TUGAS 2
KECERDASAN BUATAN - F

SOLVING 8-PUZZLE

Using Informed Search Methods

Manhatan Distance and Hamming Distance. (Misplaced Tiles)

GET TO KNOW US

ONG MING SEN AI

Arif Nugraha Santosa
5025211048

Laurivasya Gadhing Syahafidh
5025211136

Rayhan Arvianta Bayuputra
5025211217

INFORMED SEARCH

Informed Search Algorithm, disebut juga dengan Heuristic Search. Untuk menyelesaikan problem 8-Puzzle kami menggunakan metode informed search algoritma A* (A-star). Algoritma A* merupakan metode informed search yang menggabungkan strategi best-first search dan 2 fungsi heuristik (Manhattan Distance dan Misplaced Tiles).

HEURISTIC 1 : MANHATTAN DISTANCE

Heuristik 1 adalah fungsi heuristik yang mengukur jarak dari setiap angka di dalam state awal 8 puzzle ke lokasi tujuan pada goal state untuk setiap node. Dalam heuristik ini, untuk setiap angka pada puzzle, jarak horizontal (kanan dan kiri) dan vertikal (atas dan bawah) dari state awal ke goal state akan dihitung, kemudian jarak-jarak ini dijumlahkan untuk memberikan nilai heuristik. Heuristik ini dapat dinyatakan sebagai berikut:

$h1(n) = \Sigma |xi - yi|$

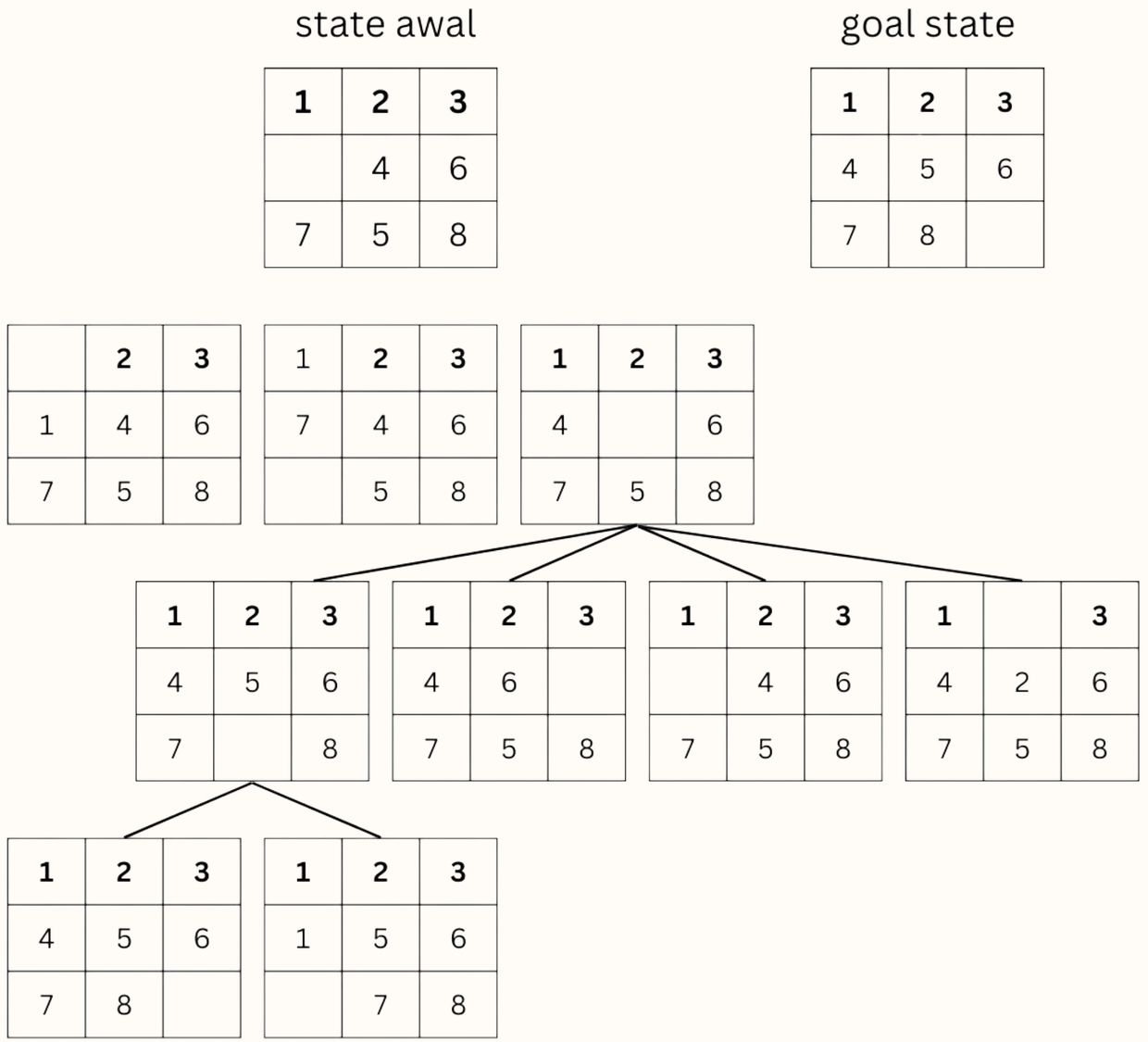
n: state awal 8 puzzle
xi: koordinat horizontal angka i dalam state awal
yi: koordinat vertikal angka i dalam state awal

