[Final Version]Assignment 1 CS152

October 17, 2017

1 The N-Puzzle

Zitong Mao

Original File at: https://github.com/ZitongMao/n_puzzle_astar.git ## Introduction ##

Python Version: Python 2

This code solves the N-Puzzle problem by using A-star searching algorithm with three admissible heuristics: Misplaced Tiles, Manhattan Distance, and Linear Conflict + Manhattan Distance.

The code strictly follows the evaluation rules:

val = heuristics[0](state)

steps, frontierSize, err = solvePuzzle(n, state, heuristic, verbose)

Input arguments:

—n - the puzzle dimension (i.e. n x n board)

—state - 2D Matrix. Example [[7 2 4],[5 0 6],[8 3 1]]

—heuristic - Misplaced Tiles, Manhattan Distance, and Linear Conflict + Manhattan Distance.

—verbose - a boolean value that indicates whether or not to print the solution

Output arguments:

—steps - the number of steps required to reach the goal state from the initial state

—frontierSize - the maximum size of the frontier during the search

—err - an error code. Error Code "0": No Error; Error Code "-1": Input Format Error; Error Code "-2": Puzzle Mathematically Unsolvable.

1.1 Section: Cantor Expansion

Cantor Expansion transforms a list into a integer using factorial. This helps us identify whether a state has already been visited in the A-star searching process, so we don't move the blocks back and forth. This specific Cantor Expansion takes advantage of the python class and makes "encode" and "decode" more intuitive.

```
In [27]: import heapq
         import numpy as np

         # Code based on Git:https://github.com/Chrstm/Cantor-expansion by JHSN on 2017.2.8
         class Cantor:
             def __init__(self, max_n = 9):
                 self.max_n = max_n
                 self.factorial = [1] * (max_n + 1)
                 for i in range(2, max_n + 1):
                     self.factorial[i] = self.factorial[i - 1] * i

             def encode(self, a):
                 n = len(a)
                 x = 0
                 unshowed = [False] + [True] * n
                 for i in range(n):
                     k = a[i] + 1
                     if k > n or k < 1 or not unshowed[k]:
                         return -1
                     unshowed[k] = False
                     x += sum(unshowed[: k]) * self.factorial[n - i - 1]
                 return x

             def decode(self, x):
                 a = []
                 n = self.max_n
                 p = [k for k in range(n)]
                 for i in range(n):
                     k = x / self.factorial[n - i - 1]
                     x %= self.factorial[n - i - 1]
                     a.append(p.pop(k))
                 return a
```

1.2 Section: Memoization

Memoization using decorator in Python. The decorators are added before the heuristic functions to store knowledge of the heuristic function from previous search runs. This would save time for us for the next run.

```
In [28]: class Memoize:
             def __init__(self, f):
                 self.f = f
                 self.memo = {}
             def __call__(self, *args):
                 nsquared = int(len(args[0]))
                 #print nsquared
                 value = Cantor(nsquared).encode(args[0])
                 if not value in self.memo:
                     self.memo[value] = self.f(*args)
                 return self.memo[value]
```

1.3 Section: PuzzleNode

Misplaced Tiles, Manhattan Distance, and Linear Conflict + Manhattan Distance.

```
In [29]: class PuzzleNode:
             def __init__(self, matrix, step, parent, n):
```

```
            self.n = n
            #use cantor
            cantor = Cantor(max_n = self.n**2)
            self.matrix = cantor.encode(matrix)
            #identify empty tile
            self.empty_tile = matrix.index(0)
            self.step = step
            #backtrack parent
            self.parent = parent

        #help us print out results
        def __str__(self):
            cantor = Cantor(max_n = self.n**2)
            #reshape to 2D
            mat = np.reshape(cantor.decode(self.matrix), (-1, self.n))
            #print mat
            output=""
            for i in range(self.n):
                for j in range(self.n):
                    output=output+str(mat[i][j])+" "
                output=output+"\n"
            return output
```

