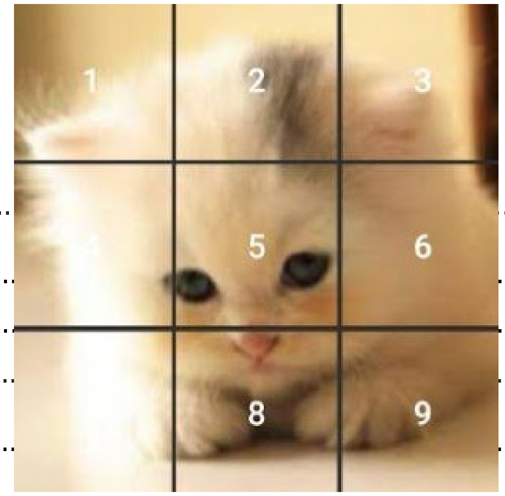
Eight Puzzle

Using Informed and Uninformed Search Algorithms

Nouran Bakry 3679
Shahenda ElSayed 3695
Dina Abdalah Aboutahoun 3521

Table of Contents



Introduction

The puzzle consists of an area divided into a grid, 3 by 3 for the 8-puzzle. On each grid square is a tile, expect for one square which remains empty. Thus, there are eight tiles in the 8-puzzle. A tile that is next to the empty grid square can be moved into the empty space, leaving its previous position empty in turn. Tiles are numbered, 1 thru 8 for the 8-puzzle, so that each tile can be uniquely identified.

The aim of the puzzle is to achieve an ordered configuration of tiles from a given (different) configuration by sliding the individual tiles around the grid as described above.

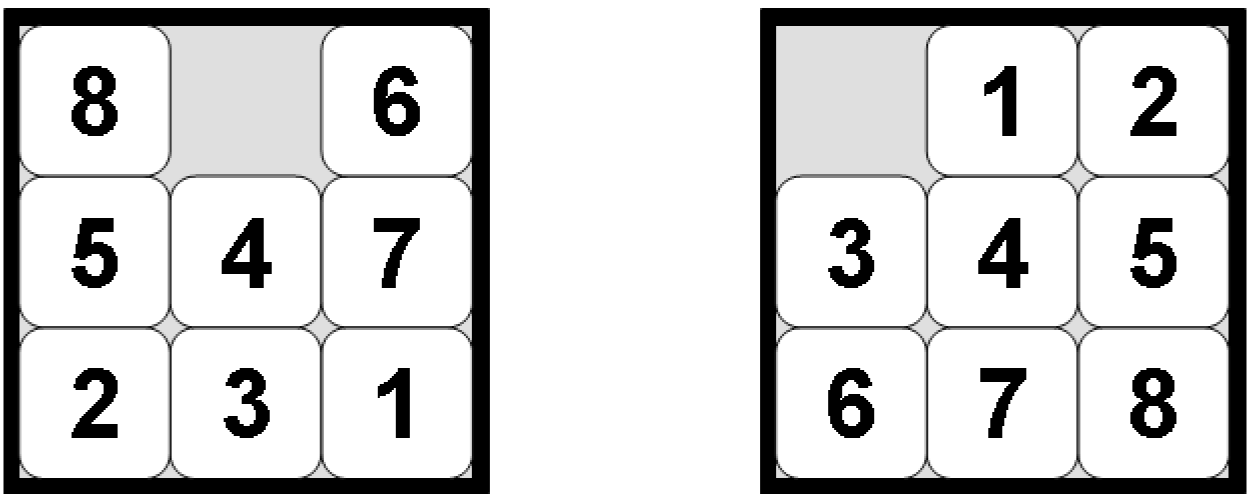
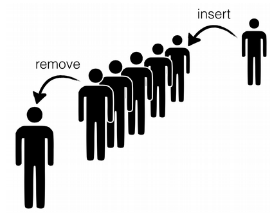


Figure 1: Configurations of the 8-Puzzle: worst-case initial state (left) for goal state (right)

Uninformed Search



Data structures used:

- To implement BFS we use a *queue* to simulate horizontal expansion of nodes, to explore the breadth of a vertex depth before moving on. This behavior guarantees that the first path located is one of the shortest-paths present.

