8 puzzle_bfs

```
import copy

import time



def printNode(node):

print(node[0], node[1], node[2])

print(node[3], node[4], node[5])

print(node[6], node[7], node[8])

global nodeNumber

print('Node:', nodeNumber)

print('Depth:', len(node[9:]))

print('Moves:', node[9:])

print('------')

nodeNumber += 1



def checkFinal(node):

if node[:9] == finalNode:

printNode(node)

return True

global insertIndex

if node[:9] not in visitedList:

printNode(node)

stack.insert(insertIndex, node)

insertIndex += 1

visitedList.append(node[:9])
```

```
return False



if __name__ == '__main__':
startNode = [1, 2, 5, 3, 4, 8, 6, 7, 0]
finalNode = [0, 1, 2, 3, 4, 5, 6, 7, 8]

found = False
nodeNumber = 0
visitedList = []
stack = [startNode]
visitedList.append(startNode)
printNode(startNode)
t0 = time.time()

while not found and not len(stack) == 0:
currentNode = stack.pop(0)
blankIndex = currentNode.index(0)
insertIndex = 0
if blankIndex != 0 and blankIndex != 1 and blankIndex != 2:
upNode = copy.deepcopy(currentNode)
upNode[blankIndex] = upNode[blankIndex - 3]
upNode[blankIndex - 3] = 0
upNode.append('up')
found = checkFinal(upNode)
if blankIndex != 0 and blankIndex != 3 and blankIndex != 6 and found == False:
leftNode = copy.deepcopy(currentNode)
leftNode[blankIndex] = leftNode[blankIndex - 1]
leftNode[blankIndex - 1] = 0
```

```
leftNode.append('left')
found = checkFinal(leftNode)
if blankIndex != 6 and blankIndex != 7 and blankIndex != 8 and found == False:
downNode = copy.deepcopy(currentNode)
downNode[blankIndex] = downNode[blankIndex + 3]
downNode[blankIndex + 3] = 0
downNode.append('down')
found = checkFinal(downNode)
if blankIndex != 2 and blankIndex != 5 and blankIndex != 8 and found == False:
rightNode = copy.deepcopy(currentNode)
rightNode[blankIndex] = rightNode[blankIndex + 1]
rightNode[blankIndex + 1] = 0
rightNode.append('right')
found = checkFinal(rightNode)

t1 = time.time()
print('Time:', t1 - t0)
print('------')
```