1.4 Section: Heuristic Functions

```
In [30]: @Memoize
         def misplaced_tile(state):
             #both 2D or 1D input will generate good results
             np_state = np.array(state)
             if len(np_state.shape) == 2:
                 state = reduce(lambda x,y :x+y, state)
             n = int(len(state)**0.5)
             #generate the correct goal board
             goal = [i for i in range(n**2)]
             #heuristic value is the misplaced number
             misplaced = 0
             for i in range(n**2):
                 if state[i] != goal[i]: misplaced += 1
             return misplaced

         @Memoize
         def manhattan_dist(state):
             np_state = np.array(state)
             if len(np_state.shape) == 2:
                 state = reduce(lambda x,y :x+y, state)
             n = int(len(state)**0.5)
             #reshape to 2D
             cur_status = np.reshape(state, (-1, n))
```

```
        actual_pos = [[i / n, i % n] for i in range(n**2)]
        sum = 0
        #manhattan distance
        for i in range(n):
            for j in range(n):
                sum += abs(actual_pos[cur_status[i][j]][0] - i) + abs(actual_pos[cur_status
        return sum


    @Memoize
    def linear_conflict(state):

        np_state = np.array(state)
        if len(np_state.shape) == 2:
            state = reduce(lambda x,y :x+y, state)
        n = int(len(state)**0.5)
        #based on Manhattan Distance, adding penalty for conflic
        cur_status = np.reshape(state, (-1, n))
        actual_pos = [[i / n, i % n] for i in range(n**2)]
        sum = 0
        for i in range(n):
            for j in range(n):
                sum += abs(actual_pos[cur_status[i][j]][0] - i) + abs(actual_pos[cur_status
                if cur_status[i][j] != 0 and cur_status[i][j] in range(i * n, (i + 1) * n):
                    for k in range(j + 1, n):
                        if cur_status[i][k] != 0 and cur_status[i][k] in range(i * n, (i +
                            if cur_status[i][j] > cur_status[i][k]:
                                #conflic penalty
                                sum = sum + 2

        return sum
```

1.5 Section: solvePuzzle

```
In [31]: def solvePuzzle(n, state, heuristic, verbose):
             #test whether or not the state provided is of the correct size and format
             #and contains every number from 0 to n^2-1 precisely once
             np_state = np.array(state)
             one_dimension_check = reduce(lambda x,y :x+y, state)

             #######################################################################
             #Error Code -1
             #check dimension, expect 2D
             if len(np_state.shape) != 2:
                 return (0, 0, -1)

             #check size
             if len(one_dimension_check) != n**2:
```

```
    return (0, 0, -1)

#check if it contains every number
correctlist = range(n**2)
sortedlist = sorted(one_dimension_check)
if sortedlist != correctlist:
    return (0, 0, -1)
#########################################################################
#Error Code -2
inversions = 0
inversion_count = reduce(lambda x,y :x+y, state)
inversion_count.remove(0)
for i in range(len(inversion_count)):
    for j in range(i,len(inversion_count)):
        if inversion_count[j] < inversion_count[i]:
            inversions += 1
#If the grid width is odd, then the number of inversions in a solvable situation is
if n % 2 == 1:
    if inversions % 2 != 0:
        return(0, 0, -2)
#If the grid width is even, and the blank is on an even row counting from the botto
#then the number of inversions in a solvable situation is odd.
else:
    if (one_dimension_check.index(0)/n)%2 == 0:
        if inversions %2 != 1:
            return(0, 0, -2)

    #If the grid width is even, and the blank is on an odd row counting from the bo
    #then the number of inversions in a solvable situation is even.
    else:
        if inversions %2 != 0:
            return(0, 0, -2)



#########################################################################

cantor = Cantor(max_n = n**2)
#goal not reached
reached = False
#counter for frontier size
max_frontier_size = 0
#use dictionary to record if a node is visited
visited = {}
priority_queue = []
#transform 2D matrix to 1D
state = reduce(lambda x,y :x+y, state)
```

```
#astar
curNode = PuzzleNode(state, 0, None, n)
#using priority queue, adding heuristic value into consideration
heuristic_value = heuristic(cantor.decode(curNode.matrix))
visited[curNode.matrix] = True
heuristic_value = heuristic(cantor.decode(curNode.matrix))
#using heap as the priority queue
heapq.heappush(priority_queue, (heuristic_value + curNode.step, curNode))
counter=0
#four movements
movement = [[0, 1], [0, -1], [-1, 0], [1, 0]]
#compare
while priority_queue!=[]:
    counter+=1
    if len(priority_queue) > max_frontier_size:
        max_frontier_size = len(priority_queue)
    newElement = heapq.heappop(priority_queue)[1]
    emptyPos=newElement.empty_tile

    moves=[1, -1, -n, n]
    for direction in range(4):

        newPos=emptyPos + moves[direction]
        if newPos >= 0 and newPos < n**2:
            #trace back the parent
            boardConfig = cantor.decode(newElement.matrix)
            newBoard = boardConfig[:]
            newBoard[emptyPos] = newBoard[newPos]
            newBoard[newPos] = 0
            nextNode = PuzzleNode(newBoard,newElement.step+1,newElement,n)

            if nextNode.matrix not in visited:
                heuristic_value = heuristic(cantor.decode(nextNode.matrix))
                #if heuristic_value == 0:
                #this adds more generality.
                #according to Cantor Expansion 0 represents a solved puzzle
                if newElement.matrix == 0:
                    #print
                    if verbose:
                        liss=[newElement]
                        while True:
                            newElement=newElement.parent
                            if newElement==None:
                                break
                            else:
                                liss.append(newElement)
                        for i in reversed(liss):
                            print i
```

```
                                        print "~~~~~~~"
                                    return (nextNode.step, max_frontier_size, 0)
                                visited[nextNode.matrix] = True
                                heapq.heappush(priority_queue, (heuristic_value + nextNode.step, ne
                return "Error, Manual Check"
```