- To implement DFS we use a *stack* to simulate depth traversal, to explore possible vertices (from a supplied root) down each branch before backtracking. Below is a listing of the actions performed upon each visit to a node:
    - Mark the current vertex as being visited.
    - Explore each adjacent vertex that is not included in the visited set.

Algorithms used:

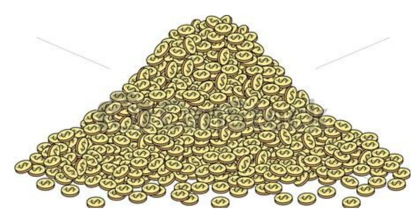
- Breadth-First Search

Starts at the tree root and explores the neighbor nodes first, before moving to the next level neighbors.

- Depth-First Search

Starts at the root and explores as far as possible along each branch before backtracking.

Informed Search



- To implement A*, at each step of the algorithm, the node with the lowest $f(x)$ value is removed from the *heap*, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the heap. The algorithm continues until a goal node has a lower f value than any node in the heap (or until the heap is empty).The f value of the goal is then the length of the shortest path, since h at the goal is zero in an admissible heuristic.

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

where n is the last node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic that estimates the cost of the cheapest path from n to the goal. The heuristic is problem-specific.

Highlights of the code

The generation of all the possible scenarios (either up, down, left or right).

```
def generate_neighbours(self):  # creates a list of neighbouring states

    actions = ((0, 1, 2, -3), (6, 7, 8, 3), (0, 3, 6, -1), (2, 5, 8, 1))  # up,down,left and right
    neighbours = []
    empty_position = self.value.index(0)

    for position in actions:
        if empty_position not in position[:3]:
            temp_list = self.value[:]
            temp_list[empty_position + position[3]], temp_list[empty_position] = temp_list[empty_position], temp_list[empty_position +
            neighbours.append(State(temp_list, self))

    return neighbours
```

Definition of the root (The node that has no parent), this is the initial state of the board.

```
class State:

    def __init__(self, initial_values, parent=None):
        self.value = initial_values
        self.parent = parent
        if self.parent:
            self.depth = self.parent.depth + 1
        else:
            self.depth = 0
```

The tracing of the path, prints all the movements taken to reach the goal.

```
#   This function traces the moves made along the path to reach the goal state.
def trace_path(state):
    path = []
    while state.parent:
        child = state
        state = state.parent
        child_index = child.value.index(0)
        parent_index = state.value.index(0)
        if child_index is not parent_index:
            difference = child_index - parent_index
            if difference == -3:
                path.append('up')
            elif difference == 3:
                path.append('down')
            elif difference == -1:
                path.append('left')
            else:
                path.append('right')
    path.reverse()
    print('path:', path)
```

