8 puzzle A* solver

- Name: Hermes Esínola González
- Date: February 07 2019
- Assignment name: A* Informed search
- Course Name: Intelligent System
- School ID: A01631677

Description

The 8-puzzle is a square board with 9 positions, filled by 8 numbered tiles and one gap. At any point, a tile adjacent to the gap can be moved into the gap, creating a new gap position. In other words the gap can be swapped with an adjacent (horizontally and vertically) tile. The objective in the game is to begin with an arbitrary configuration of tiles, and move them so as to get the numbered tiles arranged in ascending order either running around the perimeter of the board or ordered from left to right, with 1 in the top left-hand position.

This program takes a board configuration and returns:

- The **cost** of the path to the solution (number of moves).
- The **moves** made to get to the solution.
- **Number of visited** nodes at the end.
- **Running time** in seconds.
- Used **memory** (assuming each node requires only 72 bytes).

Code

```
# state.py
class State:
    def __init__(self, state: list, parent=None, move: str=None, depth=0,
cost=0, key=0):
        self.state = state
        self.parent = parent
        self.move = move
        self.depth = depth
        self.cost = cost
        self.key = key
        if self.state:
            self.id = ''.join(str(e) for e in self.state)

    def __eq__(self, other):
        return self.id == other.id

    def __lt__(self, other):
        return self.cost < other.cost

    def __hash__(self):
        return int(self.id)
```

```
    def __repr__(self):
        return str(self.state)

# helpers.py
board_len, board_side = 9, 3
target_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]

def read_board_input(conf: str, separator: str) -> list:
    initial_state = [int(x) for x in conf.split(separator)]
    if board_len != len(initial_state):
        'board must be 9 comma separated numbers'
    return initial_state

def read_file_input(path: str, separator: str) -> list:
    with open(path, 'r') as board_file:
        lines = board_file.read().splitlines()
        return read_board_input(separator.join(lines), separator)

def memoize(fun):
    values = dict()
    def m(val):
        if val in values:
            return values[val]
        else:
            values[val] = fun(val)
            return values[val]
    return m

def swap(lst: list, i1, i2) -> list:
    tmp = lst[i1]
    lst[i1] = lst[i2]
    lst[i2] = tmp
    return lst

def move(state: list, direction: str) -> list:
    # 0 represents the empty spot
    _state = state[:]
    empty_space = _state.index(0)
    if direction == 'up':
        if empty_space not in range(0, board_side):
            return swap(_state, empty_space - board_side, empty_space)
        else:
            return None
    if direction == 'down':
        if empty_space not in range(board_len - board_side, board_len):
            return swap(_state, empty_space + board_side, empty_space)
        else:
            return None
    if direction == 'left':
        if empty_space not in range(0, board_len, board_side):
            return swap(_state, empty_space - 1, empty_space)
        else:
            return None
    if direction == 'right':
```

```
        if empty_space not in range(board_side - 1, board_len,
board_side):
            return swap(_state, empty_space + 1, empty_space)
        else:
            return None

# a_star.py
from state import State
import heapq
from helpers import move, target_state, board_side, memoize

find_target_index = memoize(target_state.index)

def calc_cost(board_conf: list, depth):
    total_cost = 0
    for idx, piece in enumerate(board_conf):
        if piece == 0:
            continue
        target_idx = find_target_index(piece)
        v_cost = abs(idx // board_side - target_idx // board_side)
        h_cost = abs((idx % board_side) - (target_idx % board_side))
        total_cost += v_cost + h_cost
    return total_cost + depth

def init_struct(initial_state: State):
    h = [initial_state]
    heapq.heapify(h)
    return h

def visit(heap):
    return heapq.heappop(heap)

def expand(node: State) -> list:
    neighbors = []
    for move_dir in ['left', 'right', 'up', 'down']:
        board_conf = move(node.state, move_dir)
        if not board_conf:
            continue
        cost = calc_cost(board_conf, node.depth+1)
        # Create a list of neighbors generating all possible moves from
the current state
        neighbors.append(State(board_conf, node, move_dir, node.depth+1,
cost))
    return neighbors

def add(heap, node: State):
    heapq.heappush(heap, node)

# drivers.py
# import bfs
import a_star

methods = {
    # 'bfs': (
```

```
    #     bfs.init_struct, # create structure function
    #     bfs.visit, # visit function
    #     bfs.expand, # expand function
    #     bfs.add, # add to structure
    # ),
    'a_star': (
        a_star.init_struct, # create structure function
        a_star.visit, # visit function
        a_star.expand, # expand function
        a_star.add, # add to structure
    ),
}

# 8_puzzle.py
#!/usr/local/bin/python3
# coding=utf-8

import timeit
from pprint import pprint
from state import State
from argparse import ArgumentParser
from drivers import methods
from helpers import read_board_input, read_file_input, target_state

seen = set()

# for result output
goal_node: State = None
nodes_visited = 0
nodes_created = 0
debug = False

def solve(initial_state, method_name):
    global seen, goal_node, nodes_visited, nodes_created
    init_struct, visit, expand, add = methods[method_name]

    # keep track of which nodes we have visited and which we have to visit
    struct = init_struct(State(initial_state))
    nodes_created += 1
    level = 0
    while struct:
        if debug:
            print('level', level)
            pprint(struct)
        node = visit(struct)
        seen.add(node)

        nodes_visited += 1
        if node.state == target_state:
            # mark the node that contains the solution
            goal_node = node
            return struct

        neighbors = expand(node)
```

```
            nodes_created += len(neighbors)

            for neighbor in neighbors:
                if neighbor not in seen:
                    add(struct, neighbor)
                    seen.add(neighbor)
            level += 1

    def get_path():
        curr = goal_node
        path = [goal_node.move]
        while curr.parent:
            curr = curr.parent
            path.insert(0, curr.move)
        return path[1:]

    def output(time):
        if not goal_node:
            print('Solution not found')
        else:
            print('Cost of path:', goal_node.depth)
            print('Path to goal:', ', '.join(get_path()))
        print('# visited nodes:', nodes_visited)
        print('Running time (seconds):', format(time, '.8f'))
        print('Used memory:', 'If each node requires only 72 bytes,
    {}'.format(nodes_created * 72))

    def main():
        global debug
        parser = ArgumentParser()
        input_group = parser.add_mutually_exclusive_group()
        input_group.add_argument('-f', '--file', help='set input file path')
        input_group.add_argument('-b', '--board', help='set board input text')
        parser.add_argument('-s', '--separator', help='set separator for each
    element of board input')
        parser.add_argument('-m', '--method', help="'bfs' or 'a_star'")
        parser.add_argument('-d', '--debug', help='enable debugging logs',
    action='store_true')
        parser.set_defaults(separator=' ', method='a_star')
        input_group.set_defaults(file='board.txt')
        args = parser.parse_args()
        debug = args.debug
        initial_state = []
        if args.board:
            initial_state = read_board_input(args.board, args.separator)
        else:
            initial_state = read_file_input(args.file, args.separator)
        start = timeit.default_timer()
        solve(initial_state, args.method)
        stop = timeit.default_timer()
        output(stop-start)

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

Instructions

Here's a copy of help command `./8_puzzle.py -h`:

