SOLVING A 15-Puzzle USING THE A-STAR ALGORITHM IN PYTHON
puzzle.state = [[1, 2, 4, 8],
                [9, 0, 3, 12],
                [7, 11, 14, 10],
                [5, 13, 6, 15]]

puzzle.goal_state = [[1, 2, 3, 4],
                      [5, 6, 7, 8],
                      [9, 10, 11, 12],
                      [13, 14, 15, 0]]
A puzzle with 16 fields has $16!$ different permutations and thus

$$16! = 20,922,789,888,000$$

possible states!
Informed Search

Breadth or depth first search could take long in this big state-space.

The a-star search with a good heuristic function reduces the amount of searched states, as the state closest to the goal state will always be expanded first.
A-Star Search

# This is the main search-loop

while len(frontier) > 0:
    heuristic_value, current_puzzle_path = heapq.heappop(frontier)
    current_puzzle = current_puzzle_path[0]

    if current_puzzle.state == puzzle.goal_state:
        print "Path found"; return

    closeList.append(current_puzzle.state) if current_puzzle.state not in closeList

    for new_puzzle in possible_actions(current_puzzle):
        new_path = [new_puzzle] + current_puzzle_path[:]
        if new_puzzle.state not in closeList:
            heapq.heappush(frontier, (new_puzzle.manhattan() + len(new_path), new_path))
        # else update the value function (if it is lower)
Heuristic Functions

1. Number of misplaced tiles
2. Sum of the manhattan distances of all tiles to their goal
Manhattan distance

# Iterate through all fields and compute their manhattan distance
# Return the sum of all distances

def manhattan(self):
    sum = 0
    for x in range(self.size):  # x dimension
        for y in range(self.size):  # y dimension
            current = self.state[x][y] - 1
            if current is not 0:
                x_distance = abs(x - current / self.size)
                y_distance = abs(y - current % self.size)
                sum += x_distance + y_distance
    return distance
Solvability

About 50% of all randomly generated 15-puzzles are not solvable.

Two possibilities to detect solvability

▸ Compute a puzzle from the goal state
▸ Use an algorithm to determine solvability
# Generates and returns a random solvable puzzle

def random_solvable_puzzle(size):
    puzzle = Puzzle(size)
    numbers = [number for number in range(1, size * size)]
    numbers.append(0)
    puzzle.state = np.reshape(numbers, (size, size)).tolist()

    for move in range(40):
        actions = possible_actions(puzzle)
        random.shuffle(actions)
        puzzle = actions[0]

    return puzzle
# Computes and returns the solvability of a puzzle

def is_solvable(puzzle):
    
inversions = get_inversions(puzzle)
    flat_puzzle = flatten(puzzle)
    blank_row = int(floor(flat_puzzle.index(0) / puzzle.size))
    even_inversions = inversions % 2 == 0

    if puzzle.size % 2 == 0 and blank_row % 2 == 0:
        return not even_inversions
    else:
        return even_inversions
Usage

$ python puzzle.py -s 3 -p '4 2 1 3 6 7 5 8 0'

-s, --size   'The size of the puzzle (default is 4)'
-p, --puzzle 'The start configuration of the puzzle'
Questions?