The class *solver* performs all the search operations. BFS, DFS and A* with both Manhattan's and Euclidean`s heuristic.

```
class Solver:

    def __init__(self, state):
        self.state = state
        self.puzzle = self.state.value

    def bfs(self):
        queue = deque()
        frontier = set()
        queue.append(self.state)
        frontier.add(self.state.hash_value)
        explored = set()
        nodes_expanded = depth = 0
        while queue:
            position = queue.popleft()
            frontier.remove(position.hash_value)
            if position.value == goal:
                print_goal_from_parent(position)
                print('nodes expanded:', nodes_expanded)
                trace_path(position)
                print('max depth:', depth)
                return 'success'

            explored.add(position.hash_value)
            neighbours = position.generate_neighbours()
            nodes_expanded += 1

            for neighbour in neighbours:
                if not (neighbour.hash_value in frontier or neighbour.hash_value
                    depth = max(depth, neighbour.depth)
                    queue.append(neighbour)
                    frontier.add(neighbour.hash_value)
```

```
    def dfs(self):
        stack = deque()
        frontier = set()
        stack.append(self.state)
        frontier.add(self.state.hash_value)
        explored = set()
        nodes_expanded = depth = 0
        while stack:
            position = stack.pop()
            frontier.remove(position.hash_value)
            if position.value == goal:
                print_goal_from_parent(position)
                print('nodes expanded:', nodes_expanded)
                print('max depth:', depth)
                trace_path(position)
                return 'success'

            explored.add(position.hash_value)
            neighbours = position.generate_neighbours()
            neighbours.reverse()
            nodes_expanded += 1
            for neighbour in neighbours:
                if not (neighbour.hash_value in frontier or neighbour.hash_value
                    depth = max(depth, neighbour.depth)
                    stack.append(neighbour)
                    frontier.add(neighbour.hash_value)
```

```
def astar_manh(self):
    heap = []
    heappush(heap, (manhattan_distance(self.state), self.state))
    keys = dict()
    keys[self.state.hash_value] = self.state
    nodes_expanded = 0
    explored = set()

    while heap:
        score, position = heappop(heap)
        del keys[position.hash_value]
        explored.add(position.hash_value)

        if position.value == goal:
            print_goal_from_parent(position)
            print('nodes expanded:', nodes_expanded)
            trace_path(position)
            return 'success'
        neighbours = position.generate_neighbours()
        nodes_expanded += 1
        for neighbour in neighbours:
            if not (neighbour.hash_value in keys or neighbour.hash_value in
                heappush(heap, (manhattan_distance(neighbour) + neighbour.de
                keys[neighbour.hash_value] = neighbour
            elif neighbour.hash_value in keys:
                node = keys[neighbour.hash_value]
                if manhattan_distance(neighbour) < manhattan_distance(node):
                    node.parent = position
                    node.depth = neighbour.depth
```

```
def astar_euc(self):
    heap = []
    heappush(heap, (euclidean_distance(self.state), self.state))
    keys = dict()
    keys[self.state.hash_value] = self.state
    nodes_expanded = 0
    explored = set()

    while heap:
        score, position = heappop(heap)
        del keys[position.hash_value]
        explored.add(position.hash_value)

        if position.value == goal:
            print_goal_from_parent(position)
            print('nodes expanded:', nodes_expanded)
            trace_path(position)
            return 'success'
        neighbours = position.generate_neighbours()
        nodes_expanded += 1
        for neighbour in neighbours:
            if not (neighbour.hash_value in keys or neighbour.hash_value in
                heappush(heap, (euclidean_distance(neighbour) + neighbour.de
                keys[neighbour.hash_value] = neighbour
            elif neighbour.hash_value in keys:
                node = keys[neighbour.hash_value]
                if euclidean_distance(neighbour) < euclidean_distance(node):
                    node.parent = position
                    node.depth = neighbour.depth
```

The heuristics of A* (Both Manhattan and Euclidean)

```
#   calculates manhattan's distance h(n)
def manhattan_distance(state):
        puzzle = state.value
        dist = 0
        for i, val in enumerate(puzzle):
            if val == 0:
                continue

            goal_y, goal_x = val // 3, val % 3
            y, x = i // 3, i % 3
            dist += abs(goal_y-y) + abs(goal_x-x)
        return dist

#   calculates euclidean's distance h(n)
def euclidean_distance(state):
        puzzle = state.value
        dist = 0
        for i, val in enumerate(puzzle):
            if val == 0:
                continue

            goal_y, goal_x = val // 3, val % 3
            y, x = i // 3, i % 3
            dist += math.sqrt(abs(goal_y-y)**2 + abs(goal_x-x)**2)
        return dist
```

Sample Runs

```
import time

#   input puzzles to solve
puzzle = [1, 2, 0, 3, 4, 5, 6, 7, 8]
puzzle0 = [1, 3, 4, 8, 0, 5, 7, 2, 6]
puzzle1 = [1, 2, 3, 4, 5, 0, 6, 7, 8]
puzzle2 = [1, 2, 5, 3, 4, 0, 6, 7, 8]
puzzle3 = [0, 1, 3, 4, 2, 5, 7, 8, 6]
puzzle4 = [8, 3, 5, 4, 1, 6, 2, 7, 0]    # unsolvable
puzzle5 = [2, 8, 3, 1, 0, 5, 4, 7, 6]
puzzle6 = [8, 0, 6, 5, 4, 7, 2, 3, 1]
puzzle7 = [1, 2, 3, 4, 8, 0, 7, 6, 5]
puzzle8 = [1, 2, 0, 3, 4, 5, 6, 7, 8]
puzzle9 = [1, 2, 3, 4, 5, 0, 6, 7, 8]
puzzle10 = [1, 2, 3, 4, 0, 5, 6, 7, 8]
puzzle11 = [1, 2, 3, 0, 4, 5, 6, 7, 8]
puzzle12 = [3, 1, 2, 6, 4, 5, 0, 7, 8]
puzzle13 = [6, 1, 8, 4, 0, 2, 7, 3, 5]
game = State(puzzle2)
search = Solver(game)
print('\nBFS:')
```