HEURISTIC 2 : MISPLACED TILES

Jumlah Angka Salah (Misplaced Tiles) Heuristik 2 adalah fungsi heuristik yang menghitung jumlah angka yang tidak berada pada posisi yang benar pada puzzle saat ini dibandingkan dengan posisi yang benar pada goal state. Dalam heuristik ini, untuk setiap angka pada puzzle, dicari apakah posisinya di keadaan saat ini sama dengan posisi yang seharusnya pada goal state. Jika posisi tidak cocok, maka angka tersebut dihitung sebagai salah satu angka. Heuristik 2 dapat dinyatakan sebagai berikut:

h2(n) = jumlah angka yang salah posisi di keadaan saat ini

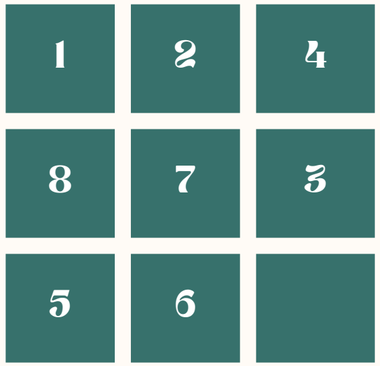
ilustrasi heuristik



UNSOLVED PUZZLE

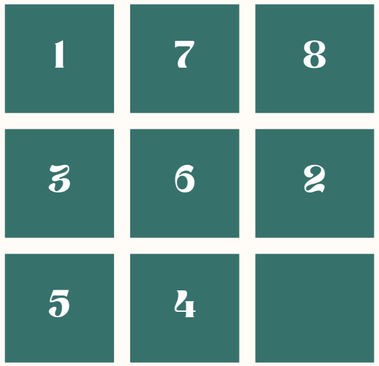
- Sebelum dilakukan pencarian, pastikan state awal puzzle dapat ditemukan solusinya, jika tidak maka akan terjadi infinite loops
- Menghitung inversi untuk setiap node (jumlah node lain disebelah kanan yang lebih kecil)
- Jika nilai inversi ganjil maka 8-puzzle unsolvable, jika nilai inversi genap maka 8-puzzle solvable

contoh :



CONTOH SOLVABLE 8-PUZZLE

1 2 4 8 7 3 5 6
1 -> 0
2 -> 0
4 -> 1 (3)
8 -> 4 (7,3,5,6)
7 -> 3 (3,5,6)
3 -> 0
5 -> 0
6 -> 0
JUMLAH: 8 (GENAP)