A star Search

```
GRAPH = {\
'Arad': {'Sibiu': 140, 'Zerind': 75, 'Timisoara': 118},\
'Zerind': {'Arad': 75, 'Oradea': 71},\
'Oradea': {'Zerind': 71, 'Sibiu': 151},\
'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu': 80},\
'Timisoara': {'Arad': 118, 'Lugoj': 111},\
'Lugoj': {'Timisoara': 111, 'Mehadia': 70},\
'Mehadia': {'Lugoj': 70, 'Drobeta': 75},\
'Drobeta': {'Mehadia': 75, 'Craiova': 120},\
'Craiova': {'Drobeta': 120, 'Rimnicu': 146, 'Pitesti': 138},\
'Rimnicu': {'Sibiu': 80, 'Craiova': 146, 'Pitesti': 97},\
'Fagaras': {'Sibiu': 99, 'Bucharest': 211},\
'Pitesti': {'Rimnicu': 97, 'Craiova': 138, 'Bucharest': 101},\
'Bucharest': {'Fagaras': 211, 'Pitesti': 101, 'Giurgiu': 90, 'Urziceni': 85},\
'Giurgiu': {'Bucharest': 90},\
'Urziceni': {'Bucharest': 85, 'Vaslui': 142, 'Hirsova': 98},\
'Hirsova': {'Urziceni': 98, 'Eforie': 86},\
'Eforie': {'Hirsova': 86},\
'Vaslui': {'Iasi': 92, 'Urziceni': 142},\
'Iasi': {'Vaslui': 92, 'Neamt': 87},\
'Neamt': {'Iasi': 87}\ }

def dfs_paths(source, destination, path=None):
"""All possible paths from source to destination using depth-first search
:param source: Source city name
:param destination: Destination city name
:param path: Current traversed path (Default value = None)
```

```
:yields: All possible paths from source to destination
"""
if path is None:
path = [source]
if source == destination:
yield path
for next_node in set(GRAPH[source].keys()) - set(path):
yield from dfs_paths(next_node, destination, path + [next_node])


def ucs(source, destination):
"""Cheapest path from source to destination using uniform cost search
:param source: Source city name
:param destination: Destination city name
:returns: Cost and path for cheapest traversal
"""
from queue import PriorityQueue
priority_queue, visited = PriorityQueue(), {}
priority_queue.put((0, source, [source]))
visited[source] = 0
while not priority_queue.empty():
(cost, vertex, path) = priority_queue.get()
if vertex == destination:
return cost, path
for next_node in GRAPH[vertex].keys():
current_cost = cost + GRAPH[vertex][next_node]
if not next_node in visited or visited[next_node] >= current_cost:
visited[next_node] = current_cost
priority_queue.put((current_cost, next_node, path + [next_node]))
```

```
def a_star(source, destination):
"""Optimal path from source to destination using straight line distance heuristic
:param source: Source city name
:param destination: Destination city name
:returns: Heuristic value, cost and path for optimal traversal
"""
# HERE THE STRAIGHT LINE DISTANCE VALUES ARE IN REFERENCE TO BUCHAREST AS THE DESTINATION
straight_line = {\
'Arad': 366,\
'Zerind': 374,\
'Oradea': 380,\
'Sibiu': 253,\
'Timisoara': 329,\
'Lugoj': 244,\
'Mehadia': 241,\
'Drobeta': 242,\
'Craiova': 160,\
'Rimnicu': 193,\
'Fagaras': 176,\
'Pitesti': 100,\
'Bucharest': 0,\
'Giurgiu': 77,\
'Urziceni': 80,\
'Hirsova': 151,\
'Eforie': 161,\
'Vaslui': 199,\
'Iasi': 226,\
'Neamt': 234\
}
```

```
from queue import PriorityQueue

priority_queue, visited = PriorityQueue(), {}
priority_queue.put((straight_line[source], 0, source, [source]))
visited[source] = straight_line[source]

while not priority_queue.empty():
(heuristic, cost, vertex, path) = priority_queue.get()
if vertex == destination:
return heuristic, cost, path
for next_node in GRAPH[vertex].keys():
current_cost = cost + GRAPH[vertex][next_node]
heuristic = current_cost + straight_line[next_node]
if not next_node in visited or visited[next_node] >= heuristic:
visited[next_node] = heuristic
priority_queue.put((heuristic, current_cost, next_node, path + [next_node]))

def main():
"""Main function"""
print('ENTER SOURCE :', end=' ')
source = input().strip()
print('ENTER GOAL :', end=' ')
goal = input().strip()
if source not in GRAPH or goal not in GRAPH:
print('ERROR: CITY DOES NOT EXIST.')
else:
print('\nALL POSSIBLE PATHS:')
paths = dfs_paths(source, goal)
for path in paths:
```

```
print(' -> '.join(city for city in path))
print('\nCHEAPEST PATH:')
cost, cheapest_path = ucs(source, goal)
print('PATH COST =', cost)
print(' -> '.join(city for city in cheapest_path))
print('\nOPTIMAL PATH:')
heuristic, cost, optimal_path = a_star(source, goal)
print('HEURISTIC =', heuristic)
print('PATH COST =', cost)
print(' -> '.join(city for city in optimal_path))

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

-→ →

```
GRAPH = {\
'Arad': {'Sibiu': 140, 'Zerind': 75, 'Timisoara': 118},\
'Zerind': {'Arad': 75, 'Oradea': 71},\
'Oradea': {'Zerind': 71, 'Sibiu': 151},\
'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu': 80},\
'Timisoara': {'Arad': 118, 'Lugoj': 111},\
'Lugoj': {'Timisoara': 111, 'Mehadia': 70},\
'Mehadia': {'Lugoj': 70, 'Drobeta': 75},\
'Drobeta': {'Mehadia': 75, 'Craiova': 120},\
'Craiova': {'Drobeta': 120, 'Rimnicu': 146, 'Pitesti': 138},\
'Rimnicu': {'Sibiu': 80, 'Craiova': 146, 'Pitesti': 97},\
'Fagaras': {'Sibiu': 99, 'Bucharest': 211},\
'Pitesti': {'Rimnicu': 97, 'Craiova': 138, 'Bucharest': 101},\
'Bucharest': {'Fagaras': 211, 'Pitesti': 101, 'Giurgiu': 90, 'Urziceni': 85},\
```