Puzzle = [1, 2, 5, 3, 4, 0, 6, 7, 8]

```
shahenda@Lenovo ~/Downloads/EightPuzzle-AI $ python Main.py

BFS:
=======================================
[[1, 2, 5, 3, 4, 0, 6, 7, 8], [1, 2, 0, 3, 4, 5, 6, 7, 8], [1, 0, 2, 3, 4, 5, 6, 7, 8], [0, 1, 2, 3, 4, 5, 6, 7, 8]]
('depth:', 3)
('moves:', 3)
('nodes expanded:', 10)
('path:', ['up', 'left', 'left'])
('max depth:', 4)
('time taken:', 0.00026702880859375)

DFS:
=======================================
[[1, 2, 5, 3, 4, 0, 6, 7, 8], [1, 2, 0, 3, 4, 5, 6, 7, 8], [1, 0, 2, 3, 4, 5, 6, 7, 8], [0, 1, 2, 3, 4, 5, 6, 7, 8]]
('depth:', 3)
('moves:', 3)
('nodes expanded:', 181437)
('max depth:', 66125)
('path:', ['up', 'left', 'left'])
('time taken:', 1.8108329772949219)

A* by Manhattan heuristic:
=======================================
[[1, 2, 5, 3, 4, 0, 6, 7, 8], [1, 2, 0, 3, 4, 5, 6, 7, 8], [1, 0, 2, 3, 4, 5, 6, 7, 8], [0, 1, 2, 3, 4, 5, 6, 7, 8]]
('depth:', 3)
('moves:', 3)
('nodes expanded:', 3)
('path:', ['up', 'left', 'left'])
('time taken:', 0.0002579689025878906)

A* by Euclidean heuristic:
=======================================
[[1, 2, 5, 3, 4, 0, 6, 7, 8], [1, 2, 0, 3, 4, 5, 6, 7, 8], [1, 0, 2, 3, 4, 5, 6, 7, 8], [0, 1, 2, 3, 4, 5, 6, 7, 8]]
('depth:', 3)
('moves:', 3)
('nodes expanded:', 3)
('path:', ['up', 'left', 'left'])
('time taken:', 0.00015401840209960938)
shahenda@Lenovo ~/Downloads/EightPuzzle-AI $
```

