CSCI 111 - Final Project
8-Puzzle Problem

Ang, Brayden Jansen O.

Zialcita, Andrea Mikaela S.

8-Puzzle Problem

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

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

- Objective: find a solution (path from initial state to goal state)
- Length of Solution & Computation Time

```
import numpy as np
```

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8]
goal_state = np.array(numbers).reshape(3, 3)
goal_state
```

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```python
def swap_left(state):
  if 0 not in state[:, 0]:
    new_state = state.copy()
    r, c = np.where(new_state == 0)
    new_state[r[0], c[0]], new_state[r[0], c[0] - 1] = new_state[r[0], c[0] - 1], new_state[r[0], c[0]]
    return new_state
  else:
    return None
```

```
array([[4, 5, 1],
       [8, 6, 0],
       [2, 3, 7]])
```

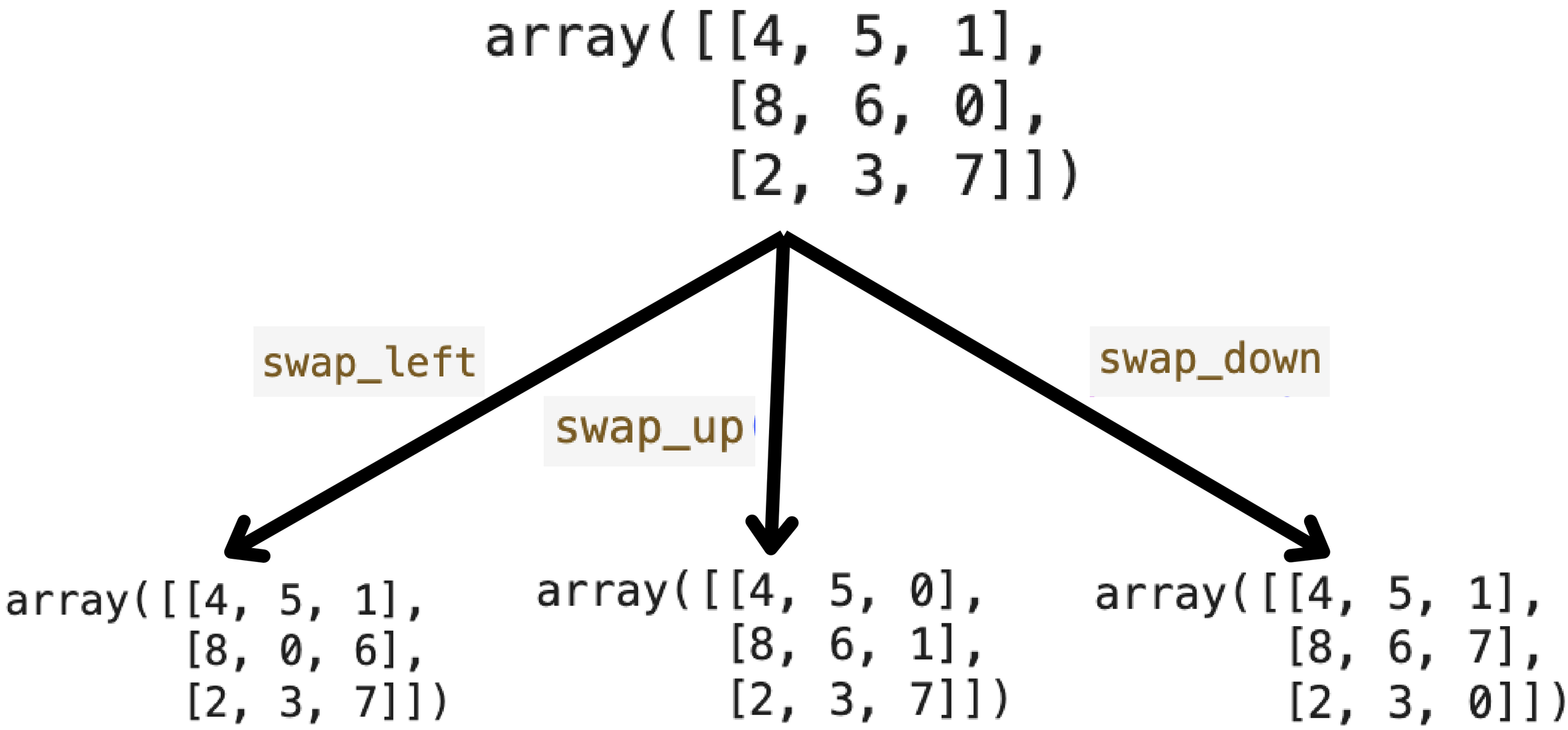
```
array([[4, 5, 1],
       [8, 0, 6],
       [2, 3, 7]])
```

```
def swap_left(state):
  if 0 not in state[:, 0]:
    new_state = state.copy()
    r, c = np.where(new_state == 0)
    new_state[r[0], c[0]], new_state[r[0], c[0] - 1] = new_state[r[0], c[0] - 1], new_state[r[0], c[0]]
    return new_state
  else:
    return None

def swap_up(state):
  if 0 not in state[0]:
    new_state = state.copy()
    r, c = np.where(new_state == 0)
    new_state[r[0], c[0]], new_state[r[0] - 1, c[0]] = new_state[r[0] - 1, c[0]], new_state[r[0], c[0]]
    return new_state
  else:
    return None
```

```
def swap_right(state):
  if 0 not in state[:, 2]:
    new_state = state.copy()
    r, c = np.where(new_state == 0)
    new_state[r[0], c[0]], new_state[r[0], c[0] + 1] = new_state[r[0], c[0] + 1], new_state[r[0], c[0]]
    return new_state
  else:
    return None

def swap_down(state):
  if 0 not in state[2]:
    new_state = state.copy()
    r, c = np.where(new_state == 0)
    new_state[r[0], c[0]], new_state[r[0] + 1, c[0]] = new_state[r[0] + 1, c[0]], new_state[r[0], c[0]]
    return new_state
  else:
    return None
```