```
'Giurgiu': {'Bucharest': 90},\
'Urziceni': {'Bucharest': 85, 'Vaslui': 142, 'Hirsova': 98},\
'Hirsova': {'Urziceni': 98, 'Eforie': 86},\
'Eforie': {'Hirsova': 86},\
'Vaslui': {'Iasi': 92, 'Urziceni': 142},\
'Iasi': {'Vaslui': 92, 'Neamt': 87},\
'Neamt': {'Iasi': 87}\
}

def a_star(source, destination):
"""Optimal path from source to destination using straight line distance heuristic
:param source: Source city name
:param destination: Destination city name
:returns: Heuristic value, cost and path for optimal traversal
"""
# HERE THE STRAIGHT LINE DISTANCE VALUES ARE IN REFERENCE TO BUCHAREST AS THE DESTINATION
straight_line = {\
'Arad': 366,\
'Zerind': 374,\
'Oradea': 380,\
'Sibiu': 253,\
'Timisoara': 329,\
'Lugoj': 244,\
'Mehadia': 241,\
'Drobeta': 242,\
'Craiova': 160,\
'Rimnicu': 193,\
'Fagaras': 176,\
'Pitesti': 100,\
```

```
'Bucharest': 0,\
'Giurgiu': 77,\
'Urziceni': 80,\
'Hirsova': 151,\
'Eforie': 161,\
'Vaslui': 199,\
'Iasi': 226,\
'Neamt': 234\
}
from queue import PriorityQueue

priority_queue, visited = PriorityQueue(), {}
priority_queue.put((straight_line[source], 0, source, [source]))
visited[source] = straight_line[source]

while not priority_queue.empty():
(heuristic, cost, vertex, path) = priority_queue.get()
if vertex == destination:
return heuristic, cost, path
for next_node in GRAPH[vertex].keys():
current_cost = cost + GRAPH[vertex][next_node]
heuristic = current_cost + straight_line[next_node]
if not next_node in visited or visited[next_node] >= heuristic:
visited[next_node] = heuristic
priority_queue.put((heuristic, current_cost, next_node, path + [next_node]))

def main():
"""Main function"""
print('ENTER SOURCE :', end=' ')
```

```
source = input().strip()

print('ENTER GOAL :', end=' ')

goal = input().strip()

if source not in GRAPH or goal not in GRAPH:

print('ERROR: CITY DOES NOT EXIST.')

else:

print('\nALL POSSIBLE PATHS:')

paths = dfs_paths(source, goal)

for path in paths:

print(' -> '.join(city for city in path))

print('\nCHEAPEST PATH:')

cost, cheapest_path = ucs(source, goal)

print('PATH COST =', cost)

print(' -> '.join(city for city in cheapest_path))

print('\nOPTIMAL PATH:')

heuristic, cost, optimal_path = a_star(source, goal)

print('HEURISTIC =', heuristic)

print('PATH COST =', cost)

print(' -> '.join(city for city in optimal_path))


if __name__ == '__main__':

main()
```

➔ --> → →

```
from collections import deque


def h(n):

H = {
```

```
'Arad': 366,

'Zerind': 374,

'Sibiu': 253,

'Timisoara': 329,

'Lugoj': 244,

'Mehadia': 241,

'Drobeta': 242,

'Craiova': 160,

'Pitesti': 100,

'Giurgiu': 77,

'Urziceni': 80,

'Hirsova': 151,

'Eforie': 161,

'Vaslui': 199,

'Iasi': 226,

'Neamt': 234,

'Fagaras': 176,

'Rimnicu': 193,

'Oradea': 380,

'Bucharest': 0

}

return H[n]

class Graph:

def __init__(self, adjacency_list):
```

```
self.adjacency_list = adjacency_list

def get_neighbors(self, v):
return self.adjacency_list[v]

def a_star_algorithm(self, start_node, stop_node):

open_list = {start_node}
closed_list = set([])

g = {start_node: 0}

parents = {start_node: start_node}

while len(open_list) > 0:
n = None

for v in open_list:
if n is None or g[v] + h(v) < g[n] + h(n):
n = v

if n is None:
print('Path does not exist!')
return None

if n == stop_node:
reconst_path = []
cost = 0
while parents[n] != n:
```

```
reconst_path.append(n)
n = parents[n]

reconst_path.append(start_node)

reconst_path.reverse()

for c in reconst_path:
cost = g[c]

print('Path found: {}'.format(reconst_path))
print('Path Cost:', cost)
return reconst_path

for (m, weight) in self.get_neighbors(n):

if m not in open_list and m not in closed_list:
open_list.add(m)
parents[m] = n
g[m] = g[n] + weight

else:
if g[m] > g[n] + weight:
g[m] = g[n] + weight
parents[m] = n

if m in closed_list:
closed_list.remove(m)
open_list.add(m)
```

```
open_list.remove(n)
closed_list.add(n)