puzzle = [1, 2, 0, 3, 4, 5, 6, 7, 8]

```
BFS:
========================================
[[1, 2, 0, 3, 4, 5, 6, 7, 8], [1, 0, 2, 3, 4, 5, 6, 7, 8], [0, 1, 2, 3, 4, 5, 6, 7, 8]]
('depth:', 2)
('moves:', 2)
('nodes expanded:', 6)
('path:', ['left', 'left'])
('max depth:', 3)
('time taken:', 0.00037598609924316406)

DFS:
========================================
[[1, 2, 0, 3, 4, 5, 6, 7, 8], [1, 2, 5, 3, 4, 0, 6, 7, 8], [1, 2, 5, 3, 4, 8, 6, 7, 0], [1, 2, 5, 3, 4, 8, 6, 0, 7], [1, 2, 5, 3, 0, 8, 6, 4, 7], [1, 0, 5, 3, 2, 8, 6,
4, 7], [0, 1, 5, 3, 2, 8, 6, 4, 7], [3, 1, 5, 0, 2, 8, 6, 4, 7], [3, 1, 5, 6, 2, 8, 0, 4, 7], [3, 1, 5, 6, 2, 8, 4, 0, 7], [3, 1, 5, 6, 0, 8, 4, 2, 7], [3, 0, 5, 6, 1,
8, 4, 2, 7], [0, 3, 5, 6, 1, 8, 4, 2, 7], [6, 3, 5, 0, 1, 8, 4, 2, 7], [6, 3, 5, 4, 1, 8, 0, 2, 7], [6, 3, 5, 4, 1, 8, 2, 0, 7], [6, 3, 5, 4, 0, 8, 2, 1, 7], [6, 0, 5,
4, 3, 8, 2, 1, 7], [0, 6, 5, 4, 3, 8, 2, 1, 7], [4, 6, 5, 0, 3, 8, 2, 1, 7], [4, 6, 5, 2, 3, 8, 0, 1, 7], [4, 6, 5, 2, 3, 8, 1, 0, 7], [4, 6, 5, 2, 0, 8, 1, 3, 7], [4,
0, 5, 2, 6, 8, 1, 3, 7], [0, 4, 5, 2, 6, 8, 1, 3, 7], [2, 4, 5, 0, 6, 8, 1, 3, 7], [2, 4, 5, 1, 6, 8, 0, 3, 7], [2, 4, 5, 1, 6, 8, 3, 0, 7], [2, 4, 5, 1, 0, 8, 3, 6, 7]
, [2, 0, 5, 1, 4, 8, 3, 6, 7], [0, 2, 5, 1, 4, 8, 3, 6, 7], [1, 2, 5, 0, 4, 8, 3, 6, 7], [1, 2, 5, 4, 0, 8, 3, 6, 7], [1, 0, 5, 4, 2, 8, 3, 6, 7], [0, 1, 5, 4, 2, 8, 3,
 6, 7], [4, 1, 5, 0, 2, 8, 3, 6, 7], [4, 1, 5, 3, 2, 8, 0, 6, 7], [4, 1, 5, 3, 2, 8, 6, 0, 7], [4, 1, 5, 3, 0, 8, 6, 2, 7], [4, 0, 5, 3, 1, 8, 6, 2, 7], [0, 4, 5, 3, 1,
 8, 6, 2, 7], [3, 4, 5, 0, 1, 8, 6, 2, 7], [3, 4, 5, 6, 1, 8, 0, 2, 7], [3, 4, 5, 6, 1, 8, 2, 0, 7], [3, 4, 5, 6, 0, 8, 2, 1, 7], [3, 0, 5, 6, 4, 8, 2, 1, 7], [3, 5, 0,
 6, 4, 8, 2, 1, 7], [3, 5, 8, 6, 4, 0, 2, 1, 7], [3, 5, 8, 6, 4, 7, 2, 1, 0], [3, 5, 8, 6, 4, 7, 2, 0, 1], [3, 5, 8, 6, 0, 7, 2, 4, 1], [3, 0, 8, 6, 5, 7, 2, 4, 1], [0,
 3, 8, 6, 5, 7, 2, 4, 1], [6, 3, 8, 0, 5, 7, 2, 4, 1], [6, 3, 8, 2, 5, 7, 0, 4, 1], [6, 3, 8, 2, 5, 7, 4, 0, 1], [6, 3, 8, 2, 0, 7, 4, 5, 1], [6, 0, 8, 2, 3, 7, 4, 5, 1
], [0, 6, 8, 2, 3, 7, 4, 5, 1], [2, 6, 8, 0, 3, 7, 4, 5, 1], [2, 6, 8, 4, 3, 7, 0, 5, 1], [2, 6, 8, 4, 3, 7, 5, 0, 1], [2, 6, 8, 4, 0, 7, 5, 3, 1], [2, 0, 8, 4, 6, 7, 5
, 3, 1], [0, 2, 8, 4, 6, 7, 5, 3, 1], [4, 2, 8, 0, 6, 7, 5, 3, 1], [4, 2, 8, 5, 6, 7, 0, 3, 1], [4, 2, 8, 5, 6, 7, 3, 0, 1], [4, 2, 8, 5, 0, 7, 3, 6, 1], [4, 0, 8, 5, 2