1.6 Section: Test Function

As we can see from the results, Linear Conflict outperformed the others in all the three test cases by having a smaller maximum size of the frontier. This is reasonable, because LC always returns higher heuristic value by penalizing conflicts.

```
In [34]: heuristics = [misplaced_tile, manhattan_dist, linear_conflict]
         heuristic_names = ['misplaced_tile', 'manhattan_dist', 'linear_conflict']
         def compareHeuristic(data):
             for curHeuristic in range(len(heuristics)):
                 steps, frontierSize, err = (solvePuzzle(3, data, heuristics[curHeuristic], Fals
                 print "The input state is:", data
                 print "We are using: ", heuristic_names[curHeuristic]
                 print "The number of steps required to reach the goal state from the initial st
                 print "The maximum size of the frontier during the search is: ", frontierSize
                 print "Error Code: ", err
                 print " "

         data = [[[5,7,6],[2,4,3],[8,1,0]], [[7,0,8],[4,6,1],[5,3,2]], [[2,3,7],[1,8,0],[6,5,4]]
         for i in range(len(data)):
             compareHeuristic(data[i])
             print "=================================================="
```

```
The input state is: [[5, 7, 6], [2, 4, 3], [8, 1, 0]]
We are using:  misplaced_tile
The number of steps required to reach the goal state from the initial state is:  23
The maximum size of the frontier during the search is:  18717
Error Code:  0

The input state is: [[5, 7, 6], [2, 4, 3], [8, 1, 0]]
We are using:  manhattan_dist
The number of steps required to reach the goal state from the initial state is:  27
The maximum size of the frontier during the search is:  5592
Error Code:  0

The input state is: [[5, 7, 6], [2, 4, 3], [8, 1, 0]]
We are using:  linear_conflict
The number of steps required to reach the goal state from the initial state is:  25
The maximum size of the frontier during the search is:  4014
Error Code:  0

==================================================
```

```
The input state is: [[7, 0, 8], [4, 6, 1], [5, 3, 2]]
We are using:  misplaced_tile
The number of steps required to reach the goal state from the initial state is:  24
The maximum size of the frontier during the search is:  22348
Error Code:  0

The input state is: [[7, 0, 8], [4, 6, 1], [5, 3, 2]]
We are using:  manhattan_dist
The number of steps required to reach the goal state from the initial state is:  26
The maximum size of the frontier during the search is:  8969
Error Code:  0

The input state is: [[7, 0, 8], [4, 6, 1], [5, 3, 2]]
We are using:  linear_conflict
The number of steps required to reach the goal state from the initial state is:  26
The maximum size of the frontier during the search is:  7262
Error Code:  0

=================================================
The input state is: [[2, 3, 7], [1, 8, 0], [6, 5, 4]]
We are using:  misplaced_tile
The number of steps required to reach the goal state from the initial state is:  18
The maximum size of the frontier during the search is:  2557
Error Code:  0

The input state is: [[2, 3, 7], [1, 8, 0], [6, 5, 4]]
We are using:  manhattan_dist
The number of steps required to reach the goal state from the initial state is:  18
The maximum size of the frontier during the search is:  400
Error Code:  0

The input state is: [[2, 3, 7], [1, 8, 0], [6, 5, 4]]
We are using:  linear_conflict
The number of steps required to reach the goal state from the initial state is:  18
The maximum size of the frontier during the search is:  279
Error Code:  0

=================================================
```