array([[4, 5, 1],
[8, 6, 0],
[2, 3, 7]])
swap_left
swap_up
swap_down
array([[4, 5, 1],
[8, 0, 6],
[2, 3, 7]])
array([[4, 5, 0],
[8, 6, 1],
[2, 3, 7]])
array([[4, 5, 1],
[8, 6, 7],
[2, 3, 0]])

8-Puzzle Problem

BFS

utility functions

def equal (array1, array2)

```
def equal(array1, array2):
    return np.array_equal(array1, array2)
```

- checks if two NumPy arrays are equal
- returns True if the shape and all elements are the same for both arrays; returns False otherwise
- we will need this to check if the current state being processed matches the goal state.

def array_in_dict (array, dictionary)

```
def array_in_dict(array, dictionary):
    flattened_array = tuple(array.flatten())
    if flattened_array in dictionary:
        return True
    return False
```

- checks if a flattened NumPy array is present as a key in a dictionary

def is_solvable (state)

```python
def is_solvable(state):
    flat_state = state.flatten()
    inversions = 0
    for i in range(len(flat_state)):
        for j in range(i + 1, len(flat_state)):
            if flat_state[i] > 0 and flat_state[j] > 0 and flat_state[i] > flat_state[j]:
                inversions += 1
    return inversions % 2 == 0
```

- Determine if a given configuration is solvable.
- The function counts the number of inversions in the puzzle (pairs of tiles where a larger number precedes a smaller number) by flattening the state array and checking all pairs.
- The puzzle is solvable if the number of inversions is even.

example

```
ns = np.array([[8, 1, 2],
               [0, 4, 3],
               [7, 6, 5]])

is_solvable(ns)
```

```
False
```

11 inversions: Not solvable

```
s = np.array([[1, 8, 2],
              [0, 4, 3],
              [7, 6, 5]])

is_solvable(s)
```

```
True
```

10 inversions: Solvable

def get_neighbors (state)

```
def get_neighbors(state):
    neighbors = []
    for move_function in [swap_left, swap_up, swap_right, swap_down]:
        neighbor = move_function(state)
        if neighbor is not None:
            neighbors.append(neighbor)
    return neighbors
```

- Generates all possible states succeeding the current state by calling the movement functions (swap_left, swap_up, swap_right, swap_down).
- It iterates through all movement functions (left, up, right, down), applies each function to the state, and collects all valid (nonempty) states.