, 7, 3, 6, 1], [0, 4, 8, 5, 2, 7, 3, 6, 1], [5, 4, 8, 0, 2, 7, 3, 6, 1], [5, 4, 8, 3, 2, 7, 0, 6, 1], [5, 4, 8, 3, 2, 7, 6, 0, 1], [5, 4, 8, 3, 0, 7, 6, 2, 1], [5, 0, 8
, 3, 4, 7, 6, 2, 1], [0, 5, 8, 3, 4, 7, 6, 2, 1], [3, 5, 8, 0, 4, 7, 6, 2, 1], [3, 5, 8, 4, 0, 7, 6, 2, 1], [3, 0, 8, 4, 5, 7, 6, 2, 1], [0, 3, 8, 4, 5, 7, 6, 2, 1], [4
, 3, 8, 0, 5, 7, 6, 2, 1], [4, 3, 8, 6, 5, 7, 0, 2, 1], [4, 3, 8, 6, 5, 7, 2, 0, 1], [4, 3, 8, 6, 0, 7, 2, 5, 1], [4, 0, 8, 6, 3, 7, 2, 5, 1], [0, 4, 8, 6, 3, 7, 2, 5,
1], [6, 4, 8, 0, 3, 7, 2, 5, 1], [6, 4, 8, 2, 3, 7, 0, 5, 1], [6, 4, 8, 2, 3, 7, 5, 0, 1], [6, 4, 8, 2, 0, 7, 5, 3, 1], [6, 0, 8, 2, 4, 7, 5, 3, 1], [6, 8, 0, 2, 4, 7,
5, 3, 1], [6, 8, 7, 2, 4, 0, 5, 3, 1], [6, 8, 7, 2, 4, 1, 5, 3, 0], [6, 8, 7, 2, 4, 1, 5, 0, 3], [6, 8, 7, 2, 0, 1, 5, 4, 3], [6, 0, 7, 2, 8, 1, 5, 4, 3], [0, 6, 7, 2,
8, 1, 5, 4, 3], [2, 6, 7, 0, 8, 1, 5, 4, 3], [2, 6, 7, 5, 8, 1, 0, 4, 3], [2, 6, 7, 5, 8, 1, 4, 0, 3], [2, 6, 7, 5, 0, 1, 4, 8, 3], [2, 0, 7, 5, 6, 1, 4, 8, 3], [0, 2,
7, 5, 6, 1, 4, 8, 3], [5, 2, 7, 0, 6, 1, 4, 8, 3], [5, 2, 7, 4, 6, 1, 0, 8, 3], [5, 2, 7, 4, 6, 1, 8, 0, 3], [5, 2, 7, 4, 0, 1, 8, 6, 3], [5, 0, 7, 4, 2, 1, 8, 6, 3], [
0, 5, 7, 4, 2, 1, 8, 6, 3], [4, 5, 7, 0, 2, 1, 8, 6, 3], [4, 5, 7, 8, 2, 1, 0, 6, 3], [4, 5, 7, 8, 2, 1, 6, 0, 3], [4, 5, 7, 8, 0, 1, 6, 2, 3], [4, 0, 7, 8, 5, 1, 6, 2,
 3], [0, 4, 7, 8, 5, 1, 6, 2, 3], [8, 4, 7, 0, 5, 1, 6, 2, 3], [8, 4, 7, 6, 5, 1, 0, 2, 3], [8, 4, 7, 6, 5, 1, 2, 0, 3], [8, 4, 7, 6, 0, 1, 2, 5, 3], [8, 0, 7, 6, 4, 1,
 2, 5, 3], [0, 8, 7, 6, 4, 1, 2, 5, 3], [6, 8, 7, 0, 4, 1, 2, 5, 3], [6, 8, 7, 4, 0, 1, 2, 5, 3], [6, 0, 7, 4, 8, 1, 2, 5, 3], [0, 6, 7, 4, 8, 1, 2, 5, 3], [4, 6, 7, 0,
 8, 1, 2, 5, 3], [4, 6, 7, 2, 8, 1, 0, 5, 3], [4, 6, 7, 2, 8, 1, 5, 0, 3], [4, 6, 7, 2, 0, 1, 5, 8, 3], [4, 0, 7, 2, 6, 1, 5, 8, 3], [0, 4, 7, 2, 6, 1, 5, 8, 3], [2, 4,
 7, 0, 6, 1, 5, 8, 3], [2, 4, 7, 5, 6, 1, 0, 8, 3], [2, 4, 7, 5, 6, 1, 8, 0, 3], [2, 4, 7, 5, 0, 1, 8, 6, 3], [2, 0, 7, 5, 4, 1, 8, 6, 3], [2, 7, 0, 5, 4, 1, 8, 6, 3],
[2, 7, 1, 5, 4, 0, 8, 6, 3], [2, 7, 1, 5, 4, 3, 8, 6, 0], [2, 7, 1, 5, 4, 3, 8, 0, 6], [2, 7, 1, 5, 0, 3, 8, 4, 6], [2, 0, 1, 5, 7, 3, 8, 4, 6], [0, 2, 1, 5, 7, 3, 8, 4
