = self._state_to_tuple(new_state)

                    if state_tuple not in visited:
                        visited.add(state_tuple)
                        queue.append((new_state, path + [direction.capitalize()], depth + 1))
                        max_search_depth = max(max_search_depth, depth + 1)

        return None
```

BR pt-br: A classe `DFSPuzzleAgent` representa um agente que resolve o problema do 8-puzzle utilizando o algoritmo de busca em profundidade (DFS). Ao instanciar um novo objeto, a classe recebe um objeto do tipo `EightPuzzle` representando o estado inicial do puzzle. O objetivo do agente é alcançar o estado final:

(1, 2, 3) (4, 5, 6) (7, 8, 0)

O método `_state_to_tuple(self, state)` converte o estado (uma lista de listas) em uma tupla de tuplas, para que possa ser armazenado e verificado em um conjunto de estados visitados. O método `solve(self, depth_limit=50)` implementa o algoritmo DFS para resolver o puzzle. Ele inicia a busca a partir do estado inicial, explorando todos os movimentos possíveis. A busca ocorre de forma recursiva, com limite de profundidade configurável, até encontrar o estado objetivo ou atingir o limite de profundidade. O algoritmo utiliza uma pilha (stack) para explorar os estados em profundidade. Ele retorna um dicionário com informações sobre o caminho até o objetivo, o custo do caminho, o número de nós expandidos, o tamanho da "franja", o uso máximo de memória, entre outras métricas.

us en: The `DFSPuzzleAgent` class represents an agent that solves the 8-puzzle problem using the Depth-First Search (DFS) algorithm. When a new object is instantiated, the class receives an `EightPuzzle` object representing the initial state of the puzzle. The goal of the agent is to reach the target state:

(1, 2, 3) (4, 5, 6) (7, 8, 0)

The `_state_to_tuple(self, state)` method converts the state (a list of lists) into a tuple of tuples, so it can be stored and checked in a set of visited states.

The `solve(self, depth_limit=50)` method implements the DFS algorithm to solve the puzzle. It starts the search from the initial state, exploring all possible moves. The search is recursive, with a configurable depth limit, until the goal state is found or the depth limit is reached. The algorithm uses a stack to explore states in depth-first order. It returns a dictionary with information about the path to the goal, the cost of the path, the number of nodes expanded, the fringe size, the maximum memory usage, and other metrics.

In [98]:
```python
class DFSPuzzleAgent:
    def __init__(self, puzzle):
        self.puzzle = puzzle
        self.goal_state = (
            (1, 2, 3),
            (4, 5, 6),
            (7, 8, 0)
        )
        self.directions = ['up', 'down', 'left', 'right']

    def _state_to_tuple(self, state):
        return tuple(tuple(row) for row in state)

    def solve(self, depth_limit=50):
        start_time = time.process_time()
        tracemalloc.start()

        visited = set()
        stack = [(copy.deepcopy(self.puzzle.state), [], 0)]
        visited.add(self._state_to_tuple(self.puzzle.state))

        nodes_expanded = 0
        max_fringe_size = 1
        max_search_depth = 0

        while stack:
            max_fringe_size = max(max_fringe_size, len(stack))
            current_state, path, depth = stack.pop()

            if self._state_to_tuple(current_state) == self.goal_state:
                end_time = time.process_time()
                current, peak = tracemalloc.get_traced_memory()
                tracemalloc.stop()

                return {
                    'path_to_goal': path,
                    'cost_of_path': len(path),
                    'nodes_expanded': nodes_expanded,
                    'fringe_size': len(stack),
                    'max_fringe_size': max_fringe_size,
                    'search_depth': len(path),
                    'max_search_depth': max_search_depth,
                    'running_time': round(end_time - start_time, 8),
                    'max_ram_usage': round(peak / 1024**2, 8)
                }

            if depth >= depth_limit:
                continue

            nodes_expanded += 1

            for direction in reversed(self.directions):
                temp_puzzle = EightPuzzle()
                temp_puzzle.state = copy.deepcopy(current_state)
                zero_pos = temp_puzzle.find_zero()

                if temp_puzzle.move_to(zero_pos, direction):
                    new_state = temp_puzzle.state
                    state_tuple = self._state_to_tuple(new_state)
```