CONTOH UNSOLVABLE 8-PUZZLE

1 7 8 3 6 2 5 4
1 -> 0
7 -> 5 (3,6,2,5,4)
8 -> 5 (3,6,2,5,4)
3 -> 1 (2)
6 -> 3 (2,5,4)
2-> 0
5-> 1 (4)
4-> 0
JUMLAH: 15 (GANJIL)

```python
"""
A program to solve the 8-puzzle problem using the A* algorithm and either the
Manhattan Distance heuristic or the Hamming Distance heuristic.
"""
from collections import defaultdict
from copy import deepcopy

import numpy as np


def choose_heuristic() -> str:
    """
    Gets the user to select either the Manhattan or Hamming Distance heuristic.

    Returns:
        The type of heuristic to use for solving the 8-puzzle problem.
    """
    heuristic_input = ""
    while heuristic_input not in ("m", "h"):
        heuristic_input = input(
            "\nThe Manhattan Distance heuristic is based on number of squares "
            "between itself and the goal position, whereas the Hamming "
            "Distance heuristic is based on total number of misplaced tiles.\n"
            "Would you like to use the Manhattan Distance heuristic (m) or "
            "the Hamming Distance heuristic (h)?\n").lower()
        if heuristic_input not in ("m", "h"):
            print("Invalid input! Please enter 'm' for the Manhattan Distance "
                  "heuristic, or 'h' for the Hamming Distance heuristic.")
        elif heuristic_input == "m":
            return "Manhattan"
        else:
            return "Hamming"
```

choose_heuristic(): Fungsi ini meminta pengguna untuk memilih jenis heuristic yang akan digunakan untuk menyelesaikan masalah 8-puzzle. Kemudian, ia akan mengembalikan jenis heuristic yang dipilih.

```python
def choose_states():
    """
    Creates the start and goal states for the 8-puzzle problem.

    Returns:
        The start state and desired goal state of the board.
    """
    start = np.array([7, 2, 4, 5, 0, 6, 8, 3, 1])
    goal = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8])

    # start = np.array([1, 2, 3, 5, 0, 6, 4, 7, 8])
    # goal = np.array([1, 2, 3, 4, 5, 6, 7, 8, 0])

    return start, goal
```

```python
def is_solvable(start) -> bool:
    """
    Checks whether the 8-puzzle problem is solvable based on inversions.

    Args:
        start: The start state of the board input by the user.

    Returns:
        Whether the 8-puzzle problem is solvable.
    """
    k = start[start != 0]
    num_inversions = sum(
        len(np.array(np.where(k[i + 1:] < k[i])).reshape(-1)) for i in
        range(len(k) - 1))
    return num_inversions % 2 == 0
```

choose_states(): Fungsi ini membuat kondisi awal dan akhir papan 8-puzzle. Fungsi ini mengembalikan kondisi awal dan akhir.

is_solvable(start): Fungsi ini memeriksa apakah masalah 8-puzzle dapat diselesaikan berdasarkan inversions. Fungsi ini mengembalikan apakah masalah dapat diselesaikan atau tidak.

```python
def assign_coordinates(board):
    """
    Assigns coordinates to each digit to calculate the Manhattan Distance.

    Args:
        board: The state of the board.

    Returns:
        Coordinates to calculate the Manhattan Distance.
    """
    coordinate = np.array(range(9))
    # Gets the index of each value, where the board is the array of values.
    for x, y in enumerate(board):
        coordinate[y] = x
    return coordinate
```

```python
def calculate_heuristic(heuristic, current, goal) -> int:
    """
    Calculates h(n) using either the Manhattan or Hamming Distance heuristic.

    Args:
        heuristic: The heuristic algorithm chosen by the user.
        current: The current state of the board.
        goal: The desired state of the board.

    Returns:
        The value of h(n) according to the heuristic algorithm used.
    """
    if heuristic == "Manhattan":
        current_c = assign_coordinates(current)
        goal_c = assign_coordinates(goal)
        result = manhattan_heuristic(current_c, goal_c)
    else:
        result = hamming_heuristic(current, goal)
    return result
```

assign_coordinates(board):
Fungsi ini menetapkan koordinat ke setiap digit untuk menghitung jarak Manhattan. Fungsi ini mengembalikan koordinat untuk menghitung jarak Manhattan.