example

```
sample = np.array([[1, 2, 3], [4, 0, 5], [7, 8, 6]])

get_neighbors(sample)
```

```
[array([[1, 2, 3],
        [0, 4, 5],
        [7, 8, 6]]),
 array([[1, 0, 3],
        [4, 2, 5],
        [7, 8, 6]]),
 array([[1, 2, 3],
        [4, 5, 0],
        [7, 8, 6]]),
 array([[1, 2, 3],
        [4, 8, 5],
        [7, 0, 6]])]
```

example

```
sample = np.array([[1, 2, 3], [0, 4, 5], [7, 8, 6]])

get_neighbors(sample)
```

```
[array([[0, 2, 3],
        [1, 4, 5],
        [7, 8, 6]]),
 array([[1, 2, 3],
        [4, 0, 5],
        [7, 8, 6]]),
 array([[1, 2, 3],
        [7, 4, 5],
        [0, 8, 6]])]
```

def find_path (initial_state, successor_list)

```
def find_path(initial_state: np.array, successor_list: list):
    path = [goal_state]
    state = goal_state
    while not (state == initial_state).all():
        for pair in successor_list:
            if (pair[1] == state).all():
                path.append(pair[0])
                state = pair[0]
                break
    path.reverse()
    return tuple(path)
```

- Reconstructs the path from the initial state to the goal state using the successor_list.
- The path is built in reverse order (goal to initial), then reversed at the end.
- This is to allow backtracking from the goal state to the initial state using the successor_list, which contains (parent, child) pairs.

main function

def bfs (start)

```
from collections import deque
```

```
def bfs(start):
    if not is_solvable(start):
        return "This puzzle is not solvable."

    queue = deque([start])
    visited = {}
    successor_list = []
    nodes_inspected = 0

    while queue:
        current_state = queue.popleft()
        nodes_inspected += 1

        # mark the current state as visited
        visited[tuple(current_state.flatten())] = True

        # check if the current state is the goal state
        if equal(current_state, goal_state):
            print("Solvable!")
            path = find_path(start, successor_list)
            return (path, len(path), nodes_inspected)

        for neighbor in get_neighbors(current_state):
            if not array_in_dict(neighbor, visited):  # check against visited dictionary
                queue.append(neighbor)
                successor_list.append((current_state, neighbor))

    return "No solution found."
```

def bfs (start)

```
def bfs(start):
    if not is_solvable(start):
        return "This puzzle is not solvable."
```

- Initial check: It first checks if the puzzle is solvable using is_solvable(). If not, it returns a message indicating the puzzle is unsolvable.

def bfs (start)

```
queue = deque([start])
visited = {}
successor_list = []
nodes_inspected = 0
```

- Initializing the queue: A queue is initialized with the start state, and a dictionary (labeled “visited”) is used to track visited states.
  - First In, First Out
  - To manage the states that need to be explored
  - States are dequed one by one and expanded
- Note: nodes_inspected is just a counter for the number of nodes inspected (used for discussion of results later)

def bfs (start)

```
while queue:
    current_state = queue.popleft()
    nodes_inspected += 1

    # mark the current state as visited
    visited[tuple(current_state.flatten())] = True

    # check if the current state is the goal state
    if equal(current_state, goal_state):
        print("Solvable!")
        path = find_path(start, successor_list)
        return (path, len(path), nodes_inspected)

    for neighbor in get_neighbors(current_state):
        if not array_in_dict(neighbor, visited):  # check against visited dictionary
            queue.append(neighbor)
            successor_list.append((current_state, neighbor))

return "No solution found."
```

- States are dequeued one at a time.
- queue.popleft(): current_state is dequed from the front of the queue of the queue for processing

def bfs (start)

```
while queue:
    current_state = queue.popleft()
    nodes_inspected += 1

    # mark the current state as visited
    visited[tuple(current_state.flatten())] = True

    # check if the current state is the goal state
    if equal(current_state, goal_state):
        print("Solvable!")
        path = find_path(start, successor_list)
        return (path, len(path), nodes_inspected)

    for neighbor in get_neighbors(current_state):
        if not array_in_dict(neighbor, visited):  # check against visited dictionary
            queue.append(neighbor)
            successor_list.append((current_state, neighbor))

return "No solution found."
```

- Ensures that each state is marked as visited before exploring its children to prevent revisiting states and looping over the same state multiple times.
- Visited states are flattened to a tuple form for efficient checking.

def bfs (start)

```
while queue:
    current_state = queue.popleft()
    nodes_inspected += 1

    # mark the current state as visited
    visited[tuple(current_state.flatten())] = True

    # check if the current state is the goal state
    if equal(current_state, goal_state):
        print("Solvable!")
        path = find_path(start, successor_list)
        return (path, len(path), nodes_inspected)

    for neighbor in get_neighbors(current_state):
        if not array_in_dict(neighbor, visited):  # check against visited dictionary
            queue.append(neighbor)
            successor_list.append((current_state, neighbor))

return "No solution found."
```

Note: the function returns a triple containing the path, the length of the path, and the number of nodes inspected before finding the path.