```
usage: 8_puzzle.py [-h] [-f FILE | -b BOARD] [-s SEPARATOR] [-m METHOD] [-
d]

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  set input file path
  -b BOARD, --board BOARD
                        set board input text
  -s SEPARATOR, --separator SEPARATOR
                        set separator for each element of board input
  -m METHOD, --method METHOD
                        'bfs' or 'a_star'
  -d, --debug           enable debugging logs
```

Execute the python script, by default it reads a file named `board.txt`. You can change the file name by providing it with the `-f` flag, or, set the input directly by using the flag `-b`.

Input

The input must be a list of 9 numbers, separated by a space (or the separator indicated by the -s parameter). The input should include a `0`, which indicates the gap of the puzzle.

Some examples:

```
python3 8_puzzle.py -b '7 2 4 5 0 6 8 3 1'

python3 8_puzzle.py -f ./inputs/other_board.txt -s ';'

python3 8_puzzle.py -s ',' -b '7, 2, 4, 5, 0, 6, 8, 3, 1' -m 'a_star'
```

Test

```
➜ python3 8_puzzle.py -f ./board.txt -m a_star
Cost of path: 20
Path to goal: down, right, up, left, left, up, right, right, down, left,
down, left, up, right, up, left, down, right, right, down
# visited nodes: 282
Running time (seconds): 0.01112909
Used memory: If each node requires only 72 bytes, 53712
```