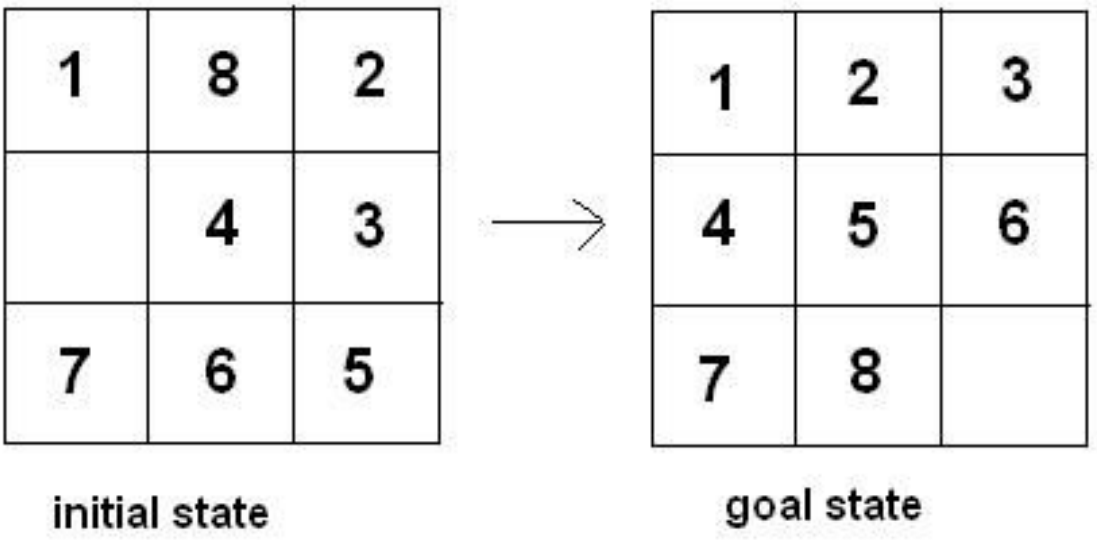
5.Solving 8 Puzzle game using steepest ascent hill climbing as heuristics.

Introduction

An 8 puzzle is a simple game consisting of a 3 x 3 grid (containing 9 squares). Tiles are numbered uniquely from 1 to 8 and One of the squares is empty. The object is to move to squares around into different positions and having the numbers displayed in the "goal state". Any given configuration can be thought of as an unsolved initial state of the "3 x 3" 8 puzzle.Two possible states of the 8-puzzle are shown in figure below.



Rules for solving the puzzle

Instead of moving the tiles in the empty space we can visualize moving the empty space in place of the tile, basically swapping the tile with the empty space. The empty space can only move in four directions viz.,

1. Up
2.Down
3. Right or
4. Left

The empty space cannot move diagonally and can take only one step at a time (i.e. move the empty space one position at a time).

Solution using STEEPEST ASCENT HILL CLIMBING

- **Steepest Ascent Hill climbing** search algorithm is one of the simplest algorithms which falls under local search and optimization techniques. It first examines all the neighboring nodes and then selects the node closest to the solution state as of next node.This solution may not be the global optimal maximum.

- **Algorithm for Steepest Ascent Hill Climbing:**

  Step 1 : *Evaluate the initial state. If it is goal state then exit else make the current state as initial state*

  Step 2 : *Repeat these steps until a solution is found or current state does not change*

  i. Let 'target' be a state such that any successor of the current state will be better than it;

  ii. for each operator that applies to the current state

  a. apply the new operator and create a new state

  b. evaluate the new state

  c. if this state is goal state then quit else compare with 'target'

  d. if this state is better than 'target', set this state as 'target'

  e. if target is better than current state set current state to Target

  Step 3 *: Exit*

Python Code Implementation

```
import sys

import random

import time

class P8(Problem):

    """A state is represented as a 9-character string containing
the

    digits 1-8 for tiles and '*' for the blank."""
```

```
    name = 'Null'

    def __init__(self, goal='*12345678', initial=None, N=20):
        self.goal = goal
        if initial:
            self.initial = initial
        else:
            self.initial = random_state(goal, successor8, N)

    def successor(self, state):
        return successor8(state)

    def h(self, node):
        """Heuristic for 8 puzzle: returns 0"""
        return 0

class P8_h1(P8):

    """Eight puzzle using a heuristic that counts the number of
tiles out of place"""

    name = 'oop'

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

