hw3 az147

report on code testing by coverage analysis tool

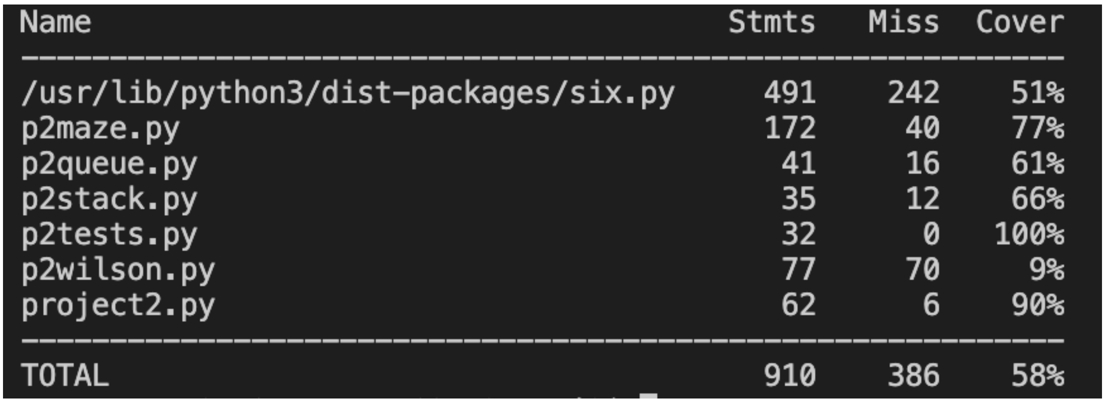
1. using my code from algorithm class implementing bfs and dfs solving the maz
2. using the tool coverage.py to analyze the code
3. main code:

```
"""
Math 560
Project 2
Fall 2021

project2.py

Partner 1: Aohua Zhang az147
Partner 2: none
Date: 11/2/2021
"""

# Import math and other p2 files.
import math
from p2tests import *

"""
BFS/DFS function

INPUTS
maze: A Maze object representing the maze.
alg:  A string that is either 'BFS' or 'DFS'.

OUTPUTS
path: The path from maze.start to maze.exit.
"""
def bdfs(maze, alg):
    # If the alg is not BFS or DFS, raise exception.
    if (alg != 'BFS') and (alg != 'DFS'):
        raise Exception('Incorrect alg! Need BFS or DFS!')

    ##### if the input is 'DFS', call the dfs algorithm to solve the maze
    #and return the path
    elif alg == 'DFS':
        path = dfs(maze)
    #else if the input is 'BFS', call the bfs algorithm to solve the maze
    #and return the path
    else:
        path = bfs(maze)
    #     path is a []
    return path
    ##### Your implementation goes here. #####

"""
getpath function

INPUTS
currentv: current vertex
start: start vertex
```

```
path: a list to store the path

OUTPUTS
none
"""
#implementation of BFS
def getpath(currentv, start, path):
    #
    if currentv.isEqual(start):
        path.append(currentv.rank)
        return
    #add the node to the path
    path.append(currentv.rank)
    #recursively call itself to get the next node in the path
    getpath(currentv.prev, start, path)
    return

"""
helpbfs function

INPUTS
exitv: exit vertex
pathqueue: a queque to push and pop vertices

OUTPUTS
bool value on whether find the exit node
"""
#helper functions of bfs
def helpbfs(exitv, pathqueue):
    #pop the queue and get the current node
    currvertex = pathqueue.pop()
    #if the queue is empty, return
    if currvertex == None:
        return
    #if current vertex is the exit vertex, return true to indicate that exit is found
    if currvertex.isEqual(exitv):
        return True
    #push all neighbour vetices to the queue and mark them as visited
    for v in currvertex.neigh:
        if not v.visited:
            v.prev = currvertex
            v.visited = True
            pathqueue.push(v)
    #recursive call and return true if the recuesive call returns true
    #(which means that have found the exit and go all the way back to start)
    if helpbfs(exitv, pathqueue):
        return True

"""
bfs function

INPUTS
maze: the maze to solve

OUTPUTS
path : the list storing the path from start to exit vertex
"""
#wrapper function of bfs, call the recursive helper with that start node and
#initialize path and getpath, then reverse path to get the right order.
def bfs(maze):
    pathqueue = Queue()
    maze.start.visited = True
    pathqueue.push(maze.start)
    path = []
    helpbfs(maze.exit, pathqueue)
    getpath(maze.exit, maze.start, path)
    path.reverse()
    return path

"""
helpdfs function

INPUTS
exitv: exit vertex
pathstack: the stack to push and pop the vertex
path: a list to store the path

OUTPUTS
none
"""
# dfs helper
def helpdfs(exitv, pathstack, path):
    #pop the node from stack and set as current vertex
    currvertex = pathstack.pop()
    #if find the exit return true
    if currvertex.isEqual(exitv):
        path.append(exitv.rank)
        return True
    #push the unvisited neighbour to the stack and mark as visited
    for v in currvertex.neigh:
        if not v.visited:
            v.prev = currvertex
            v.visited = True
            pathstack.push(v)
            #recursive call and return true if the recuesive call returns true
            #(which means that have found the exit and go all the way back to start)
            if helpdfs(exitv, pathstack, path):
                path.append(currvertex.rank)
                return True
    return

"""
dfs function

INPUTS
maze: the maze to solve

OUTPUTS
path: a list to store the path
"""
#wrapper function of dfs, which call the dfshelper and get the path in right order and then return
def dfs(maze):
    pathstack = Stack()
    maze.start.visited = True
    pathstack.push(maze.start)
    path = []
    helpdfs(maze.exit, pathstack, path)
    path.reverse()
    return path




"""
Main function.
"""
if __name__ == "__main__":
    testMazes(True)
    # m = Maze(5, True) #getMaze(1, 12)
    # m.solve('BFS')
```