calculate_heuristic(heuristic, current, goal): Fungsi ini menghitung nilai h(n) menggunakan heuristic jarak Manhattan atau Hamming Distance. Fungsi ini mengembalikan nilai h(n) sesuai dengan jenis heuristic yang dipilih.

```python
def manhattan_heuristic(current, goal) -> int:
    """
    Performs the Manhattan Distance heuristic.

    Args:
        current: The current state of the board.
        goal: The desired state of the board.

    Returns:
        Minimum number of moves to reach the goal state from the current state.
    """
    manhattan = abs(current // 3 - goal // 3) + abs(current % 3 - goal % 3)
    return sum(manhattan[1:])


def hamming_heuristic(current, goal) -> int:
    """
    Performs the Hamming Distance heuristic.

    Args:
        current: The current state of the board.
        goal: The desired state of the board.

    Returns:
        Number of misplaced tiles on the current state of the board.
    """
    return np.sum(current != goal) - 1
```

manhattan_heuristic(current, goal): Fungsi ini melakukan heuristic jarak Manhattan. Fungsi ini mengembalikan jumlah minimum langkah untuk mencapai kondisi akhir dari kondisi saat ini.

hamming_heuristic(current, goal): Fungsi ini melakukan heuristic Hamming Distance. Fungsi ini mengembalikan jumlah kotak yang salah di kondisi saat ini.

```python
def search(heuristic, start, goal):
    (goal, moves, previous_boards, priority, priority_queue_type,
     state, state_type) = setup_search(heuristic, start, goal)

    # Searches until the goal state is found.
    while True:
        # Sorts the priority queue according to the total cost.
        priority = np.sort(priority, kind="mergesort",
                           order=["f_function", "position"])
        # Explores the first node from the priority queue, and removes it from
        # the priority queue.
        position = priority[0][0]
        priority = np.delete(priority, 0, 0)
        board = state[position][0]
        g_function = state[position][2] + 1
        location = int(np.where(board == 0)[0])

        # Checks all possible moves which can be made from the current state.
        for move in moves:
            # Performs the move if it is valid.
            if location not in move["position"]:
                # Copies the current state of the board.
                new_state = deepcopy(board)
                # Performs the move.
                delta_location = location + move["delta"]
                new_state[location], new_state[delta_location] = \
                    new_state[delta_location], new_state[location]

                # Ensures that boards aren't repeatedly processed.
                if previous_boards[tuple(new_state)]:
                    continue
                previous_boards[tuple(new_state)] = True

                # Calculates the estimated cost of reaching the goal state from
                # the current state of the board using the chosen heuristic.
                h_function = calculate_heuristic(heuristic, new_state, goal)
                # Generates and adds the new board state.
                new_state_details = np.array(
                    [(new_state, position, g_function, h_function)],
                    dtype=state_type)
                state = np.append(state, new_state_details, 0)
                # Calculates the total cost, and adds the new state to the
                # priority queue.
                f_function = g_function + h_function
                new_state_details = np.array([(len(state) - 1, f_function)],
                                             dtype=priority_queue_type)
                priority = np.append(priority, new_state_details, 0)

                # Stops the search if the goal state has been achieved.
                if np.array_equal(new_state, goal):
                    print("\nGoal state has been achieved with the "
                          "following steps:\n")
                    return state, len(priority)
```

search(heuristic, start, goal):
Fungsi ini melakukan pencarian A* pada masalah 8-puzzle. Fungsi ini mengembalikan keadaan papan dan berapa banyak keadaan yang dieksplorasi. Fungsi ini juga memeriksa apakah masalah 8-puzzle dapat diselesaikan atau tidak menggunakan fungsi is_solvable(). Kemudian, fungsi ini menyiapkan keadaan awal dan akhir serta menambahkannya ke dalam antrian prioritas. Fungsi kemudian mencari keadaan yang meminimalkan f(n) dengan menjelajahi antrian prioritas. Fungsi ini juga menghitung nilai g(n) dan h(n) untuk setiap keadaan dan menambahkannya ke dalam antrian prioritas. Fungsi akan terus mencari sampai kondisi akhir ditemukan. Setelah itu, fungsi akan mengembalikan daftar kondisi papan dan jumlah keadaan yang dieksplorasi.