```
                    if state_tuple not in visited:
                        visited.add(state_tuple)
                        stack.append((new_state, path + [direction.capitalize()], depth + 1))
                        max_search_depth = max(max_search_depth, depth + 1)

        return None
```

BR pt-br: A classe `IDFSPuzzleAgent` é uma extensão da classe `BFSPuzzleAgent` e implementa o algoritmo de busca em profundidade iterativa (IDFS) para resolver o problema do 8-puzzle. O método `solve(self, max_depth=50)` realiza a busca iterativa, começando com um limite de profundidade de 0 e aumentando progressivamente até o máximo de `max_depth`. Em cada iteração, o agente executa a busca em profundidade (DFS) até atingir o limite de profundidade especificado. O processo é repetido para profundidades maiores, permitindo que o algoritmo explore o estado do puzzle de maneira iterativa. O agente retorna um dicionário com informações sobre o caminho até o objetivo, o custo do caminho, o número de nós expandidos, o tamanho da "franja", o uso máximo de memória, entre outras métricas.

us en: The `IDFSPuzzleAgent` class is an extension of the `BFSPuzzleAgent` class and implements the Iterative Deepening Depth-First Search (IDFS) algorithm to solve the 8-puzzle problem. The `solve(self, max_depth=50)` method performs the iterative deepening search, starting with a depth limit of 0 and progressively increasing until the maximum max_depth is reached. In each iteration, the agent performs a depth-first search (DFS) up to the specified depth limit. This process is repeated for increasing depths, allowing the algorithm to explore the puzzle state iteratively. The agent returns a dictionary with information about the path to the goal, the cost of the path, the number of nodes expanded, the fringe size, the maximum memory usage, and other metrics.

In [ ]:
```
class IDFSPuzzleAgent(BFSPuzzleAgent):
    def solve(self, max_depth=50):
        start_time = time.process_time()
        tracemalloc.start()

        nodes_expanded = 0
        max_fringe_size = 0
        max_search_depth = 0

        for depth_limit in range(max_depth + 1):
            visited = set()
            stack = [(copy.deepcopy(self.puzzle.state), [], 0)]
            visited.add(self._state_to_tuple(self.puzzle.state))

            while stack:
                max_fringe_size = max(max_fringe_size, len(stack))
                current_state, path, depth = stack.pop()

                if self._state_to_tuple(current_state) == self.goal_state:
                    end_time = time.process_time()
                    current, peak = tracemalloc.get_traced_memory()
                    tracemalloc.stop()

                    return {
                        'path_to_goal': path,
                        'cost_of_path': len(path),
                        'nodes_expanded': nodes_expanded,
                        'fringe_size': len(stack),
                        'max_fringe_size': max_fringe_size,
                        'search_depth': len(path),
                        'max_search_depth': max_search_depth,
                        'running_time': round(end_time - start_time, 8),
                        'max_ram_usage': round(peak / 1024**2, 8)
                    }

                if depth >= depth_limit:
                    continue

                nodes_expanded += 1

                for direction in reversed(self.directions):
                    temp_puzzle = EightPuzzle()
                    temp_puzzle.state = copy.deepcopy(current_state)
                    zero_pos = temp_puzzle.find_zero()

                    if temp_puzzle.move_to(zero_pos, direction):
                        new_state = temp_puzzle.state
                        state_tuple = self._state_to_tuple(new_state)
                        if state_tuple not in visited:
                            visited.add(state_tuple)
                            stack.append((new_state, path + [direction.capitalize()], depth + 1))
                            max_search_depth = max(max_search_depth, depth + 1)

        return None
```

In [100…
```
puzzle = EightPuzzle()

print("Initial state:")
for row in puzzle.state:
    print(row)

bfs_agent = BFSPuzzleAgent(puzzle)
```

```python
bfs_result = bfs_agent.solve()
print("\n--- BFS Result ---")
if bfs_result:
    for key, value in bfs_result.items():
        print(f"{key}: {value}")
else:
    print("No solution found.")