```
class P8_h2(P8):

    name = 'mhd'

    def h(self, node):
        """Heuristic for 8 puzzle: returns sum for each tile of
manhattan
        distance between it's position in node's state and goal"""
        sum = 0
        for c in '12345678':
            sum =+ mhd(node.state.index(c), self.goal.index(c))
        return sum


def mhd(n, m):
    """Given indices in a 9 character strings corresponding to a
3x3 array,
       return mhd betwen the two positions"""
    x1,y1 = coordinates[n]
    x2,y2 = coordinates[m]
    return abs(x1-x2) + abs(y1-y2)
    #return abs((n - m) / 3) + abs( ((n / 3) % 3) - ((m / 3) % 3))

coordinates = {0:(0,0), 1:(1,0), 2:(2,0),
               3:(0,1), 4:(1,1), 5:(2,1),
               6:(0,2), 7:(1,2), 8:(2,2)}


def random_state(S, successor_function, N=20):
    """Returns a state reached by N random actions generated by
       successor_function starting from state S"""
    for i in range(N):
        S = random.choice(successor_function(S))[1]
```

```
    return S


def successor8(S):
    """Returns a list of successors of state S for the eight
puzzle.
       A state is represented by a nine character string with *
       representing the blank and 1..8 representing the digits."""


    # index of the blank
    blank = S.index('*')


    succs = []


    # UP: if blank not on top row, swap it with tile above it
    if blank > 2:
        swap = blank - 3
        succs.append(('U', S[0:swap] + '*' + S[swap+1:blank] +
S[swap] + S[blank+1:]))


    # DOWN: If blank not on bottom row, swap it with tile below it
    if blank < 6:
        swap = blank + 3
        succs.append(('D', S[0:blank] + S[swap] + S[blank+1:swap]
+ '*' + S[swap+1:]))


    # LEFT: If blank not in left column, swap it with tile to the
left
    if blank % 3 > 0:
        swap = blank - 1
        succs.append(('L', S[0:swap] + '*' + S[swap] +
S[blank+1:]))


    # RIGHT: If blank not on right column, swap it with tile to
the right
    if blank % 3 < 2:
```

```
        swap = blank + 1
        succs.append(('R', S[0:blank] + S[swap] + '*' +
S[swap+1:]))

    return succs



def printsoln(goal):
    """shows solution to 8 puzzle"""
    # path is list of states from initial to goal
    path = goal.path()
    path.reverse()
    initial = path[-1]
    # print the solution
    print "%s steps from %s to %s" % (len(path), initial.state,
goal.state)
    for n in path:
        print_state(n)

def print_state(n):
    """Print the action and resulting state"""
    a = n.action
    s = n.state
    print "%s\t%s\n\t%s\n\t%s\n" % (a,s[0:3],s[3:6],s[6:9])

def solve(pr=True, n=10):
    """Solves a random 8 puzzle problem and prints info"""

    print "Problems using %s random steps from goal" % (n,)
    s = random_state("*12345678", successor8, n)

    for p in [P8(initial=s), P8_h1(initial=s), P8_h2(initial=s)]:
        ip = InstrumentedProblem(p)
```

```
        if pr: print 'Using %s from %s to %s' % (p.name,
p.initial, p.goal)
        begin_time = time.time()
        solution = astar_search(ip)
        end_time =  time.time()
        if pr: print '  %s states, %s successors, %s goal tests,
%9.6f sec' % (ip.states, ip.succs, ip.goal_tests, end_time -
begin_time)
        if solution:
            # path is list of states from initial to goal
            path = solution.path()
            path.reverse()
            # print the solution length
            if pr: print "  Solution of length %s" % (len(path),)
        else:
            if pr: print 'No solution found :-('



def main():
    for i in [10,20,30,40]:
        solve()

# if called from the command line, call main()
if __name__ == "__main__":
    main()
```