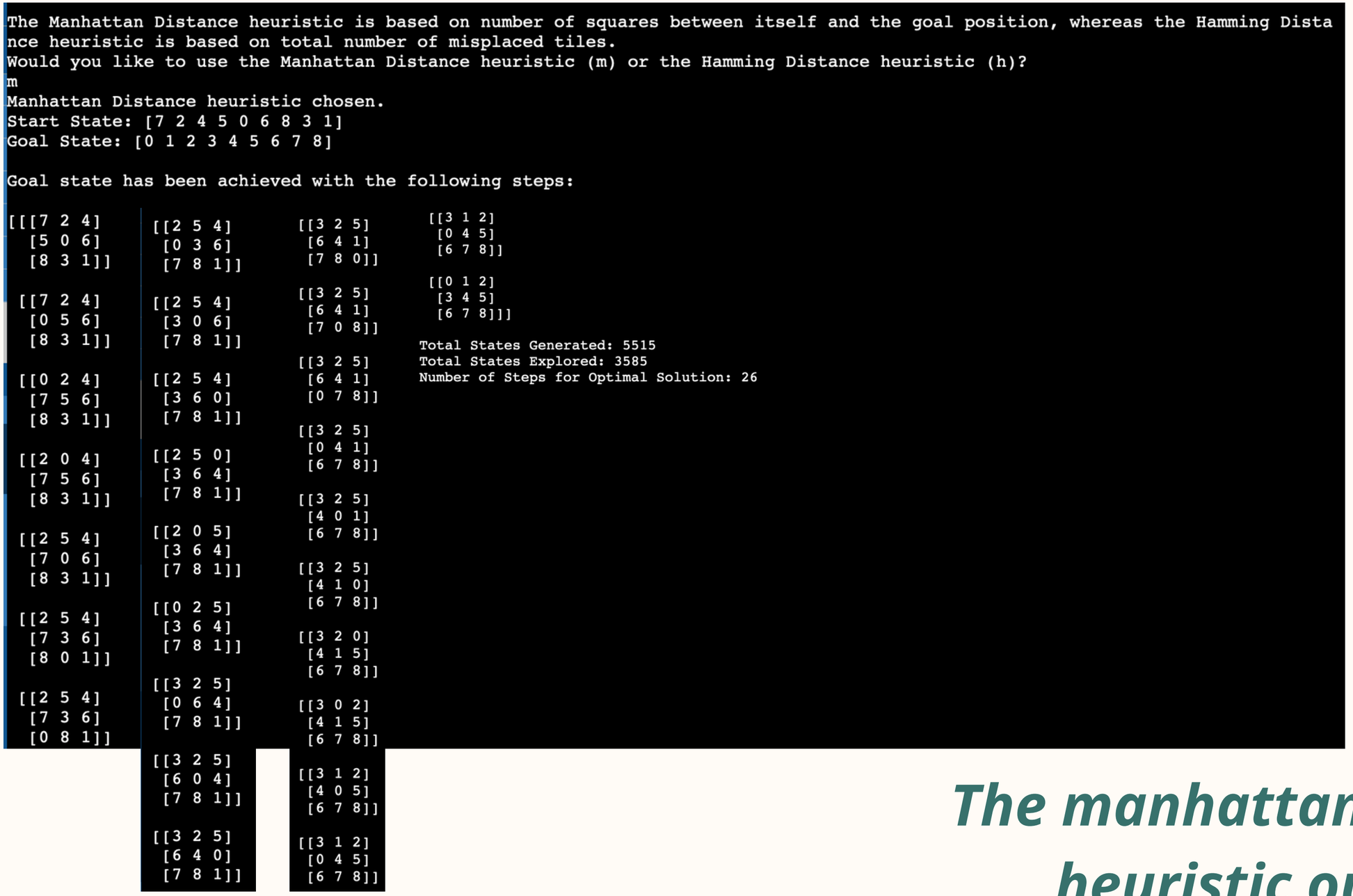
```python
def setup_search(heuristic, start, goal):
    """
    Sets up the valid moves, priority queue, and state tracking for the search.

    Args:
        heuristic: Algorithm used to estimate the cost of reaching the goal.
        start: The start state of the board input by the user.
        goal: The desired state of the board.

    Returns:
        A record of valid moves, priority queue, and states.
    """
    # Sets the rules for the moves a tile can make, and when it can do them.
    moves = np.array([("up", [0, 1, 2], -3),
                      ("down", [6, 7, 8], 3),
                      ("left", [0, 3, 6], -1),
                      ("right", [2, 5, 8], 1)],
                     dtype=[("move", str, 1),
                            ("position", list),
                            ("delta", int)])
    # Creates the data structures for board state and priority queue.
    state_type = [("board", list),
                  ("parent", int),
                  ("g_function", int),
                  ("h_function", int)]
    priority_queue_type = [("position", int),
                           ("f_function", int)]
    # Creates a dictionary to keep track of boards which have been processed.
    previous_boards = defaultdict(bool)
    # Processes the start position of the board.
    h_function = calculate_heuristic(heuristic, start, goal)
    state = np.array([(start, -1, 0, h_function)],
                     dtype=state_type)
    priority = np.array([(0, h_function)], dtype=priority_queue_type)

    return (goal, moves, previous_boards, priority, priority_queue_type,
            state, state_type)
```

