P8.py

8 puzzle in python

- Look at a simple implementation of an eight puzzle solver in python
- [p8.py]
- Solve using A* with three different heuristics
  - NIL: h = 1
  - OOP: h = # of tiles out of place
  - MHD: h = sum of Manhattan distance between each tile's current & goal positions
- All three are admissible

What must we model?

- A state
- Goal test
- Actions
- Result of doing action in state
- Heuristic function

Representing states and actions

- Represent state as string of nine characters with blank as *

  E.g.: s = '1234*5678'

1	2	3
4	*	5
6	7	8

- Position of blank in state S is

```
> s.index('*')
4
```

- Represent an action as one of four possible ways to move the blank:

```
up down right left
```

Legal Actions

def actions8(s):

returns list of possible actions in state s

Possible moves are mapped to position (not the tile itself)

```
action_table = {
  0:['down', 'right'],
  1:['down', 'left', 'right'],
  2:['down', 'left'],
  3:['up', 'down', 'right'],
  4:['up', 'down', 'left', 'right'],
  5:['up', 'down', 'left'],
  6:['up', 'right'],
  7:['up', 'left', 'right'],
  8:['up', 'left'] }
return action_table[s.index('*')]
```

0	1	2
3	4	5
6	7	8

Function maps a **position** to a list of **possible moves** for a tile in that position

Result of action A on state S

```
def result8(S, A):
    blank = S.index('*')    # blank position
    if A == 'up':
        swap = blank - 3
        return S[0:swap] + '*' + S[swap+1:blank] + S[swap] + S[blank+1:]
    elif A == 'down':
        swap = blank + 3
        return S[0:blank] + S[swap] + S[blank+1:swap] + '*' + S[swap+1:]
    elif A == 'left':
        swap = blank - 1
        return S[0:swap] + '*' + S[swap] + S[blank+1:]
    elif A == 'right':
        swap = blank + 1
        return S[0:blank] + S[swap] + '*' + S[swap+1:]
    raise ValueError('Unrecognized action: ' + A)
```

Heuristic functions

```
class P8_h1(P8):
    """ Eight puzzle using a heuristic function that counts number
    of tiles out of place"””
    name = 'Out of Place Heuristic (OOP)’

 def h(self, node):
      ""”OOP 8 puzzle heuristic: number of tiles 'out of place'
      between a node's state and the goal"""
      mismatches = 0
      for (t1, t2) in zip(node.state, self.goal):
          if  t1 != t2: mismatches =+ 1
      return mismatches
```

Path_cost method

Since path cost is just the number of steps, we can use the default version defined in Problem

```
def path_cost(self, c, state1, action, state2):
    """Return cost of a solution path that arrives at state2 from
    state1 via action, assuming cost c to get up to state1. If problem
    is such that the path doesn't matter, this function will only look at
    state2.  If the path does matter, it will consider c and maybe state1
    and action. The default method costs 1 for every step in the path."""
    return c + 1
```

How can we test this?

- Need solvable test problems that aren’t too hard
  - Recall that the state space has two disjoint sets!
  - Generating a random initial & goal states will result in no possible solution 50% of the time
- Idea: take a random walk of N steps from the goal
  - Resulting state is solvable in $\leq$ N moves
  - Ensure random walk has no loops for a better test
- What metrics can we use to compare heuristics?
  - # of states generated, # of states expanded, effective branching factor (efb), and run time

Example

- Generate tests of different distances from *12345678

  15 steps: 4*3275681 => *12345678

  19 steps: 4258361*7 => *12345678

- Solve using three heuristics, collect data

heuristic used	solution length	states generated	successors computed	effective branching fac.	runtime in seconds
NIL	15	14,386	5,173	1.77	5.47145
OOP	15	761	283	1.46	0.02097
MHD	15	87	31	1.26	0.00086
NIL	19	78,872	28,567	1.72	159.1051
OOP	19	3,906	1,457	1.47	0.4217
MHD	19	499	185	1.32	0.1238

P8 Problem on Colab

- P8.ipynb which uses p8.py and search.py