- After dequeuing a state, the algorithm checks if the state matches the goal state using the equal() function. If it does, the algorithm terminates and the solution path is reconstructed by callling the find_path() function.

def bfs (start)

```
while queue:
    current_state = queue.popleft()
    nodes_inspected += 1

    # mark the current state as visited
    visited[tuple(current_state.flatten())] = True

    # check if the current state is the goal state
    if equal(current_state, goal_state):
        print("Solvable!")
        path = find_path(start, successor_list)
        return (path, len(path), nodes_inspected)

    for neighbor in get_neighbors(current_state):
        if not array_in_dict(neighbor, visited):  # check against visited dictionary
            queue.append(neighbor)
            successor_list.append((current_state, neighbor))

return "No solution found."
```

- If the state is not the goal, its neighbors are generated using get_neighbors(). If a neighbor hasn't been visited, it is queued, and the parent-child pair is added to the successor_list.

def bfs (start)

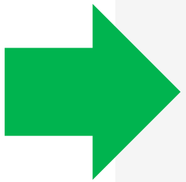
```
while queue:
    current_state = queue.popleft()
    nodes_inspected += 1

    # mark the current state as visited
    visited[tuple(current_state.flatten())] = True

    # check if the current state is the goal state
    if equal(current_state, goal_state):
        print("Solvable!")
        path = find_path(start, successor_list)
        return (path, len(path), nodes_inspected)

    for neighbor in get_neighbors(current_state):
        if not array_in_dict(neighbor, visited):  # check against visited dictionary
            queue.append(neighbor)
            successor_list.append((current_state, neighbor))

return "No solution found."
```



- If the neighbor has not been visited (i.e. checked using array_in_dict), it is added to the queue for future exploration.
- A (current_state, neighbor) pair is added to successor_list to keep track of how the neighbor was reached.

def bfs (start)

```
while queue:
    current_state = queue.popleft()
    nodes_inspected += 1

    # mark the current state as visited
    visited[tuple(current_state.flatten())] = True

    # check if the current state is the goal state
    if equal(current_state, goal_state):
        print("Solvable!")
        path = find_path(start, successor_list)
        return (path, len(path), nodes_inspected)

    for neighbor in get_neighbors(current_state):
        if not array_in_dict(neighbor, visited):  # check against visited dictionary
            queue.append(neighbor)
            successor_list.append((current_state, neighbor))

return "No solution found."
```

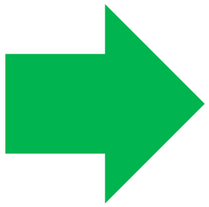
- If the queue is emptied without finding the goal state, then the function concludes that no solution exists. It returns a "No solution found" message.