dfs_agent = DFSPuzzleAgent(puzzle)
dfs_result = dfs_agent.solve(depth_limit=50)
print("\n--- DFS Result ---")
if dfs_result:
    for key, value in dfs_result.items():
        print(f"{key}: {value}")
else:
    print("No solution found.")

idfs_agent = IDFSPuzzleAgent(puzzle)
idfs_result = idfs_agent.solve(max_depth=50)
print("\n--- IDFS Result ---")
if idfs_result:
    for key, value in idfs_result.items():
        print(f"{key}: {value}")
else:
    print("No solution found.")
```

```
Initial state:
[8, 0, 5]
[7, 6, 4]
[3, 2, 1]

--- BFS Result ---
path_to_goal: ['Right', 'Down', 'Down', 'Left', 'Left', 'Up', 'Up', 'Right', 'Right', 'Down', 'Down', 'Left', 'Up', 'Left', 'Up',
'Right', 'Right', 'Down', 'Down', 'Left', 'Up', 'Left', 'Up', 'Right', 'Right', 'Down', 'Down']
cost_of_path: 27
nodes_expanded: 178031
fringe_size: 2348
max_fringe_size: 25133
search_depth: 27
max_search_depth: 28
running_time: 68.328125
max_ram_usage: 78.79300022

--- DFS Result ---
path_to_goal: ['Down', 'Down', 'Left', 'Up', 'Up', 'Right', 'Down', 'Down', 'Left', 'Up', 'Up', 'Right', 'Down', 'Down', 'Left', 'U
p', 'Up', 'Right', 'Down', 'Down', 'Right', 'Up', 'Left', 'Down', 'Left', 'Up', 'Up', 'Right', 'Down', 'Down', 'Right', 'Up', 'Lef
t', 'Left', 'Up', 'Right', 'Down', 'Down', 'Left', 'Up', 'Right', 'Down', 'Right', 'Up', 'Up', 'Left', 'Down', 'Right', 'Down']
cost_of_path: 49
nodes_expanded: 19264
fringe_size: 33
max_fringe_size: 43
search_depth: 49
max_search_depth: 50
running_time: 6.84375
max_ram_usage: 9.15756035

--- IDFS Result ---
path_to_goal: ['Down', 'Right', 'Up', 'Left', 'Left', 'Down', 'Down', 'Right', 'Up', 'Up', 'Left', 'Down', 'Down', 'Right', 'Righ
t', 'Up', 'Left', 'Up', 'Left', 'Down', 'Down', 'Right', 'Up', 'Left', 'Down', 'Right', 'Right', 'Up', 'Up', 'Left', 'Left', 'Dow
n', 'Right', 'Down', 'Right']
cost_of_path: 35
nodes_expanded: 456567
fringe_size: 13
max_fringe_size: 34
search_depth: 35
max_search_depth: 37
running_time: 167.0
max_ram_usage: 27.75058842
```

Conclusão:

Os resultados fazem sentido e são coerentes com as características de cada algoritmo.

- BFS: Sempre encontra o caminho mais curto (menor custo de caminho). No entanto, ela consome muita memória e tempo, especialmente em problemas como o 8-puzzle, devido à explosão combinatória da árvore de estados. O uso de RAM e o número de nós expandidos são muito altos, como esperado.
- DFS: É mais rápida e usa menos memória, mas não garante o caminho mais curto.É comum encontrar soluções longas e pouco otimizadas, o que aparece claramente nesse resultado.
- IDFS: Combina o melhor da BFS e DFS: encontra o menor caminho (ou próximo) com menos uso de memória que a BFS.O tempo de execução pode ser alto porque ele repete várias vezes a DFS com profundidade crescente. É esperado que o número de nós expandidos seja alto e o tempo também.