, 6], [5, 2, 1, 0, 7, 3, 8, 4, 6], [5, 2, 1, 8, 7, 3, 0, 4, 6], [5, 2, 1, 8, 7, 3, 4, 0, 6], [5, 2, 1, 8, 0, 3, 4, 7, 6], [5, 0, 1, 8, 2, 3, 4, 7, 6], [0, 5, 1, 8, 2, 3
, 4, 7, 6], [8, 5, 1, 0, 2, 3, 4, 7, 6], [8, 5, 1, 4, 2, 3, 0, 7, 6], [8, 5, 1, 4, 2, 3, 7, 0, 6], [8, 5, 1, 4, 0, 3, 7, 2, 6], [8, 0, 1, 4, 5, 3, 7, 2, 6], [0, 8, 1, 4
, 5, 3, 7, 2, 6], [4, 8, 1, 0, 5, 3, 7, 2, 6], [4, 8, 1, 7, 5, 3, 0, 2, 6], [4, 8, 1, 7, 5, 3, 2, 0, 6], [4, 8, 1, 7, 0, 3, 2, 5, 6], [4, 0, 1, 7, 8, 3, 2, 5, 6], [0, 4
, 1, 7, 8, 3, 2, 5, 6], [7, 4, 1, 0, 8, 3, 2, 5, 6], [7, 4, 1, 2, 8, 3, 0, 5, 6], [7, 4, 1, 2, 8, 3, 5, 0, 6], [7, 4, 1, 2, 0, 3, 5, 8, 6], [7, 0, 1, 2, 4, 3, 5, 8, 6],
 [0, 7, 1, 2, 4, 3, 5, 8, 6], [2, 7, 1, 0, 4, 3, 5, 8, 6], [2, 7, 1, 4, 0, 3, 5, 8, 6], [2, 0, 1, 4, 7, 3, 5, 8, 6], [0, 2, 1, 4, 7, 3, 5, 8, 6], [4, 2, 1, 0, 7, 3, 5,
8, 6], [4, 2, 1, 5, 7, 3, 0, 8, 6], [4, 2, 1, 5, 7, 3, 8, 0, 6], [4, 2, 1, 5, 0, 3, 8, 7, 6], [4, 0, 1, 5, 2, 3, 8, 7, 6], [0, 4, 1, 5, 2, 3, 8, 7, 6], [5, 4, 1, 0, 2,
```

```
Terminal
File  Edit  View  Search  Terminal  Help
5], [3, 4, 6, 0, 1, 2, 8, 7, 5], [3, 4, 6, 8, 1, 2, 0, 7, 5], [3, 4, 6, 8, 1, 2, 7, 0, 5], [3, 4, 6, 8, 0, 2, 7, 1, 5], [3, 0, 6, 8, 4, 2, 7, 1, 5], [0, 3, 6, 8, 4, 2,
7, 1, 5], [8, 3, 6, 0, 4, 2, 7, 1, 5], [8, 3, 6, 4, 0, 2, 7, 1, 5], [8, 0, 6, 4, 3, 2, 7, 1, 5], [0, 8, 6, 4, 3, 2, 7, 1, 5], [4, 8, 6, 0, 3, 2, 7, 1, 5], [4, 8, 6, 7,
3, 2, 0, 1, 5], [4, 8, 6, 7, 3, 2, 1, 0, 5], [4, 8, 6, 7, 0, 2, 1, 3, 5], [4, 0, 6, 7, 8, 2, 1, 3, 5], [0, 4, 6, 7, 8, 2, 1, 3, 5], [7, 4, 6, 0, 8, 2, 1, 3, 5], [7, 4,
6, 1, 8, 2, 0, 3, 5], [7, 4, 6, 1, 8, 2, 3, 0, 5], [7, 4, 6, 1, 0, 2, 3, 8, 5], [7, 0, 6, 1, 4, 2, 3, 8, 5], [7, 6, 0, 1, 4, 2, 3, 8, 5], [7, 6, 2, 1, 4, 0, 3, 8, 5], [
7, 6, 2, 1, 4, 5, 3, 8, 0], [7, 6, 2, 1, 4, 5, 3, 0, 8], [7, 6, 2, 1, 0, 5, 3, 4, 8], [7, 0, 2, 1, 6, 5, 3, 4, 8], [0, 7, 2, 1, 6, 5, 3, 4, 8], [1, 7, 2, 0, 6, 5, 3, 4,
 8], [1, 7, 2, 3, 6, 5, 0, 4, 8], [1, 7, 2, 3, 6, 5, 4, 0, 8], [1, 7, 2, 3, 0, 5, 4, 6, 8], [1, 0, 2, 3, 7, 5, 4, 6, 8], [0, 1, 2, 3, 7, 5, 4, 6, 8], [3, 1, 2, 0, 7, 5,
 4, 6, 8], [3, 1, 2, 4, 7, 5, 0, 6, 8], [3, 1, 2, 4, 7, 5, 6, 0, 8], [3, 1, 2, 4, 0, 5, 6, 7, 8], [3, 1, 2, 0, 4, 5, 6, 7, 8], [0, 1, 2, 3, 4, 5, 6, 7, 8]]
('depth:', 294)
('moves:', 294)
('nodes expanded:', 181203)
('max depth:', 66488)
