



```
In [1]: import random
import time
import tracemalloc
import copy
import concurrent.futures
from collections import deque
```

 **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 garantir 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.

 **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 [2]: 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]
            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)

```

 **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.

 **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 [3]:

```

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

```

 **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.

 **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 [4]: 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
```

 **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.

 **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 [5]: 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

```

🇧🇷 **pt-br:** A função `run_parallel()` executa os algoritmos de busca em largura (BFS), profundidade (DFS) e profundidade iterativa (IDFS) em paralelo, usando um `ThreadPoolExecutor` para paralelizar a execução. Essa função recebe um `puzzle` como entrada, imprime o estado inicial do quebra-cabeça, e então cria três tarefas paralelas que resolvem o problema com os algoritmos BFS, DFS (com limite de profundidade) e IDFS (com profundidade máxima). Ao final, ela imprime os resultados de cada algoritmo, incluindo o caminho até o objetivo, custo do caminho, número de nós expandidos, profundidade alcançada, tempo de execução e uso de memória.

🇺🇸 **en:** The `run_parallel()` function runs the Breadth-First Search (BFS), Depth-First Search (DFS), and Iterative Deepening DFS (IDFS) algorithms in parallel using a `ThreadPoolExecutor` to parallelize the execution. This function takes a `puzzle` as input, prints the initial state of the puzzle, and then creates three parallel tasks that solve the problem using BFS, DFS (with a depth limit), and IDFS (with a maximum depth). At the end, it prints the results of each algorithm, including the path to the goal, path cost, number of nodes expanded, depth reached, runtime, and memory usage.

```

In [6]: DEPTH_LIMIT = 200
        MAX_DEPTH = 50

        def run_agent(agent_class, puzzle, **kwargs):
            agent = agent_class(copy.deepcopy(puzzle))
            return agent.solve(**kwargs)

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

            start_time = time.time()

            with concurrent.futures.ThreadPoolExecutor() as executor:
                futures = {
                    'BFS': executor.submit(run_agent, BFSPuzzleAgent, puzzle),
                    'DFS': executor.submit(run_agent, DFSPuzzleAgent, puzzle, depth_limit=DEPTH_LIMIT),
                    'IDFS': executor.submit(run_agent, IDFSPuzzleAgent, puzzle, max_depth=MAX_DEPTH),
                }

                results = {}
                for name, future in futures.items():
                    results[name] = future.result()

            end_time = time.time()

            for name, result in results.items():
                print(f"\n--- {name} Result ---")
                if result:
                    for key, value in result.items():
                        print(f"{key}: {value}")
                else:
                    print("No solution found.")

            print(f"\nTotal execution time (parallel): {round(end_time - start_time, 4)}s")

```

```

In [7]: puzzle = EightPuzzle()
        run_parallel(puzzle)

```

[8, 6, 3]

```
max_ram_usage: 50.30559635
```

```
max_ram_usage: 0.0
```

```
max ram usage: 0.0
```

 **pt-br:**

Encontrou o caminho de menor custo (20), o que confirma sua principal vantagem: garantir a solução ótima em termos de profundidade. Por outro lado, o algoritmo apresentou um alto consumo de memória

e tempo de execução, devido à grande quantidade de nós expandidos (mais de 42 mil). Esses valores são compatíveis com o que se espera da BFS, especialmente em problemas com espaço de estados muito grande.

- **DFS (Busca em Profundidade):**

Embora tenha encontrado uma solução, o caminho foi bastante longo (custo 184), evidenciando a limitação da DFS em encontrar soluções ótimas. Apesar de utilizar menos memória, o número de nós expandidos foi ainda maior que na BFS, o que mostra que, mesmo sendo mais econômica em RAM, a DFS pode se perder em caminhos profundos e pouco eficientes.

- **IDFS (Busca em Profundidade Iterativa):**

Apresentou um bom equilíbrio entre custo do caminho (24) e uso de recursos. A IDFS conseguiu encontrar uma solução próxima da ótima, utilizando menos memória que a BFS. O tempo de execução foi relativamente alto, o que é esperado, já que o algoritmo executa múltiplas buscas em profundidade com profundidades progressivas. Ainda assim, seu desempenho foi bastante satisfatório e alinhado com as características teóricas do algoritmo.

 en:

Conclusion:

The results align well with the expected behavior of each algorithm when applied to the 8-puzzle problem:

- **BFS (Breadth-First Search):**

Successfully found the shortest path (20), which confirms its strength in providing optimal solutions in terms of depth. However, it required a high amount of memory and execution time due to the large number of expanded nodes (over 42,000). These values are consistent with what is typically expected from BFS, especially in problems with large state spaces.

- **DFS (Depth-First Search):**

Although it found a solution, the path was quite long (cost of 184), highlighting DFS's tendency to return suboptimal solutions. While it uses less memory, it ended up expanding even more nodes than BFS, showing that despite being memory-efficient, DFS can get lost in deep and inefficient paths.

- **IDFS (Iterative Deepening DFS):**

Demonstrated a good balance between path cost (24) and resource usage. It managed to find a solution close to optimal while using significantly less memory than BFS. The execution time was relatively high, which is expected since it repeatedly performs DFS with increasing depth