setup_search(heuristic, start, goal):
Fungsi ini digunakan untuk menyiapkan konfigurasi untuk algoritma A*. Fungsi akan mengembalikan record yang berisi move yang valid, priority queue, dan states.

```python
def generate_steps(state):
    """
    Generates the step-by-step solution to reach the goal state.

    Args:
        state: A record of the state of each 3x3 board.

    Returns:
        The state of the 3x3 board after each step to the goal state.
    """
    # Creates an array of integers to display each board state.
    optimal = np.array([], int)
    last = len(state) - 1
    while last != -1:
        optimal = np.insert(optimal, 0, state[last]["board"])
        last = int(state[last]["parent"])
    return optimal.reshape(-1, 3, 3)


def main():
    """
    Runs the A* algorithm to solve the 8-puzzle problem.
    """
    heuristic = choose_heuristic()
    start, goal = choose_states()
    print(("{} Distance heuristic chosen.\nStart State: {}"
           "\nGoal State: {}").format(heuristic, start, goal))

    # Stops the program if the 8-puzzle is unsolvable.
    if is_solvable(start) is False:
        print("\nThe 8-puzzle problem is unsolvable with this start state!")
        return

    state, explored = search(heuristic, start, goal)
    optimal = generate_steps(state)
    print(("{}\n\nTotal States Generated: {}\nTotal States Explored: {}"
           "\nNumber of Steps for Optimal Solution: {}").format(
        optimal, len(state), len(state) - explored, len(optimal) - 1))


if __name__ == "__main__":
    main()
```

generate_steps(state):
Fungsi ini digunakan untuk menghasilkan langkah-langkah solusi dari state awal ke goal state. Fungsi akan mengembalikan state dari papan 3x3 setelah setiap langkah ke goal state.
main():
Fungsi utama yang digunakan untuk menjalankan algoritma A*. Fungsi ini memanggil fungsi choose_heuristic() dan choose_states() untuk memilih jenis heuristic dan state awal dan goal. Kemudian fungsi akan memeriksa apakah state awal dapat diselesaikan dengan memanggil fungsi is_solvable(). Jika state awal tidak dapat diselesaikan, maka program akan dihentikan. Jika dapat diselesaikan, maka fungsi search() akan dipanggil untuk menyelesaikan permasalahan. Setelah itu, fungsi generate_steps() dipanggil untuk menghasilkan solusi dari state awal ke goal state dan mencetak hasil ke layar.

choose_heuristic(): Fungsi ini digunakan untuk memilih jenis heuristic yang akan digunakan oleh algoritma A*.
choose_states(): Fungsi ini digunakan untuk memilih state awal dan goal.
is_solvable(state): Fungsi ini digunakan untuk memeriksa apakah suatu state awal dapat diselesaikan atau tidak.
search(heuristic, start, goal): Fungsi ini digunakan untuk menyelesaikan permasalahan dengan menggunakan algoritma A*. Fungsi akan mengembalikan record yang berisi state dari setiap papan 3x3, dan jumlah state yang dieksplorasi.



The manhattan distance heuristic output