print('Path does not exist!')
return None
adjacency_list = {'Arad': [['Zerind', 75], ['Sibiu', 140], ['Timisoara', 118]],
'Sibiu': [['Arad', 140], ['Fagaras', 99], ['Rimnicu', 80], ['Oradea', 151]],
'Rimnicu': [['Sibiu', 80], ['Craiova', 146], ['Pitesti', 97]],
'Fagaras': [['Sibiu', 99], ['Bucharest', 211]],
'Pitesti': [['Rimnicu', 97], ['Craiova', 138], ['Bucharest', 101]]
}

graph1 = Graph(adjacency_list)

print('ENTER SOURCE :', end=' ')
source = input().strip()
print('ENTER GOAL :', end=' ')
goal = input().strip()

graph1.a_star_algorithm(source, goal)
# graph1.a_star_algorithm('Arad', 'Bucharest')
```

Eight_puzzle

```
import random
import itertools
import collections
import time

class Node:
"""
A class representing an Solver node
- 'puzzle' is a Puzzle instance
- 'parent' is the preceding node generated by the solver, if any
- 'action' is the action taken to produce puzzle, if any
"""
def __init__(self, puzzle, parent=None, action=None):
self.puzzle = puzzle
self.parent = parent
self.action = action
if (self.parent != None):
self.g = parent.g + 1
else:
self.g = 0

@property
def score(self):
return (self.g + self.h)
```

```
@property
def state(self):
"""
Return a hashable representation of self
"""
return str(self)

@property
def path(self):
"""
Reconstruct a path from to the root 'parent'
"""
node, p = self, []
while node:
p.append(node)
node = node.parent
yield from reversed(p)

@property
def solved(self):
""" Wrapper to check if 'puzzle' is solved """
return self.puzzle.solved

@property
def actions(self):
""" Wrapper for 'actions' accessible at current state """
return self.puzzle.actions
```

```
@property
def h(self):
"""h"""
return self.puzzle.manhattan

@property
def f(self):
"""f"""
return self.h + self.g

def __str__(self):
return str(self.puzzle)

class Solver:
"""
An '8-puzzle' solver
- 'start' is a Puzzle instance
"""
def __init__(self, start):
self.start = start

def solve(self):
"""
Perform breadth first search and return a path
to the solution, if it exists
"""
```

```
queue = collections.deque([Node(self.start)])
seen = set()
seen.add(queue[0].state)
while queue:
queue = collections.deque(sorted(list(queue), key=lambda node: node.f))
node = queue.popleft()
if node.solved:
return node.path

for move, action in node.actions:
child = Node(move(), node, action)

if child.state not in seen:
queue.appendleft(child)
seen.add(child.state)

class Puzzle:
"""
A class representing an '8-puzzle'.
- 'board' should be a square list of lists with integer entries 0...width^2 - 1
e.g. [[1,2,3],[4,0,6],[7,5,8]]
"""
def __init__(self, board):
self.width = len(board[0])
self.board = board

@property
```

```
def solved(self):
"""
The puzzle is solved if the flattened board's numbers are in
increasing order from left to right and the '0' tile is in the
last position on the board
"""
N = self.width * self.width
return str(self) == ''.join(map(str, range(1,N))) + '0'

@property
def actions(self):
"""
Return a list of 'move', 'action' pairs. 'move' can be called
to return a new puzzle that results in sliding the '0' tile in
the direction of 'action'.
"""
def create_move(at, to):
return lambda: self._move(at, to)

moves = []
for i, j in itertools.product(range(self.width),
range(self.width)):
direcs = {'R':(i, j-1),
'L':(i, j+1),
'D':(i-1, j),
'U':(i+1, j)}
```

```
for action, (r, c) in direcs.items():
if r >= 0 and c >= 0 and r < self.width and c < self.width and \
self.board[r][c] == 0:
move = create_move((i,j), (r,c)), action
moves.append(move)
return moves