demonstration

Solvable

```
solvable_state = np.array([[1, 8, 2],
                           [0, 4, 3],
                           [7, 6, 5]])

bfs(solvable_state)
```



```
Solvable!
((array([[1, 8, 2],
         [0, 4, 3],
         [7, 6, 5]]),
  array([[1, 8, 2],
         [4, 0, 3],
         [7, 6, 5]]),
  array([[1, 0, 2],
         [4, 8, 3],
         [7, 6, 5]]),
  array([[0, 1, 2],
         [4, 8, 3],
         [7, 6, 5]]),
  array([[4, 1, 2],
         [0, 8, 3],
         [7, 6, 5]]),
  array([[4, 1, 2],
         [7, 8, 3],
         [0, 6, 5]]),
  array([[4, 1, 2],
         [7, 8, 3],
         [6, 0, 5]]),
  array([[4, 1, 2],
         [7, 0, 3],
         [6, 8, 5]]),
  array([[4, 1, 2],
         [7, 3, 0],
         [6, 8, 5]]),
  array([[4, 1, 0],
         [7, 3, 2],
         [6, 8, 5]]),
  array([[4, 0, 1],
         [7, 3, 2],
         [6, 8, 5]]),
  array([[4, 3, 1],
         [7, 0, 2],
         [6, 8, 5]]),
  array([[4, 3, 1],
         [0, 7, 2],
         [6, 8, 5]]),
  array([[0, 3, 1],
         [4, 7, 2],
         [6, 8, 5]]),
  array([[3, 0, 1],
         [4, 7, 2],
         [6, 8, 5]]),
  array([[3, 1, 0],
         [4, 7, 2],
         [6, 8, 5]]),
  array([[3, 1, 2],
         [4, 7, 0],
         [6, 8, 5]]),
  array([[3, 1, 2],
         [4, 7, 5],
         [6, 8, 0]]),
  array([[3, 1, 2],
         [4, 7, 5],
         [6, 0, 8]]),
  array([[3, 1, 2],
         [4, 0, 5],
         [6, 7, 8]]),
  array([[3, 1, 2],
         [0, 4, 5],
         [6, 7, 8]]),
  array([[0, 1, 2],
         [3, 4, 5],
         [6, 7, 8]])),
 22,
 103525)
```

(solution is 22 moves long, and 103,525 nodes were inspected)

demonstration

Not Solvable

```
unsolvable_state = np.array([[8, 1, 2],
                             [0, 4, 3],
                             [7, 6, 5]])


bfs(unsolvable_state)
```

```
'This puzzle is not solvable.'
```