('path:', ['down', 'down', 'left', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'down', 'right'
, 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'right', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'down', 'right',
'up', 'up', 'right', 'down', 'down', 'left', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'down
', 'right', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'right', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'down',
 'right', 'up', 'up', 'right', 'down', 'down', 'left', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'do
wn', 'down', 'right', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'right', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down
', 'down', 'right', 'up', 'up', 'right', 'down', 'down', 'left', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', '
left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'right', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'le
ft', 'down', 'down', 'right', 'up', 'up', 'right', 'down', 'down', 'left', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'down', 'right', 'up
', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'right', 'up', 'left', 'down', 'down', 'right', 'up',
 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'right', 'down', 'down', 'left', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'down', 'r
ight', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down', 'right', 'up', 'left', 'down', 'down', 'rig
ht', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'right', 'down', 'down', 'left', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'up', 'left', 'down',
'down', 'right', 'up', 'left', 'up'])
('time taken:', 1.8039100170135498)

A* by Manhattan heuristic:
========================================
[[1, 2, 0, 3, 4, 5, 6, 7, 8], [1, 0, 2, 3, 4, 5, 6, 7, 8], [0, 1, 2, 3, 4, 5, 6, 7, 8]]
('depth:', 2)
('moves:', 2)
('nodes expanded:', 2)
('path:', ['left', 'left'])
('time taken:', 0.00015401840209960938)

A* by Euclidean heuristic:
========================================
[[1, 2, 0, 3, 4, 5, 6, 7, 8], [1, 0, 2, 3, 4, 5, 6, 7, 8], [0, 1, 2, 3, 4, 5, 6, 7, 8]]
('depth:', 2)
('moves:', 2)
('nodes expanded:', 2)
('path:', ['left', 'left'])
('time taken:', 0.00010895729064941406)
shahenda@Lenovo ~/Downloads/EightPuzzle-AI $
```