@property
def manhattan(self):
distance = 0
for i in range(3):
for j in range(3):
if self.board[i][j] != 0:
x, y = divmod(self.board[i][j]-1, 3)
distance += abs(x - i) + abs(y - j)
return distance

def shuffle(self):
"""
Return a new puzzle that has been shuffled with 1000 random moves
"""
puzzle = self
for _ in range(1000):
puzzle = random.choice(puzzle.actions)[0]()
return puzzle

def copy(self):
```

```
"""
Return a new puzzle with the same board as 'self'
"""
board = []
for row in self.board:
board.append([x for x in row])
return Puzzle(board)

def _move(self, at, to):
"""
Return a new puzzle where 'at' and 'to' tiles have been swapped.
NOTE: all moves should be 'actions' that have been executed
"""
copy = self.copy()
i, j = at
r, c = to
copy.board[i][j], copy.board[r][c] = copy.board[r][c], copy.board[i][j]
return copy

def pprint(self):
for row in self.board:
print(row)
print()

def __str__(self):
return ''.join(map(str, self))
```

```
def __iter__(self):
for row in self.board:
yield from row



# example of use
board = [[1,2,3],[4,5,0],[6,7,8]]
puzzle = Puzzle(board)
#puzzle = puzzle.shuffle()
s = Solver(puzzle)
tic = time.clock()
p = s.solve()
toc = time.clock()

steps = 0
for node in p:
print(node.action)
node.puzzle.pprint()
steps += 1

print("Total number of steps: " + str(steps))
print("Total amount of time in search: " + str(toc - tic) + " second(s)")
```

prime_number.py

```
num = int(input("Number:"))

if num > 1:
for i in range(2,num):
if (num % i) == 0:
print(num,"Not a prime number")
print(i,"times",num//i,"is",num)
break
else:
print(num,"is a prime number")

else:
print(num,"is not a prime number")
```

Vacuum cleaner

```
def vacuum_world():
# initializing goal_state
# 0 indicates Clean and 1 indicates Dirty
goal_state = {'A': '0', 'B': '0'}
cost = 0

location_input = input("Enter Location of Vacuum") #user_input of location vacuum is placed
status_input = input("Enter status of " + location_input) #user_input if location is dirty or clean
status_input_complement = input("Enter status of other room")
print("Initial Location Condition" + str(goal_state))

if location_input == 'A':
# Location A is Dirty.
print("Vacuum is placed in Location A")
if status_input == '1':
print("Location A is Dirty.")
# suck the dirt  and mark it as clean
goal_state['A'] = '0'
cost += 1                    #cost for suck
print("Cost for CLEANING A " + str(cost))
print("Location A has been Cleaned.")

if status_input_complement == '1':
# if B is Dirty
print("Location B is Dirty.")
print("Moving right to the Location B. ")
```

```
cost += 1                    #cost for moving right
print("COST for moving RIGHT" + str(cost))
# suck the dirt and mark it as clean
goal_state['B'] = '0'
cost += 1                    #cost for suck
print("COST for SUCK " + str(cost))
print("Location B has been Cleaned. ")
else:
print("No action" + str(cost))
# suck and mark clean
print("Location B is already clean.")

if status_input == '0':
print("Location A is already clean ")
if status_input_complement == '1':# if B is Dirty
print("Location B is Dirty.")
print("Moving RIGHT to the Location B. ")
cost += 1                    #cost for moving right
print("COST for moving RIGHT " + str(cost))
# suck the dirt and mark it as clean
goal_state['B'] = '0'
cost += 1                    #cost for suck
print("Cost for SUCK" + str(cost))
print("Location B has been Cleaned. ")
else:
print("No action " + str(cost))
print(cost)
```

```
# suck and mark clean
print("Location B is already clean.")

else:
print("Vacuum is placed in location B")
# Location B is Dirty.
if status_input == '1':
print("Location B is Dirty.")
# suck the dirt  and mark it as clean
goal_state['B'] = '0'
cost += 1  # cost for suck
print("COST for CLEANING " + str(cost))
print("Location B has been Cleaned.")

if status_input_complement == '1':
# if A is Dirty
print("Location A is Dirty.")
print("Moving LEFT to the Location A. ")
cost += 1  # cost for moving right
print("COST for moving LEFT" + str(cost))
# suck the dirt and mark it as clean
goal_state['A'] = '0'
cost += 1  # cost for suck
print("COST for SUCK " + str(cost))
print("Location A has been Cleaned.")

else:
```

```
print(cost)
# suck and mark clean
print("Location B is already clean.")

if status_input_complement == '1':  # if A is Dirty
print("Location A is Dirty.")
print("Moving LEFT to the Location A. ")
cost += 1  # cost for moving right
print("COST for moving LEFT " + str(cost))
# suck the dirt and mark it as clean
goal_state['A'] = '0'
cost += 1  # cost for suck
print("Cost for SUCK " + str(cost))
print("Location A has been Cleaned. ")
else:
print("No action " + str(cost))
# suck and mark clean
print("Location A is already clean.")

# done cleaning
print("GOAL STATE: ")
print(goal_state)
print("Performance Measurement: " + str(cost))

vacuum_world()
```