8-Puzzle Problem

Greedy Best-First Search

```
def correct_position(n):
  if n == 1:
    return (0, 1)
  elif n == 2:
    return (0, 2)
  elif n == 3:
    return (1, 0)
  elif n == 4:
    return (1, 1)
  elif n == 5:
    return (1, 2)
  elif n == 6:
    return (2, 0)
  elif n == 7:
    return (2, 1)
  elif n == 8:
    return (2, 2)
  else:
    return
# Define the correct position of each tile.
```

```
def manhattan_distance(state: np.array):
  m = 0
  for row in state:
    for number in row:
      if number != 0:
        r, c = np.where(state == number)
        r0, c0 = correct_position(number)[0], correct_position(number)[1]
        m += abs(r[0] - r0) + abs(c[0] - c0)
  return m
```

```
sample = np.array([[0, 3, 2], [1, 4, 5], [6, 7, 8]])

manhattan_distance(sample)
```

```
4
```

```
array([[0, 3, 2],
       [1, 4, 5],
       [6, 7, 8]])
```

```python
def sort_frontier(frontier: dict):
  sorted_items = sorted(frontier.items(), key=lambda item: item[1])
  frontier.clear()
  frontier.update(sorted_items)
# Sort the frontier by increasing total Manhattan distance.
```

```
{(0, 8, 2, 1, 4, 3, 7, 6, 5): 10, (1, 8, 2, 4, 0, 3, 7, 6, 5): 10, (1, 8, 2, 7, 4, 3, 0, 6, 5): 10}
{(1, 8, 2, 4, 0, 3, 7, 6, 5): 10, (1, 8, 2, 7, 4, 3, 0, 6, 5): 10, (8, 0, 2, 1, 4, 3, 7, 6, 5): 11}
{(1, 0, 2, 4, 8, 3, 7, 6, 5): 9, (1, 8, 2, 4, 3, 0, 7, 6, 5): 9, (1, 8, 2, 7, 4, 3, 0, 6, 5): 10, (8, 0, 2, 1, 4, 3, 7, 6, 5): 11,
{(0, 1, 2, 4, 8, 3, 7, 6, 5): 8, (1, 8, 2, 4, 3, 0, 7, 6, 5): 9, (1, 8, 2, 7, 4, 3, 0, 6, 5): 10, (1, 2, 0, 4, 8, 3, 7, 6, 5): 10,
{(1, 8, 2, 4, 3, 0, 7, 6, 5): 9, (4, 1, 2, 0, 8, 3, 7, 6, 5): 9, (1, 8, 2, 7, 4, 3, 0, 6, 5): 10, (1, 2, 0, 4, 8, 3, 7, 6, 5): 10,
{(1, 8, 2, 4, 3, 5, 7, 6, 0): 8, (4, 1, 2, 0, 8, 3, 7, 6, 5): 9, (1, 8, 2, 7, 4, 3, 0, 6, 5): 10, (1, 2, 0, 4, 8, 3, 7, 6, 5): 10,
{(4, 1, 2, 0, 8, 3, 7, 6, 5): 9, (1, 8, 2, 4, 3, 5, 7, 0, 6): 9, (1, 8, 2, 7, 4, 3, 0, 6, 5): 10, (1, 2, 0, 4, 8, 3, 7, 6, 5): 10,
{(1, 8, 2, 4, 3, 5, 7, 0, 6): 9, (1, 8, 2, 7, 4, 3, 0, 6, 5): 10, (1, 2, 0, 4, 8, 3, 7, 6, 5): 10, (1, 8, 0, 4, 3, 2, 7, 6, 5): 10,
{(1, 8, 2, 4, 3, 5, 0, 7, 6): 8, (1, 8, 2, 7, 4, 3, 0, 6, 5): 10, (1, 2, 0, 4, 8, 3, 7, 6, 5): 10, (1, 8, 0, 4, 3, 2, 7, 6, 5): 10,
{(1, 8, 2, 0, 3, 5, 4, 7, 6): 9, (1, 8, 2, 7, 4, 3, 0, 6, 5): 10, (1, 2, 0, 4, 8, 3, 7, 6, 5): 10, (1, 8, 0, 4, 3, 2, 7, 6, 5): 10,
{(1, 8, 2, 3, 0, 5, 4, 7, 6): 8, (1, 8, 2, 7, 4, 3, 0, 6, 5): 10, (1, 2, 0, 4, 8, 3, 7, 6, 5): 10, (1, 8, 0, 4, 3, 2, 7, 6, 5): 10,
{(1, 0, 2, 3, 8, 5, 4, 7, 6): 7, (1, 8, 2, 3, 5, 0, 4, 7, 6): 9, (1, 8, 2, 3, 7, 5, 4, 0, 6): 9, (1, 8, 2, 7, 4, 3, 0, 6, 5): 10, (
{(0, 1, 2, 3, 8, 5, 4, 7, 6): 6, (1, 2, 0, 3, 8, 5, 4, 7, 6): 8, (1, 8, 2, 3, 5, 0, 4, 7, 6): 9, (1, 8, 2, 3, 7, 5, 4, 0, 6): 9, (1
{(3, 1, 2, 0, 8, 5, 4, 7, 6): 7, (1, 2, 0, 3, 8, 5, 4, 7, 6): 8, (1, 8, 2, 3, 5, 0, 4, 7, 6): 9, (1, 8, 2, 3, 7, 5, 4, 0, 6): 9, (1
{(3, 1, 2, 4, 8, 5, 0, 7, 6): 6, (1, 2, 0, 3, 8, 5, 4, 7, 6): 8, (3, 1, 2, 8, 0, 5, 4, 7, 6): 8, (1, 8, 2, 3, 5, 0, 4, 7, 6): 9, (1
{(3, 1, 2, 4, 8, 5, 7, 0, 6): 7, (1, 2, 0, 3, 8, 5, 4, 7, 6): 8, (3, 1, 2, 8, 0, 5, 4, 7, 6): 8, (1, 8, 2, 3, 5, 0, 4, 7, 6): 9, (1
{(3, 1, 2, 4, 0, 5, 7, 8, 6): 6, (3, 1, 2, 4, 8, 5, 7, 6, 0): 6, (1, 2, 0, 3, 8, 5, 4, 7, 6): 8, (3, 1, 2, 8, 0, 5, 4, 7, 6): 8, (1
{(3, 1, 2, 0, 4, 5, 7, 8, 6): 5, (3, 1, 2, 4, 8, 5, 7, 6, 0): 6, (3, 0, 2, 4, 1, 5, 7, 8, 6): 7, (3, 1, 2, 4, 5, 0, 7, 8, 6): 7, (1
{(0, 1, 2, 3, 4, 5, 7, 8, 6): 4, (3, 1, 2, 4, 8, 5, 7, 6, 0): 6, (3, 1, 2, 7, 4, 5, 0, 8, 6): 6, (3, 0, 2, 4, 1, 5, 7, 8, 6): 7, (3
{(1, 0, 2, 3, 4, 5, 7, 8, 6): 5, (3, 1, 2, 4, 8, 5, 7, 6, 0): 6, (3, 1, 2, 7, 4, 5, 0, 8, 6): 6, (3, 0, 2, 4, 1, 5, 7, 8, 6): 7, (3
{(3, 1, 2, 4, 8, 5, 7, 6, 0): 6, (3, 1, 2, 7, 4, 5, 0, 8, 6): 6, (1, 2, 0, 3, 4, 5, 7, 8, 6): 6, (1, 4, 2, 3, 0, 5, 7, 8, 6): 6, (3
{(3, 1, 2, 7, 4, 5, 0, 8, 6): 6, (1, 2, 0, 3, 4, 5, 7, 8, 6): 6, (1, 4, 2, 3, 0, 5, 7, 8, 6): 6, (3, 0, 2, 4, 1, 5, 7, 8, 6): 7, (3
{(1, 2, 0, 3, 4, 5, 7, 8, 6): 6, (1, 4, 2, 3, 0, 5, 7, 8, 6): 6, (3, 0, 2, 4, 1, 5, 7, 8, 6): 7, (3, 1, 2, 4, 5, 0, 7, 8, 6): 7, (3