1.7 Section: Special Cases

Works well! Error Code "-1": Input Format Error; Error Code "-2": Puzzle Mathematically Unsolvable.

```
In [35]: def compareHeuristic(data):
             for curHeuristic in range(len(heuristics)):
                 steps, frontierSize, err = (solvePuzzle(3, data, heuristics[curHeuristic], Fals
                 print "The input state is:", data
                 print "We are using: ", heuristic_names[curHeuristic]
                 print "The number of steps required to reach the goal state from the initial st
                 print "The maximum size of the frontier during the search is: ", frontierSize
                 print "Error Code: ", err
                 print " "

         data = [[[2,3,3,7],[1,8],[6,5,4]], [[2,3,4],[1,8,0],[6,5,4]], [[2,3,7,9],[1,8,0],[6,5,4
                 , [[7,0,2],[8,5,3],[6,4,1]], [[1,0,3],[2,4,5],[6,7,8]]]
         for i in range(len(data)):
             compareHeuristic(data[i])
             print "=================================================="

The input state is: [[2, 3, 3, 7], [1, 8], [6, 5, 4]]
We are using:  misplaced_tile
The number of steps required to reach the goal state from the initial state is:  0
The maximum size of the frontier during the search is:  0
Error Code:  -1

The input state is: [[2, 3, 3, 7], [1, 8], [6, 5, 4]]
We are using:  manhattan_dist
The number of steps required to reach the goal state from the initial state is:  0
The maximum size of the frontier during the search is:  0
Error Code:  -1

The input state is: [[2, 3, 3, 7], [1, 8], [6, 5, 4]]
We are using:  linear_conflict
The number of steps required to reach the goal state from the initial state is:  0
The maximum size of the frontier during the search is:  0
Error Code:  -1

==================================================
The input state is: [[2, 3, 4], [1, 8, 0], [6, 5, 4]]
We are using:  misplaced_tile
The number of steps required to reach the goal state from the initial state is:  0
The maximum size of the frontier during the search is:  0
Error Code:  -1

The input state is: [[2, 3, 4], [1, 8, 0], [6, 5, 4]]
We are using:  manhattan_dist
The number of steps required to reach the goal state from the initial state is:  0
The maximum size of the frontier during the search is:  0
Error Code:  -1

The input state is: [[2, 3, 4], [1, 8, 0], [6, 5, 4]]
We are using:  linear_conflict
The number of steps required to reach the goal state from the initial state is:  0
The maximum size of the frontier during the search is:  0
```

```
Error Code:  -1

==================================================
The input state is: [[2, 3, 7, 9], [1, 8, 0], [6, 5, 4]]
We are using:  misplaced_tile
The number of steps required to reach the goal state from the initial state is:  0
The maximum size of the frontier during the search is:  0
Error Code:  -1

The input state is: [[2, 3, 7, 9], [1, 8, 0], [6, 5, 4]]
We are using:  manhattan_dist
The number of steps required to reach the goal state from the initial state is:  0
The maximum size of the frontier during the search is:  0
Error Code:  -1

The input state is: [[2, 3, 7, 9], [1, 8, 0], [6, 5, 4]]
We are using:  linear_conflict
The number of steps required to reach the goal state from the initial state is:  0
The maximum size of the frontier during the search is:  0
Error Code:  -1

==================================================
The input state is: [[7, 0, 2], [8, 5, 3], [6, 4, 1]]
We are using:  misplaced_tile
The number of steps required to reach the goal state from the initial state is:  0
The maximum size of the frontier during the search is:  0
Error Code:  -2

The input state is: [[7, 0, 2], [8, 5, 3], [6, 4, 1]]
We are using:  manhattan_dist
The number of steps required to reach the goal state from the initial state is:  0
The maximum size of the frontier during the search is:  0
Error Code:  -2

The input state is: [[7, 0, 2], [8, 5, 3], [6, 4, 1]]
We are using:  linear_conflict
The number of steps required to reach the goal state from the initial state is:  0
The maximum size of the frontier during the search is:  0
Error Code:  -2

==================================================
The input state is: [[1, 0, 3], [2, 4, 5], [6, 7, 8]]
We are using:  misplaced_tile
The number of steps required to reach the goal state from the initial state is:  0
The maximum size of the frontier during the search is:  0
Error Code:  -2

The input state is: [[1, 0, 3], [2, 4, 5], [6, 7, 8]]
```

```
We are using:  manhattan_dist
The number of steps required to reach the goal state from the initial state is:  0
The maximum size of the frontier during the search is:  0
Error Code:  -2

The input state is: [[1, 0, 3], [2, 4, 5], [6, 7, 8]]
We are using:  linear_conflict
The number of steps required to reach the goal state from the initial state is:  0
The maximum size of the frontier during the search is:  0
Error Code:  -2

=================================================
```

Works well for 4 times 4 puzzles as well! Feel free to increase the n value.

```
In [26]: solvePuzzle(4, [[2,3,4,5],[6,7,8,9],[10,11,12,13],[14,15,1,0]], manhattan_dist, False)

Out[26]: (42, 112755, 0)
```