4. results analysis

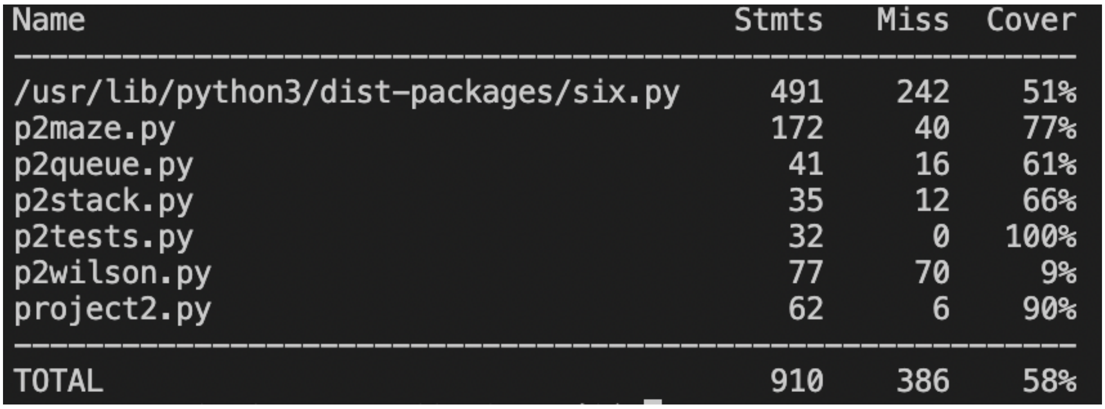
    a. statement coverage



```
Name                                     Stmts   Miss  Cover
------------------------------------------------------------
/usr/lib/python3/dist-packages/six.py      491    242    51%
p2maze.py                                  172     40    77%
p2queue.py                                  41     16    61%
p2stack.py                                  35     12    66%
p2tests.py                                  32      0   100%
p2wilson.py                                 77     70     9%
project2.py                                 62      6    90%
------------------------------------------------------------
TOTAL                                      910    386    58%
```

the main testing object is project2.py, and the statement coverage is 90%

    b. branch coverage



```
Name                                     Stmts   Miss  Cover
------------------------------------------------------------
/usr/lib/python3/dist-packages/six.py      491    242    51%
p2maze.py                                  172     40    77%
p2queue.py                                  41     16    61%
p2stack.py                                  35     12    66%
p2tests.py                                  32      0   100%
p2wilson.py                                 77     70     9%
project2.py                                 62      6    90%
------------------------------------------------------------
TOTAL                                      910    386    58%
```

the main testing object is project2.py, and the branch coverage is also 90%

    c. function coverage

function coverage is 100%, for the main testing object is project2.py, and all four functions have been called in the testing process.

    d. parts of your code that are hard to cover and discuss why.

        i. the corner cases parts are ones that hard to cover, because we always encounter with normal and general cases, which won’t trigger the part of code written for corner cases.