{(1, 4, 2, 3, 0, 5, 7, 8, 6): 6, (3, 0, 2, 4, 1, 5, 7, 8, 6): 7, (3, 1, 2, 4, 5, 0, 7, 8, 6): 7, (3, 1, 2, 4, 8, 0, 7, 6, 5): 7, (3
```




```
def greedy(initial_state):
  if not is_solvable(initial_state):
    return "The goal state is not reachable."
  # Frontier is a dictionary so that we can sort states by the heuristic function.
  frontier = {tuple(initial_state.flatten()) : manhattan_distance(initial_state)}
  visited = {}
  successor_list = []

  # Count how many nodes are inspected.
  nodes_inspected = 0

  if equal(initial_state, goal_state):
    return tuple([initial_state])
  node = initial_state
```

```
  while not equal(node, goal_state):
    # Record the node as visited.
    visited[tuple(node.flatten())] = True

    nodes_inspected += 1

    # Remove the currently inspected node from the frontier.
    frontier.pop(tuple(node.flatten()))

    # Expand the currently inspected node.
    successors = get_neighbors(node)
    for successor in successors:
      if not array_in_dict(successor, visited) and not array_in_dict(successor, frontier):
        successor_list.append((node, successor))
        frontier[tuple(successor.flatten())] = manhattan_distance(successor)

    # Sort the frontier by total Manhattan distance.
    sort_frontier(frontier)

    # print(frontier)

    # Inspect the next node in the frontier after sorting.
    node = np.array(next(iter(frontier))).reshape(3, 3)

# Trace back the path found from the initial state to the goal state.
path = find_path(initial_state, successor_list)
return (path, len(path), nodes_inspected)
```

```
solvable_state
```

```
array([[1, 8, 2],
       [0, 4, 3],
       [7, 6, 5]])
```

```
greedy(solvable_state)
```

```
((array([[1, 8, 2],
         [0, 4, 3],
         [7, 6, 5]]),
  array([[1, 8, 2],
         [4, 0, 3],
         [7, 6, 5]]),
  array([[1, 8, 2],
         [4, 3, 0],
         [7, 6, 5]]),
  array([[1, 8, 2],
         [4, 3, 5],
         [7, 6, 0]]),
  array([[1, 8, 2],
         [4, 3, 5],
         [7, 0, 6]]),
  array([[1, 8, 2],
         [4, 3, 5],
         [0, 7, 6]]),
  array([[1, 8, 2],
         [0, 3, 5],
         [4, 7, 6]]),
  array([[1, 8, 2],
         [3, 0, 5],
         [4, 7, 6]]),
  array([[1, 0, 2],
         [3, 8, 5],
         [4, 7, 6]]),
  array([[0, 1, 2],
         [3, 8, 5],
         [4, 7, 6]]),
```

```
 array([[3, 1, 2],
        [0, 8, 5],
        [4, 7, 6]]),
 array([[3, 1, 2],
        [4, 8, 5],
        [0, 7, 6]]),
 array([[3, 1, 2],
        [4, 8, 5],
        [7, 0, 6]]),
 array([[3, 1, 2],
        [4, 8, 5],
        [7, 6, 0]]),
 array([[3, 1, 2],
        [4, 8, 0],
        [7, 6, 5]]),
 array([[3, 1, 2],
        [4, 0, 8],
        [7, 6, 5]]),
 array([[3, 1, 2],
        [0, 4, 8],
        [7, 6, 5]]),
 array([[3, 1, 2],
        [7, 4, 8],
        [0, 6, 5]]),
 array([[3, 1, 2],
        [7, 4, 8],
        [6, 0, 5]]),
 array([[3, 1, 2],
        [7, 4, 8],
        [6, 5, 0]]),
```

```
 array([[3, 1, 2],
        [7, 4, 0],
        [6, 5, 8]]),
 array([[3, 1, 2],
        [7, 0, 4],
        [6, 5, 8]]),
 array([[3, 1, 2],
        [0, 7, 4],
        [6, 5, 8]]),
 array([[3, 1, 2],
        [6, 7, 4],
        [0, 5, 8]]),
 array([[3, 1, 2],
        [6, 7, 4],
        [5, 0, 8]]),
 array([[3, 1, 2],
        [6, 0, 4],
        [5, 7, 8]]),
 array([[3, 1, 2],
        [6, 4, 0],
        [5, 7, 8]]),
 array([[3, 1, 2],
        [6, 4, 8],
        [5, 7, 0]]),
 array([[3, 1, 2],
        [6, 4, 8],
        [5, 0, 7]]),
 array([[3, 1, 2],
        [6, 4, 8],
        [0, 5, 7]]),
```

```
 array([[3, 1, 2],
        [0, 4, 8],
        [6, 5, 7]]),
 array([[3, 1, 2],
        [4, 0, 8],
        [6, 5, 7]]),
 array([[3, 1, 2],
        [4, 5, 8],
        [6, 0, 7]]),
 array([[3, 1, 2],
        [4, 5, 8],
        [6, 7, 0]]),
 array([[3, 1, 2],
        [4, 5, 0],
        [6, 7, 8]]),
 array([[3, 1, 2],
        [4, 0, 5],
        [6, 7, 8]]),
 array([[3, 1, 2],
        [0, 4, 5],
        [6, 7, 8]]),
 array([[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]])),
38,
324)
```

(solution is 38 moves long, and 324 nodes were inspected)

```
unsolvable_state
```

```
array([[8, 1, 2],
       [0, 4, 3],
       [7, 6, 5]])
```

```
greedy(unsolvable_state)
```

```
'The goal state is not reachable.'
```

```
# This can be run to create a different initial state each time.
initial_state = np.random.permutation(numbers).reshape(3, 3)
initial_state
```

```
bfs(initial_state)
```

```
greedy(initial_state)
```

Trials

We create a random initial state, that is, a random permutation of the numbers `0, 1, 2, ..., 8` arranged in a 3 by 3 numpy array.

```
# This can be run to create a different initial state each time.
initial_state = np.random.permutation(numbers).reshape(3, 3)
initial_state
```

```
array([[5, 7, 2],
       [4, 6, 3],
       [8, 0, 1]])
```

You can run both search algorithms and compare their performance. When we performed 20 trials, all of them (where the initial state was solvable) had similar results.

That is, BFS gave shorter solutions (paths to the goal state), but inspected a lot more nodes (in the hundred-thousands), while the greedy algorithm gave solutions of longer or equal length, but inspected way less nodes (in the hundreds).

This is because, in the search tree, BFS searches horizontally by "level" or distance from the initial state. So, when it stumbles upon a goal state, the path it took is guaranteed to be of minimum distance from the initial state. But, as it is an uninformed search strategy, it does not find the goal state efficiently, and so it ends up inspecting and expanding many nodes.

The greedy algorithm, on the other hand, searches by total manhattan distance, so it is not necessarily concerned if it "strays too far" from the initial state. But, since the total Manhattan distance is an appropriate representation of "how far" a state is from the goal state, the search is

(You can watch this recording in the video presentation)

Breadth-First Search

Greedy Best-First Search

26, 372639)

Length of Solution

Nodes Inspected

52, 261)

- Shorter solution
- More nodes inspected (longer computing time)

- Longer solution
- Less nodes inspected (shorter computing time)

Thank you!

Try our code!